Java Au Naturel

Guide to Object Oriented Design

Java 2 with Swing

Dr. William C. Jones, Jr.
email jonesw@ccsu.edu
Department of Computer Science
Central Connecticut State University
web page: www.cs.ccsu.edu/~jones/book.htm
printed 2/17/02

iii

Java Au Naturel
Table Of Contents

Chapter 8 Elementary Graphics 34 pages

Chapter 9 Event-Driven Programming 43 pages

Chapter 10 Exception-Handling 37 pages

total: 689 pages plus preface and contents

Preface for Students

Software development with an object-oriented approach is the fundamental subject of this book. Java is a programming language used to create the animations you see when you browse the web. Of all the programming languages whose use is wide-spread, Java is the best for learning and doing object-oriented software development. That is why Java is used in this computer science book.

You only learn by doing. So in this book, you will work with many different situations where software development is necessary. That way, you can see how to apply newly-learned techniques in a variety of contexts. When you need a language feature to accomplish a certain purpose, that is when you learn it. If it has several alternatives and you do not need them at that point or in the near future, you do not learn them then; you wait until you need them.

When you learn to speak a natural language such as German or Spanish, you start with a few sentences that are useful in key situations. You gradually expand your repertoire of sentences. You learn grammatical principles that apply to several sentences you already know how to use.

You do not start by learning all the conjugations of verbs and all the declensions of nouns. Incremental development is far more effective: Add items a few at a time, mastering those before you add more. And, most important, learn what you can immediately use in realistic conversations.

You should learn a programming language the same way. You should not start by learning Java's eight different primitive types or its five different control statements. You should start by seeing how to develop software for a particular realistic situation. And you should learn the language features you need for the situation as the need arises.

This book starts by introducing you to objects that make drawings. In this context you learn to send messages to objects to carry out simple tasks. More important, you learn how to "teach" those objects to put together a sequence of simple actions to perform a complex task. These "better-trained" objects have new capabilities in addition to the ones they inherit from the original specifications. This **inheritance** technique of course makes your job much easier. You can do bigger jobs with less work and less chance of getting it wrong if you have objects take over most of the work. When objects are used this way, they should be thought of as your agents or executive assistants.

Chapter One explains the details of compiling and executing a program. It introduces much of the vocabulary you need. It establishes the framework for Java programs. And it gives an overview of the book with a look at other contexts where objects are used to perform useful tasks. This is a foretaste of what is to come, so you are not expected to fully understand at this point everything it mentions.

Chapter Two introduces a quite different software context. The software provided to you defines objects that control the basic physical actions of electronic equipment. Your job is to develop additional software that puts these basic actions together in combinations that perform tasks that the purchaser of the electronic equipment finds useful. You learn to use inheritance in this new context. Then you see how to have objects select between two courses of action depending on the circumstances. You also learn a simple method of diagramming the relationships among classes of objects. It is a widely used technique that is part of the Unified Modeling Language (UML). The UML is the industry standard for modeling software.

Chapter Three shows you how to teach objects to repeat a sequence of actions many times until a task is accomplished. This ability leads to more complex programs, so we discuss a reliable process for developing the logic to solve a problem and translate it into a Java program. This is the one chapter that does not introduce a new major context for developing software.

Chapter Four switches to the context of game-playing programs. In this context, you learn how to build objects "from scratch", specifying what they know as well as what they can do. By this time you can write interesting and useful programs using only the standard library of objects that comes with every Java installation. The game-playing software interacts with the outside world through the keyboard and screen rather than via signals to and from electronic equipment or to a drawing surface.

Chapter Five completes the presentation of basic language features you need for working with objects in your programs. At this point you can construct a complete string-based simulation of the electronic equipment you worked with in earlier chapters. A case study on networks lets you see the interaction of all of the object-oriented concepts and most of the language features applied in another context. The situation that the networking material describes is key to many important real-life problems. The analysis and solution of some of these problems can be quite complex; it is the subject of more advanced courses in computer science. But the discussion here is quite elementary.

Chapter Six expands your arsenal of basic types of values to include characters and numbers with decimal points (heretofore you only had the whole-number and true-false kinds of values). It also gives you a full set of methods for dealing with strings of characters. These language features are introduced in the context of software to schedule work orders at a car repair shop using the Model/View/Controller approach and several kinds of objects at once.

Chapter Seven gives you the tools you need to work with large masses of data. This is in the context of software to handle a database of people working for a particular company. A re-implementation of a simulation of the electronic-equipment software from Chapters Two and Three helps you solidify your understanding of the key concept of arrays.

The first seven chapters present the features of Java most frequently used in this book, together with a moderate number of examples. Many of the concepts, especially the various uses of arrays, cannot be learned well enough without a great deal of practice. The remaining ten chapters give you that practice. Their primary purpose is to (a) improve your understanding of principles and techniques of software design and development, and (b) reinforce the concepts in the basic first seven chapters.

Each of those last ten chapters presents a different software design and development situation. The emphasis is on techniques for creating quality software. Each of Chapters Eight through Eleven presents only a few new Java language features. Chapters Twelve and later do not present any new language features.

In summary

- Chapter One: Turtle objects can do simple tasks such as draw a line or move to another position. You use Java to combine these simple actions into complex tasks such as having a Turtle draw a flower garden.
- Chapters Two and Three: Stick figures called Vic objects can do simple tasks such as put a compact disk in or out of the current slot and move to the slot before or after the current slot. You use Java to combine these simple actions into complex tasks that move several CDs where you want them.

- Chapters Four and Five: You use Java to define the simple tasks that objects can do. You apply these language skills to define how game-playing objects interact with a human player, and to define how the stick figure objects perform their simple tasks. You may also, if you wish, skip directly to Chapter Eight on graphics to learn to define how Turtle objects perform their simple tasks.
- Chapter Six: You use Java to develop commercial software that defines how several classes of objects perform simple tasks and then has these several classes of objects cooperate to accomplish a complex task.
- Chapter Seven: You use Java for software handling many objects at once.

Re-read this summary once or twice as you progress through the textbook, whenever you get the feeling that you cannot see the forest for the trees.

You cannot avoid heavy technical vocabulary in a first course in computer science. But as a student you have a right to short paragraphs and simple sentences to explain the vocabulary and the difficult concepts accurately. This book makes a special effort to give you just that.

You should work out most of the unstarred exercises in the book. Most should take less than ten minutes. The answers to them are at the end of the corresponding chapter. This will help you gain a thorough understanding of the concepts rather than a superficial one. This book does not require any knowledge of programming or any mathematics beyond elementary algebra and a bit of trig. But it does require that you practice what you read about. Reading alone will not suffice -- you only learn by doing!

1 Objects

This chapter presents an overview of object-oriented programming in Java. Sections 1.1 through 1.4 show you how to control an object that can draw pictures. This object is called a turtle because it is based on the Logo programming language, in which a turtle icon does the drawing.

You will learn to create turtle objects and to send them messages to take actions. A turtle understands only eight elementary kinds of messages, but Java lets you teach the turtle new messages that are combinations of existing ones. You supply the artistic ability by deciding which messages to send in what order; the turtle carries out your requests.

The real purpose of the first part of this chapter, however, is to give you the opportunity to write programs that will impress your friends and relatives. Naturally, you cannot expect to be able to create an interesting program from scratch until you have been studying computer science for several weeks. But you can download the three-page turtle software from this book's website or type it in from the listings in Chapter Eight. Then, with the help of these turtle objects, you can create a program that draws complex pictures. When your friends and relatives ask you what you have learned to do in the course, you will have something good to show them by the end of your first week.

In this context, you will learn to define executable Java programs and to define instance methods without parameters or return values. Section 1.5 explains in detail how to compile and execute your programs in the terminal window.

Later sections give examples of programs using other kinds of objects. One is a program that sends messages to portfolio objects to perform financial-market tasks. Another is a program that sends messages to other kinds of objects to perform personnel database tasks. We even include a very simple program developed in Java from scratch, using only facilities that come with every Java compiler. This foretaste requires showing you language features whose full explanation is in Chapters Two through Four, so you are not responsible for remembering these features at this point.

1.1 Using Turtle Objects To Draw Pictures

Turtle objects are derived from the Logo programming language, which has been around for decades. When you run a Java program that uses Turtle objects, you have a drawing surface on which a Turtle is positioned, initially facing due East. The Turtle is carrying a bucket of paint and a paint brush (too tiny for you to see on the screen). The Turtle moves one little Turtle step at a time (these steps are also known as **pixels**). The three most useful requests that a Turtle understands are named `paint`, `move`, and `swingAround`. Examples of their use are as follows:

- `move(45,20)` sends a message that causes the Turtle to turn 45 degrees to its left and then walk 20 little Turtle steps, without leaving any marks. This message is used to move the Turtle from one position to another on the drawing surface.
- `paint(90,30)` sends a message that causes the Turtle to turn 90 degrees to its left and then walk 30 little Turtle steps, dragging its paintbrush behind it. This message is used to draw a straight line on the drawing surface, going off at the specified angle from the current position.
- `paint(-60,50)` sends a message that causes the Turtle to turn 60 degrees to its right and then walk 50 little Turtle steps, drawing a line as it goes. That is, a negative number for the angle causes a right turn; a positive number for the angle causes a left turn; and zero for the angle causes no turn at all. This applies to `move` as well.

- `swingAround(100)` sends a message that causes the Turtle to put the paintbrush on a rope and swing it around its head, thereby drawing a circle. The length of the rope is given as 100 Turtle steps (pixels). So the Turtle paints a circle of radius 100 with itself at the center.

Objects in Turtle programs

The programs to control the computer chip are loaded from a permanent storage space, such as a hard disk, into RAM (Random Access Memory). RAM storage gives very fast access to the programs and the data they use. Each time you run the program, it loads from permanent storage again and starts afresh.

A program that uses Turtles must create an internal description of each Turtle and put that data in RAM. This internal description is an **object**; it records the Turtle's current position and heading. The phrase `new Turtle()` in a program creates the Turtle object.

The phrase `Turtle sam` in a program sets aside a part of the RAM's data area, called a **variable**, which can refer to a Turtle object (the internal description of a drawing instrument). The phrase `Turtle sam` also **declares** that `sam` is the name for that variable. It is difficult to tell a particular Turtle to carry out some action if the Turtle does not have a name you can use. So Turtle programs often contain a sequence such as the following, which lets later parts of the program refer to the Turtle as `sam`:

```
Turtle sam;
sam = new Turtle();
```

When the Turtle is first created, it is in the center of the drawing surface, facing due East and carrying a can of black paint. The drawing surface is 760 pixels wide and 600 pixels tall (because some computer monitors are not much larger than that in size). Drawings outside of that range will not appear on the drawing surface.

Illustration of basic Turtle messages

Figure 1.1 shows how the drawing looks after a newly-created Turtle receives the three messages `paint (90, 7); move (0, 2); paint (0, 2);` in that order. The tiny figure indicates the position of the Turtle, and the Turtle's head indicates its heading. Each `paint` message colors in the pixel the Turtle starts on but not the one it ends up on.

If for instance `sue` refers to some Turtle, `sue.move(0,7)` is what you write in a Java program to tell that Turtle to move 7 little Turtle steps straight ahead (without first turning), and `sue.paint(180,30)` is what you write to tell that Turtle to turn completely around and then draw a line 30 little Turtle steps long. Later in this chapter you learn Turtle messages that write words and change the color of parts of the drawing. But we keep things simple for now with just these three kinds of messages.

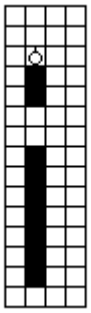


Figure 1.1 the effect of paint(90,7); move(0,2); paint(0,2);

Exercise 1.1 Write a sequence of messages that causes a Turtle named `sue` to draw a rectangle twice as wide as it is tall, with a height of 60 pixels.
Exercise 1.2 Write a sequence of messages that causes a Turtle named `sam` to draw a lowercase letter 'r' 12 pixels tall and 8 pixels wide. Include the angled part of the 'r'.

1.2 A Complete Java Application Program Using The Basic Four Turtle Methods

You write a program to perform a complex task by breaking the task down into a combination of simple actions. An instruction to a Turtle to take an action is a **command**. In a program, you need a semicolon at the end of each command in a sequence of commands. The command and semicolon together are called a **statement**.

Suppose you want a Turtle to draw the capital letter 'H', 12 pixels tall and 6 pixels wide, with a circle around it. The following sequence of statements can do this. The remark at the right of each command explains the meaning of the command:

```
Turtle sam;            // declare the variable named sam
sam = new Turtle();    // create the object sam refers to
sam.paint (90, 12);    // draw the left side of the H
sam.move (-180, 6);    // return to the center of the H
sam.paint (90, 6);     // draw the crossbar of the H
sam.move (90, -6);     // move to the bottom of the right side
sam.paint (0, 12);     // draw the right side of the H
sam.move (150, 6);     // go to just above the center of the H
sam.swingAround (9);   // draw a circle enclosing the H
```

The `//` symbol in a program indicates a **comment**. That symbol is a signal that everything on the rest of its line is to be ignored. Comments are only for humans to read; they do not affect the operation of a program.

The structure of an application program

The preceding sequence of nine statements describes a method for accomplishing a task. Before you can have the computer follow this method of doing things, you have to give the method structure. In Java, the way you do this is to put those statements between matching left and right **braces** `{` and `}` and put the following heading above them:

```
public static void main (String[ ] args)
```

You have then constructed a **main method**. The words in the heading of the main method have meanings that will be explained in the next section. This section tells you what you do to make a program; the next section tells you why you do it. That way, you have an overview of the entire process before going into the details.

Some programs need hundreds of methods in order to do what they need to do. It would be very difficult to keep track of all of them, their meanings and relationships, if they were not organized in some reasonable way. The primary organizing unit in Java is called a **class** (for reasons that will become clear later in this chapter). The general idea is, you collect several methods together that are very closely related to each other and put them in a single class. For instance, `paint`, `swingAround`, `move`, and others are collected in the Turtle class.

In Java, the way you signal that a number of methods belong to a particular class is to put the method(s) between matching left and right braces `{` and `}` and put a heading like the following above them, although you have a free choice of the name you use in place of SomeClass:

```
public class SomeClass
```

A class containing a main method is usually called an **application program**. This book does not put any method in an application program other than the main method. Listing 1.1 shows a complete Java application program using the preceding sequence of nine statements to draw a letter 'H'.

Listing 1.1 An application program using a Turtle object

```
public class ProgramOne
{
   // Draw an uppercase letter 'H', 12 pixels tall and 6 wide.
   // Put a circle around the outside of the 'H'.

   public static void main (String[ ] args)
   {  Turtle sam;            // create the variable named sam
      sam = new Turtle();    // create the object sam refers to
      sam.paint (90, 12);    // draw the left side of the H
      sam.move (-180, 6);
      sam.paint (90, 6);     // draw the crossbar of the H
      sam.move (90, -6);
      sam.paint (0, 12);     // draw the right side of the H
      sam.move (150, 6);
      sam.swingAround (9);   // draw a circle enclosing the H
   }  // this right-brace marks the end of the main method
}     // this right-brace marks the end of the class
```

Programming Style Comments (the parts after the `//` symbols) are optional in a program, but you should always have at least a comment before the main method describing the purpose of the program. You should also make sure that each right brace is lined up vertically with the corresponding left brace, and that everything between the braces is indented by one tab position.

The result of executing ProgramOne is shown in Figure 1.2. Note that the capital 'H' actually spans a total of seven pixels horizontally, since `paint` colors in the pixel the Turtle starts on but not the pixel the Turtle ends on. The Turtle finishes just above the middle of the 'H' facing south-south-west, indicated by the tiny circle.

Edit, compile, and run

First you type the lines of this program into a plain-text file named `ProgramOne.java` (because ProgramOne is the name of the class) using a word processor. The next thing you do is submit this text file named `ProgramOne.java` to the **compiler** program to see if it is correctly expressed in the Java language. On the simplest systems, you do this by entering `javac ProgramOne.java` at the prompt in a **terminal window** (a window that only allows plain text input and output; often called the DOS window).

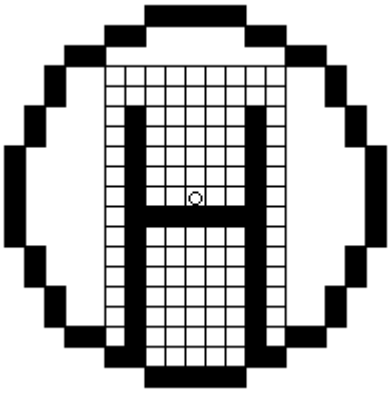


Figure 1.2 After execution of Listing 1.1

If the compiler does not detect any errors when you have it check out your program, it translates your text file to an executable file named `ProgramOne.class`. It does not translate anything after the `//` symbol on a line.

You then run the program by entering `java ProgramOne` at the prompt in the terminal window. This tells the **runtime system** to carry out the commands in the executable file. You also need a compiled form of the Turtle class, which is available on this book's website (or you could type it in from Section 8.11). This Turtle class lets you test the programs you write.

Caution You must be careful in programs to capitalize letters in words exactly as shown. The compiler program sees `paint` and `Paint` and `PAINT` all as three totally unrelated words, i.e., Java is case-sensitive. Of those three words, Turtles only understand `paint`.

If you replaced `sam` by `sue` in every statement of Listing 1.1, it would make no difference in the effect of the program. The choice of the name `sam` for the Turtle object is arbitrary, as long as you spell and capitalize it the same way throughout your program.

Application program to draw two squares

If you want a Turtle to draw two squares side by side, you could have the runtime system execute the program in Listing 1.2. The first two statements create the Turtle object and position it in the center of the drawing area facing East. They also make `sue` contain a reference to that Turtle object.

Listing 1.2 An application program using a Turtle object

```
public class TwoSquares
{
   // Draw two 40x40 squares side by side, 10 pixels apart.

   public static void main (String[ ] args)
   {  Turtle sue;
      sue = new Turtle();

      sue.paint (90, 40);    // draw the right side of square #1
      sue.paint (90, 40);    // draw the top of square #1
      sue.paint (90, 40);    // draw the left side of square #1
      sue.paint (90, 40);    // draw the bottom of square #1

      sue.move (0, 50);      // move 50 pixels to the right
      sue.paint (90, 40);    // draw the right side of square #2
      sue.paint (90, 40);    // draw the top of square #2
      sue.paint (90, 40);    // draw the left side of square #2
      sue.paint (90, 40);    // draw the bottom of square #2
   }  //=====================
}
```

After the Turtle object is created, the next four statements draw the first square, 40 pixels on a side. The last five statements of the program draw the second square of the same size, 10 pixels away from the first square. Figure 1.3 shows the status of the Turtle object and the drawing after execution of this program.

The order of the commands is important: You cannot send a message to `sue` to perform an action before you store a Turtle object in the variable named `sue` (statement #2), and you cannot do that before you declare that variable (statement #1).

the Turtle starts at the lower-right corner of the lefthand square, facing east,
and ends at the lower-right corner of the other square, facing east.

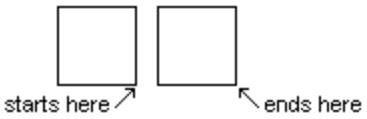


Figure 1.3 After execution of Listing 1.2

The meaning of some program elements

You are surely wondering why you need `public` and `void` and all the rest to make a simple program. Each of the parts of a Java program has a purpose. The following paragraphs, while not a full explanation, should give you some idea of what the purpose is. These paragraphs also preview what you will see in the first half of this book.

A **method** is so called because it describes the method by which some objective is achieved. For instance, the main method in Listing 1.2 describes a method of drawing two squares.

Question: Why the matched pair of braces? *Answer:* They tell the compiler where a class begins and ends and where a method begins and ends. This is needed because you can have more than one class within a file and you can have more than one method within a class. You will see an example of the latter in the next section.

Question: Why must one declare `Turtle sue` when the next command `sue = new Turtle()` makes it quite clear that `sue` is a Turtle variable? *Answer:* You will sometimes misspell a name. If the compiler were to accept the misspelled name as a different variable, that could cause hard-to-find errors in your programs. But because Java requires you to explicitly declare all names, you have little trouble finding misspellings; the compiler points them out to you.

Question: Why the word `public` in the headings? *Answer:* An alternative is `private`. If the main method were private, it could not be used by anything outside the class. The terminal window is outside the class. So the runtime system cannot execute the main method from the terminal window unless the main method is public. Similarly, the class should be public instead of private so it can be used by anything outside the class. You will see private methods in Chapter Three.

Question: Why the word `static` in the method heading? *Answer:* When a method heading does not include `static`, the compiler will not let you use the method unless you first create an object to send the message to. At the time the `java` command in the terminal window executes the main method, the runtime system has not yet created any object to send the message to. So the main method must be marked `static`. You will see other kinds of methods marked `static` in Chapter Five.

Question: Why the word `void` in the method heading? *Answer:* When you send a message, sometimes you get an answer back to use later in the program, and sometimes you do not. The word `void` signals that no answer will be sent back by this particular method. Since there is no "later in the program" after the main method is executed, the main method should be marked `void`. You will see non-void methods in Chapter Two.

Question: Why "void" instead of say "noAnswerGiven"? Why "main" instead of "programStartsHere"? Why semicolons instead of commas? *Answer:* The designers of Java decided that, where the choice was quite arbitrary, they would use the symbols and signals from the C programming language, because most professional programmers are familiar with it.

Question: Why the `(String[ ] args)` part? *Answer:* It can be used to get the user's input (though it does not do so in this particular program). For instance, you may have a Euchre-playing program you start by entering the basic `java Euchre` command in the terminal window followed by two extra words. You can start it by entering `java Euchre 4 English`; it then allows four players and communicates in English. Or you can start it by entering `java Euchre 3 French`, so it allows three players speaking French. The runtime system uses the `(String[ ] args)` part of the main heading to send those two extra pieces of information to the main method.

For now, it would not hurt to treat that phrase as just one of those things you have to have to make things work right in Java. It is like the "ne" in the French phrase "ne pouvez pas"; the "pas" means "not", but for some reason you have to tack on a "ne" to be speaking correct French.

Exercise 1.3 Write an application program that creates one Turtle and has it draw a lowercase 'b' 6 pixels wide and 12 pixels tall, without going over the same pixel twice.
Exercise 1.4 Write an application program that creates one Turtle and has it draw a lowercase 'm' 8 pixels wide and 6 pixels tall.
Exercise 1.5 Write an application program that creates one Turtle and has it draw a hexagon 50 pixels on a side.
Exercise 1.6* Write an application program that creates one Turtle and has it draw a lowercase 'g' 6 pixels wide and 6 pixels tall, descending 3 pixels below the baseline, without going over the same pixel twice.
Exercise 1.7* Write an application program that creates a Turtle and then draws a house 200 pixels wide and 150 pixels tall, with a door and two windows.
Note: A star on an exercise means the answer is <u>not</u> in the book. Unstarred exercises have the answers at the end of the corresponding chapter.

1.3 A First Look At Inheritance: Defining Instance Methods In Turtle Subclasses

You can expect to draw a square in several different programs, with various Turtle objects receiving that sequence of four `paint(90,40)` messages. This was done twice for `sue` in Listing 1.2. Fortunately, you can invent new messages for the Turtle that are combinations of existing messages. This will simplify your programs.

For instance, you can define a new message named `makeBigSquare`:
`sue.makeBigSquare()` tells `sue` to execute those four `paint(90,40)` actions, and
`sam.makeBigSquare()` tells `sam` to execute those four `paint(90,40)` actions.

You may also want to draw a small square in several different situations. The sequence of four messages

```
paint (90, 10);
paint (90, 10);
paint (90, 10);
paint (90, 10);
```

could be quite common, sent to various Turtle objects. You can define a new message named `makeSmallSquare`: `sam.makeSmallSquare()` tells `sam` to carry out those four `paint(90,10)` actions, and `sue.makeSmallSquare()` tells `sue` to carry out those four `paint(90,10)` actions.

A simple Turtle object does not know the meaning of the two words `makeBigSquare` and `makeSmallSquare`. They are not part of its vocabulary. You need a new class of

objects that can understand these two messages plus all the messages a Turtle understands. Let us call this new kind of Turtle object a SmartTurtle. Then you could rewrite the main method in Listing 1.2 to do exactly the same thing but with a simpler sequence of statements, as follows:

```
public static void main (String[ ] args)
{  SmartTurtle sam;
   sam = new SmartTurtle();
   sam.makeBigSquare();
   sam.move (0, 50);
   sam.makeBigSquare();
}  //=====================
```

How to define a class of objects

You may define a class that provides new messages and a new kind of object that understands those messages. The class definition in Listing 1.3 says that, if you create an object using the phrase `new SmartTurtle()` instead of `new Turtle()`, that object will understand the `makeBigSquare` and `makeSmallSquare` messages as well as all of the usual Turtle messages. In a sense, a SmartTurtle object is better educated than a basic Turtle object.

Listing 1.3 The SmartTurtle class of objects

```
public class SmartTurtle extends Turtle
{
   // Make a 10x10 square; finish with the same position/heading.

   public void makeSmallSquare()
   {  paint (90, 10);
      paint (90, 10);
      paint (90, 10);
      paint (90, 10);
   }  //=====================

   // Make a 40x40 square; finish with the same position/heading.

   public void makeBigSquare()
   {  paint (90, 40);
      paint (90, 40);
      paint (90, 40);
      paint (90, 40);
   }  //=====================
}
```

A class definition that extends the capabilities of a Turtle object must have the heading

```
public class WhateverNameYouChoose extends Turtle
```

followed by a matched pair of braces that contain some definitions. The SmartTurtle class definition contains two parts beginning `public void`. These two parts are **method definitions**. The names chosen here are `makeBigSquare` and `makeSmallSquare`, but they could be anything you choose. Just be sure that, when you send these messages to an object, as in `sam.makeBigSquare()` or `sue.makeSmallSquare()`, you always spell them the way the method definition shows, including capitalization, and finish with the empty pair of parentheses.

Each of the two method definitions in the SmartTurtle class describes one new message in terms of previously-known messages. In the heading of the method definition, `public` means that the statements in any class can send these methods to its SmartTurtles. `void` means that these are definitions of actions to be taken instead of questions to be answered; you will see how to use and define questions in the next chapter. The comments following the right braces (beginning with `//`) are dividers that help visually separate the various method definitions within the class definition.

Within the definition of this kind of method, you leave out the name of the variable that refers to the object that receives the message. This lets you use any variable name you like (such as `sue`, `sam`, or whomever) when you write the command outside the method. So the `makeSmallSquare` definition says that for any `x`, if you have previously defined `x = new SmartTurtle()`, then `x.makeSmallSquare()` has the same meaning as the following:

```
x.paint (90, 10);
x.paint (90, 10);
x.paint (90, 10);
x.paint (90, 10);
```

You can read the second method definition verbally as follows: Define an action method named `makeBigSquare` that any SmartTurtle object in any class can be asked to carry out. Using this method sends a message asking the object to draw the four sides of a square 40 pixels on a side, ending up with the same position and heading as at the start.

Programming Style You will find it much easier to understand a class definition you have written if you mark the end of each method with a comment that stands out clearly. This book puts "`==========`" at the end of each method definition. Some people prefer the phrase "End of method" or something more specific such as "End of makeBigSquare".

Executors and instances

An object created by `new Turtle()` is called an **instance** of the Turtle class, and an object created by `new SmartTurtle()` is called an **instance** of the SmartTurtle class.

You cannot use either of the two SmartTurtle methods of Listing 1.3 in an application program without mentioning an instance of the SmartTurtle class in front of the method name, separated from it by a dot (period). So these two methods are called **instance methods**. The absence of the word `static` in the heading signals this restriction -- a main method is not an instance method.

A command of the form `someObject.someMessage()` is a **method call**. We say that the object referenced before the dot is the **executor** of the method, since it executes the commands in the method. Every call of an instance method requires an executor.

This vocabulary applies to Turtle methods as well as to SmartTurtle methods. So `sue` refers to the executor in the method call `sue.swingAround(30)`. All three of `paint`, `move`, and `swingAround` require an executor when they are used. So these three are instance methods of the Turtle class. [Technical Note: Java has no official terminology for what this book calls the "executor"; other names some people use are "target object", "receiver", and "implicit parameter".]

The object that executes the `paint(90, 40)` commands inside the definition of `makeBigSquare` **defaults** to the executor of the `makeBigSquare` method call, since the executor of those commands is not explicitly stated. By contrast, you cannot call an instance method from the main method without stating its executor, because the main

method itself has no executor to default to. The word `static` in the heading of the main method signals that it has no executor.

Listing 1.4 illustrates the use of these new SmartTurtle commands in a program that makes an X-shaped pattern of one large square with four small squares at its four corners. Figure 1.4 shows the result of running this program.

Listing 1.4 An application program using a SmartTurtle object

```
public class SquarePattern
{
   // Make an X shape with one big 40x40 square in the center
   // and a small 10x10 square in each corner.

   public static void main (String[ ] args)
   {  SmartTurtle sue;
      sue = new SmartTurtle();
      sue.makeBigSquare();      // draw the center square

      sue.move (-90, 15);       // go south
      sue.move (90, 15);        // move to the southeast corner
      sue.makeSmallSquare();    // draw the southeast square

      sue.move (90, 70);        // move to the northeast corner
      sue.makeSmallSquare();    // draw the northeast square

      sue.move (90, 70);        // move to the northwest corner
      sue.makeSmallSquare();    // draw the northwest square

      sue.move (90, 70);        // move to the southwest corner
      sue.makeSmallSquare();    // draw the southwest square
   }  //====================
}
```

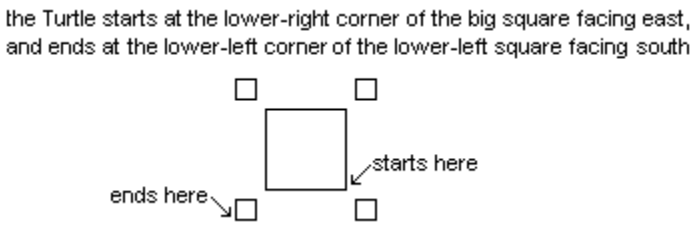


Figure 1.4 After execution of SquarePattern

Inheritance from the superclass

The SmartTurtle class in the earlier Listing 1.3 contains the `makeBigSquare` method definition and the `makeSmallSquare` method definition explicitly. The phrase `extends Turtle` in the heading of the class definition means the SmartTurtle class indirectly contains all the public method definitions it gets from the Turtle class. We say that SmartTurtle objects **inherit** the Turtle methods (`paint`, `move`, etc.).

Inheritance is what allows any SmartTurtle object to use all of the regular Turtle methods in addition to those directly defined in the SmartTurtle class. In short, a SmartTurtle is a kind of Turtle. Turtle is the **superclass** and SmartTurtle is the **subclass** in this inheritance relationship. You will see examples of inheritance in most of the chapters of this book.

When you decide what methods you need in a subclass of Turtle to accomplish a task, you are doing **object design**. Designing useful objects "from scratch" requires knowing a large number of language features, so it must wait until Chapter Four. Until then, our object classes will extend interesting classes such as Turtle.

"Object orientation, involving encapsulation, inheritance, polymorphism, and abstraction, is an important approach in programming and program design. It is widely accepted and used in industry and is growing in popularity in the first and second college-level programming courses. It facilitates the reuse of program components and the management of program complexity, allowing large and complex programs to be written more effectively and efficiently and with more confidence in their correctness than with the more traditional purely procedural approach." [AP Computer Science Ad Hoc Committee Recommendations, October 2000]

Programming Style It is good style to use indentation and spacing in a program to make the relationship of one line of the program to another clear. The convention in Java is to indent one tab position at each line inside the braces of a class definition. Indent another tab position at each line that is also inside a method definition. If you want to separate statements into groups that do separate tasks, do not use indentation to do so. Instead, use a blank line between the groups, as shown in Listing 1.4.

Exercise 1.8 Write an application program that uses a SmartTurtle to draw two large squares side by side, each with a small square centered inside it.
Exercise 1.9 Write a new instance method to be added to the SmartTurtle class: It has the executor carry out the last two commands of Listing 1.4. Then rewrite Listing 1.4 to call this new method three times, thereby shortening the logic.
Exercise 1.10 Write a `drawHexagon` instance method to be added to the SmartTurtle class: The executor draws a hexagon 30 pixels on a side in just six statements.
Exercise 1.11 Write an application program that uses a SmartTurtle object, as changed by the preceding exercise, to draw three hexagons such that any two of them meet along one side.
Exercise 1.12* Write a StarTurtle class with two instance methods: One draws a five-point star (hint: turn 144 degrees) and one draws a six-point star (two overlapping equilateral triangles with symmetry). Make them 60 pixels per line segment.
Exercise 1.13* Write an application program that uses a StarTurtle object, as defined in the preceding exercise, to make an interesting drawing with at least five stars in it.
Exercise 1.14* Write an application program that draws ten big squares in the same arrangement as the setup for bowling pins. Use a SmartTurtle object.

1.4 Additional Turtle Methods; Identifiers Versus Keywords

The Turtle carries ten cans of paint of various colors, not just one. If you want to switch to e.g. red, you can do so with the following message:

```
sam.switchTo (Turtle.RED).
```

Thereafter, all drawings are made in red (until you switch to another color). The ten available colors are BLACK, GRAY, BLUE, GREEN, RED, YELLOW, ORANGE, PINK, MAGENTA, and WHITE. You use `Turtle.WHITE` when you want to erase something you drew earlier; this helps you do animations.

You have to spell the names of the colors entirely in capitals, because that is how they are defined in the Turtle class. You put "Turtle" in front of each color name so that the compiler knows to look for the definition of the color name in the Turtle class. But within an instance method in a subclass of Turtle, you can use the color names without the class name, just as you can use method names without mentioning the executor.

The Turtle class has four more kinds of messages that you can send to a Turtle object:

- `fillCircle(80)` is the same as `swingAround(80)` except that the circle is completely filled with whatever the current drawing color is.
- `fillBox(20,90)` draws a rectangle of width 20 and height 90 with the Turtle at the center, and fills it in with whatever the current drawing color is.
- `say("whatever")` prints what you have in quotes at the Turtle's current location.
- `sleep(70)` causes the Turtle to stop what it is doing for 70 milliseconds (0.070 seconds). This command lets you control the speed of animations.

These new methods do not change the position and heading of the object. If a number in parentheses is not positive, the methods simply do nothing. By contrast, the `move` and `paint` methods do something useful when the numbers within their parentheses are negative or zero: A negative angle indicates a turn to the right, and a negative distance indicates a Turtle walking backwards.

A program using a FlowerMaker

Listing 1.5 contains an application program that creates a FlowerMaker kind of Turtle and has it draw six flowers in a row, centered on the drawing surface. A FlowerMaker is capable of drawing two flowers next to each other when you ask it to, 60 pixels apart. After each pair of flowers the Turtle pauses for 300 milliseconds (0.3 seconds), because that makes the drawing a little more interesting (hopefully). Then it prints a message above the row of flowers. Figure 1.5 shows the result of executing this program.

Listing 1.5 An application program using most of the new Turtle methods

```
public class GardenApp
{
   // Draw 6 flowers all in a row, with a word title.

   public static void main (String[ ] args)
   {  FlowerMaker florist;
      florist = new FlowerMaker();
      florist.drawTwoFlowers();    // the central two
      florist.sleep (300);

      florist.move (0, 120);
      florist.drawTwoFlowers();    // the two right of center
      florist.sleep (300);

      florist.move (0, -240);
      florist.drawTwoFlowers();    // the two left of center
      florist.sleep (300);

      florist.move (40, 130);
      florist.switchTo (Turtle.BLUE);
      florist.say ("My flower garden");  // above the flowers
   }  //=====================
}
```

Figure 1.5 After execution of GardenApp

The FlowerMaker class

The class definition in Listing 1.6 says that objects declared as FlowerMakers have the ability to understand the three new messages there in addition to all Turtle messages. The `drawTwoFlowers` method in the top part of Listing 1.6 draws one flower, moves to the right 60 pixels, draws a second flower, and then returns to the original starting point and resumes the original heading. To do this right, you need to know that the `drawFlower` method leaves the Turtle one pixel to the left of where it started, facing south instead of east.

Listing 1.6 The FlowerMaker class of objects

```
public class FlowerMaker extends Turtle
{
   // Draw two flowers each 60 pixels tall.
   // Start and end facing east at the base of the left flower.

   public void drawTwoFlowers()
   {  drawFlower();
      move (90, 61);
      drawFlower();
      move (90, -59);
   }  //=====================

   // Starts facing east at the base of the flower, right side,
   // with the current drawing color being BLACK (for the stem).
   // Ends facing south at the base of the flower, center.

   public void drawFlower()
   {  paint (90, 50);    // right side of stem
      paint (90, 2);
      paint (90, 50);    // left side of stem
      paint (90, 1);
      paint (90, 10);    // one-fourth of the way up the stem
      paint (-45, 8);    // draw the twig for the right leaf
      drawLeaf();

      paint (45, 10);    // one-half of the way up the stem
      paint (45, 8);     // draw the twig for the left leaf
      drawLeaf();

      paint (-45, 30);   // to top of stem, in the center
      switchTo (RED);
      fillCircle (15);   // draw the flower petals
      switchTo (BLACK);
      move (180, 50);    // return to the base of the flower
   }  //=====================

   public void drawLeaf()
   {  switchTo (GREEN);
      fillCircle (3);
      move (0, 3);
      fillCircle (2);
      move (0, 2);
      fillCircle (1);
      move (0, -13);
      switchTo (BLACK);
   }  //=====================
}
```

A Turtle that executes the `drawFlower` method is initially facing east, assuming that the flower is to grow to the north, which normal flowers do. The Turtle starts by drawing a thick BLACK stem 3 pixels wide. As it makes the middle of the three strokes for the stem, it stops 10 pixels up to make a leaf off to the right, then stops again 20 pixels up to make a leaf off to the left. Then it draws a RED circle at the top of the stem to represent the flower. Finally, it returns to the base of the stem, ending up facing south.

A Turtle that executes the `drawLeaf` method draws three overlapping GREEN circles, each smaller than the one before. That will hopefully look rather like a leaf. Then it moves back to the base of the 8-pixel-long twig and switches the drawing color back to BLACK, which is what it was when the method was called. Note that the return to the base is made using `move(0,-13)` rather than `move(180,13)`, so that the Turtle's heading remains as it was initially.

The definition of `drawFlower` is in terms of `drawLeaf`, and the definition of `drawTwoFlowers` is in terms of `drawFlower`. You may define a method in any terms the object can understand. In particular, you may define a method in the FlowerMaker class as a sequence of messages any Turtle object or any FlowerMaker object can understand.

Your first thought is probably that this allows you to do something really silly, such as define `drawOne` to mean two messages of `drawTwo` and define `drawTwo` to mean two messages of `drawOne`. Yes, you can do that. If you then sent either of those messages to an object, it would cause the program to crash. It is your responsibility to avoid such silliness; objects are not smart enough to know when you make a logic error.

Two kinds of methods and classes

You have now seen two kinds of method definitions: A main method is a sequence of instructions normally initiated from the terminal window. The main method is normally called by `java Whatever` in the terminal window, where the Whatever class contains that main method. Its heading must be `public static void main (String[ ] args)` (except `args` could be any name you like). A method without `static` in the heading is an instance method, specifying a sequence of actions carried out by its executor. An instance method is typically called by putting an object value in front of the method name, separated by a dot. That object value must refer to an instance of the class the method belongs to, or a subclass of that class.

A method definition has two parts: the **method heading** (everything up to but not including the first left brace) and the **method body** (the matched pair of braces and its contents). For instance, the heading of the main method in Listing 1.5 is the line that begins with `public static` and ends with the right parenthesis; the body of that method is the thirteen statements that follow, together with the enclosing braces.

The heading `public class Whatever` signals a **class definition**. You have now seen two kinds of class definitions: An application program is a class that contains a main method. An **object class** is a class that contains one or more public instance methods, such as the SmartTurtle class or the FlowerMaker class, and no main method. These instance methods define what messages individual objects of that class can "understand". Most classes are object classes, i.e., they define a new class of object (hence the name "class").

Identifiers and keywords

The name you choose for a method, variable or class is its **identifier** (e.g., `makeLeaf`, `sam`, and `SmartTurtle`). You can use letters, digits, and underscores to form an identifier: `Flower_maker` and `Bring_3_back` are permissible names.

Caution You cannot have a blank within an identifier or have a digit as its first character. A very common mistake is to start a program with something like `public class Program Two`. A blank in the middle of the class name is not allowed.

You cannot change the spelling or capitalization of any of the **keywords** in a program, such as `public`, `class`, `extends`, `static`, `void`, and `new`. You never use capital letters for them, and you never name a method, variable, or class with one of them.

The Java developers at Sun Microsystems Inc. have developed a library of over one thousand classes for use by Java programmers. It comes with the standard installation of Java that Sun provides. It does not include the Turtle class -- that was developed for this book. We begin the serious use of the **Sun standard library** in Chapter Four, and introduce over 150 of those classes by the end of Chapter Fifteen.

The word "String" in the heading of the main method is the name of a class in the Sun standard library. The people who wrote the String class could choose whatever name they wanted; but now that they have, you have to spell it and capitalize it exactly the same way they did. Otherwise your program will not compile. Similarly, the word "Turtle" was chosen arbitrarily by the developer of the Turtle class, and you have to spell it that way to use it.

Programming Style The convention in Java is to name all classes starting with a capital letter and all methods and non-constant variables starting with a lowercase letter. It is good style to keep to that convention. You should also use **titlecase** for names -- capitalizing only the first letter of each word within a name (as in `useItNow`) -- or underscoring (as in `use_it_now`).

It is good programming style to choose a name for a method, variable, or class that conveys its meaning. Note, for instance, that the object variable in Listing 1.5 is named `florist`, so you can easily remember that it draws flowers. The names `sue` and `sam` have been used only for objects that are not particularly distinguishable.

Language elements

Beginning in Chapter Two, sections that introduce new features of the Java language usually conclude with a formal description of the new language elements. The following is an example of such a description for most of what you have seen so far in this chapter:

Language elements

A CompilableUnit can be: public class ClassName { DeclarationGroup }
or: public class ClassName extends ClassName { DeclarationGroup }
A DeclarationGroup is any number of consecutive Declarations.
A Declaration can be: public static void main (String [ ] args) { StatementGroup }
or: public void MethodName ( ) { StatementGroup }
A StatementGroup is any number of consecutive Statements.
A Statement can be: ClassName VariableName ; e.g., Turtle sam;
or: VariableName = new ClassName ( ) ; e.g., sam = new Turtle();
or: VariableName . MethodName ( ) ; e.g., sam.drawFlower();

These descriptions are compact, but they can be difficult to make sense of at times (examples of a principle often clarify the principle better than an explicit statement of the principle). We will analyze the notation used in this particular description so that you will be better able to understand similar descriptions later in this book. This notation is not used anywhere in the book except in these special descriptions and the language review at the end of Chapter Five.

Lines 1 and 2: The basic unit in a programming language is a CompilableUnit, i.e., the contents of a text file that you can submit to a compiler and have translated to an executable form. Line 1 says that one kind of CompilableUnit in Java is three words followed by material in a matched pair of braces. The three words are the two unalterable keywords "public class" followed by any name you choose for the class being defined. The material in braces is any number of Declarations you would like to have. Line 2 says that, in the class heading, you can insert just after the name of the class the word "extends" followed by the name of a superclass from which it inherits.

Lines 4 and 5: You have seen two kinds of Declaration so far, described in these two lines (you will see others later). The first line describes the structure of a main method declaration as shown in Listing 1.5. The second line describes the structure of an instance method declaration as shown in Listing 1.6; the choice of the MethodName is arbitrary for an instance method.

The notation used here should be clearer now. With the special exception of the heading of the main method:

- A word that begins with a capital letter and ends in "Name" indicates an identifier; you can have any name you choose for that category of declaration.
- A word that begins with a capital letter and does not end in "Name" indicates a language construct that is defined elsewhere.
- Any other item means that item itself must appear there.

Lines 7 through 9: Here you have descriptions of the three kinds of statements you have seen so far (there are many more). Examples are given at the right of each description. Line 7 says a statement may declare a variable to have a particular VariableName and be able to store a reference to a particular instance of the class named ClassName. Line 8 describes the general format of a statement that assigns a reference to a newly-created instance of the class named ClassName to a variable named VariableName. Line 9 describes the general format of a statement that calls a method named MethodName with an executor referred to by the variable named VariableName.

Exercise 1.15 Revise the `drawTwoFlowers` method to have the second flower not only 60 pixels to the right of the first but also 20 pixels higher than the first.
Exercise 1.16 Write a `drawSmallFlower` method to be added to the FlowerMaker class: The executor makes a tiny flower 10 pixels tall with no leaves.
Exercise 1.17 How many statements would the `drawTwoFlowers` method have if it did exactly the same thing but did not call on any other methods outside the Turtle class (so the statements from `drawFlower` and `drawLeaf` are in `drawTwoFlowers`)?
Exercise 1.18 Write an application program that draws a target: a solid black circle inside a solid blue circle inside a solid yellow circle inside a solid red circle.
Exercise 1.19* Revise the `drawFlower` method to have four leaves on alternating sides, branching off at 10, 15, 20, and 25 pixels up the stem.
Exercise 1.20* Revise the `drawFlower` method to have six smaller circular yellow petals distributed around the outside of the red center, overlapping it somewhat.
Exercise 1.21* Write an application program that draws five circles of different colors and sizes at a variety of points on the drawing surface. Pause half a second between circles.
Exercise 1.22* Describe all the kinds of situations you have seen within a class in which you use parentheses.
Exercise 1.23** In Listing 1.5, which ones of the first twelve statements could be swapped with the statement directly following it without changing the effect of the program? List all of them that can be swapped.
Note: Double-starred exercises are the hardest ones. The answers are not in the back.

1.5 Compiling And Running An Application Program

This section goes into the technical details of writing, compiling, and running a program. First, make a folder on your hard disk for these Turtle programs and others you might use. Then copy all the files from this book's website or CD-ROM disk that end in `.java` into that one folder so you can use them.

The following describes what you do in one common situation, using a Windows operating system and the free JDK you can download off the Internet. Your programming environment may be different. Even if so, you should be aware of what is going on behind the scenes, as described below.

You would try out ProgramOne as follows, assuming the program folder is `c:\cs1` and the JDK has been properly installed with path settings:

1. Obtain the terminal window: Under Windows, click Start, then click Programs, then click MS-DOS Prompt or Command Prompt or something equivalent. You should see a terminal window with probably the `C:\windows>` prompt.
2. Switch to the program folder: Type the command `cd C:\cs1` in the terminal window and press the Enter key. You should see the `C:\cs1>` prompt. If the programs from the book are in this folder, one of them is in a file named `Turtle.java` and another is in a file named `ProgramOne.java`.
3. Compile Turtle: Type `javac Turtle.java` and press the Enter key. You should see the `C:\cs1>` prompt reappear after a short wait. This indicates the compiler has translated `Turtle.java` into `Turtle.class`.
4. Compile ProgramOne: Same as Step 3 except use `javac ProgramOne.java`.
5. Run that program: Type `java ProgramOne` and press the Enter key. You should see the Turtle window appear. ProgramOne from Listing 1.1 will run, drawing a capital letter 'H'.

Say you run GardenApp from Listing 1.5. The runtime system **links in** the FlowerMaker class it mentions. This causes the runtime system to link in the Turtle class FlowerMaker mentions, which causes it to link in the graphics classes that the Turtle class mentions. This is done automatically for you if all Turtle-related classes are in the same disk folder.

Computer storage

A computer has two kinds of places to store data. One is RAM, which gives very fast access to the data. However, when the computer is turned off, the data in RAM is lost. And when a program runs in a window and that window is closed, the data in RAM is lost. So RAM is called volatile memory.

The other kind of place to store data is permanent storage, such as the computer's hard disk, a floppy disk, or a CD-ROM. Data stored in these places is not lost even when the computer is turned off. However, the computer chip needs much more time to get data from permanent storage than to get it from RAM. The components of the computer (chip, RAM, disk, etc.) are **hardware**; the programs that control the computer are **software**.

Writing and running your own program

When you have worked out the logic for your own program, type it in a WordPad editor window (or any word-processor program with the spell/grammar check turned off). Save it in permanent storage, in the `c:\cs1` folder as a text file with the name `ProgramMine.java` (except change ProgramMine to whatever your class's name is). You may need quotes around the name the first time you save it; otherwise the word processor may add an extra word (e.g. ".txt") to the name that makes it uncompilable.

Once your program is saved on disk, open the terminal window alongside your editor window. Compile your program using `javac ProgramMine.java`. If the compiler detects errors, it prints descriptive messages in the terminal window. Correct them in the editor window, click Save, then compile the program again in the terminal window. Some general problems that can occur in compiling are as follows:

- Getting more than 100 error messages generally means you saved the file with all of the formatting codes that your word-processor uses for margins, fonts, etc. Go back and this time save it right, as an ordinary text file.
- "Bad command or file name" usually means your javac compiler is not properly installed including the path.
- "Can't read: X.java" usually means you have probably misspelled the name of the file, in capitalization if nothing else. The DOS command `dir p*` will list all of the files you have in the directory that start with the letter p, and similarly for other letters. Use the `dir` command to see what the true file name is.
- "Public class X must be defined in a file called X.java" means just what it says: The name of the file must match exactly the name of the class, except for the ".java" part.

Source code versus object code

When you have a successful compile (no error message before the prompt reappears), the compiler program translates the contents of your `ProgramMine.java` file, called the **source code**, into a form called the **object code**, stored on disk in a file named `ProgramMine.class`. You can then execute the object code in the file by entering `java ProgramMine`. This loads the program into RAM and begins its execution.

Downloading the JDK

If your computer system has not been set up properly for using Java, it may not recognize the `javac` or `java` command. You may download the JDK (which includes the compiler) from `http://java.sun.com/products`. Then enter the following three commands each time you open the terminal window for compiling or running (check your disk folders to see what is the exact name of the folder containing the javac program; it may not be jdk1.3.2):

```
DOSKEY
set path=c:\jdk1.3.2\bin;%path%
set classpath=.
```

The `DOSKEY` command makes it possible to use the up-arrow to get back commands you entered in the terminal window earlier, which saves you some typing. If you know how to do so safely, you could add those three commands to the end of your `autoexec.bat` file. Of course, if you use a commercial software development package instead of the free JDK download, the process for writing, compiling, and running programs is different. Some other free Java development packages may be available at one of `http://www.bluej.org`, `http://www.realj.com`, `http://www.netbeans.org`., or `http://www.eng.auburn.edu/grasp`.

Caution The most common compiler errors beginning Java programmers make are: (a) capitalizing the "p" in `public`; (b) not capitalizing the "s" in `String`; (c) forgetting the pair of parentheses at the end of each message; and (d) forgetting the semicolon at the end of each statement. The most common non-compiler error is failure to make a backup copy of the source code files on floppy disk every hour or so.

Exercise 1.24* Find on your computer the folder that contains the file named `javac.exe`, the Java compiler.

1.6 Sending Messages To Objects

It is essential you understand the concept of sending messages to objects. Think of it this way: Each object is a person you can contact over the Internet. You can send email messages to them and receive email answers from them.

- When you declare `Turtle sam` in a program, that creates space for an entry in your email address book; `sam` is the name of the entry space.

- When you execute `sam = new Turtle()` in a program, that puts the email address of a particular person in the entry space with the name `sam`. Now your program can communicate with that person (or turtle).

- When you execute `sam.move(30,50)` in a program, that sends an email message to the person whose address is stored with the name `sam`. The message is a request to the person to move from one place to another. The subject line is `move` and the body of the email is `30,50`, which describes how to move. The `move` message does not get a response from the recipient. It simply produces a change in the state of the object to whom the message was sent. So do `paint` and `switchTo`.

- When you execute `pat.drawFlower()` in a program, having previously executed `pat = new FlowerMaker()`, that sends a message to the person whose address is stored with the name `pat`. The message is a request to the person to carry out the commands given in the definition of the `drawFlower` method. Nothing is in the body of the email. Only a FlowerMaker can understand this message; the compiler will not let you send it to a plain uneducated Turtle such as `sam`.

- When you put `bill = sam` in a program, having previously declared `Turtle bill`, that copies the address in `sam` into the entry space named `bill`. Then `sam` and `bill` contain the same address, so they refer to the same object. If you send a message to `sam`'s object, you are perforce sending a message to `bill`'s object, and vice versa. We will not have occasion to do this for Turtle objects, but it is quite useful for some other kinds of objects you will see later.

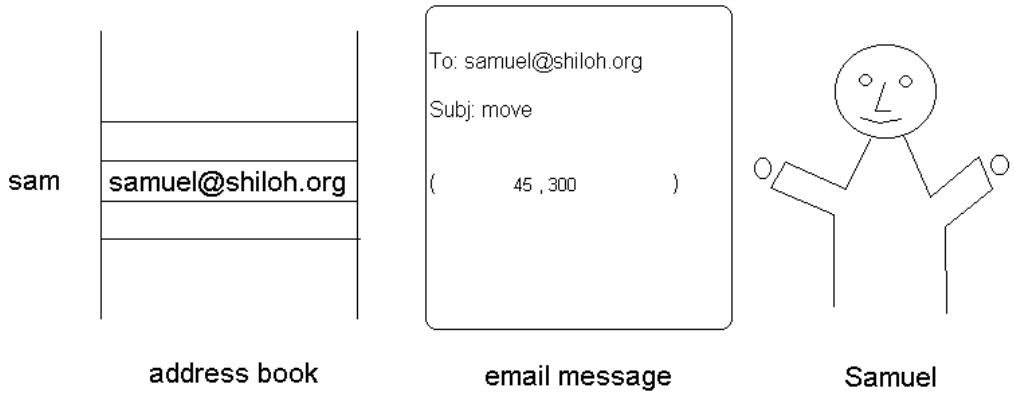


Figure 1.6 Sending an email message to a person

Remember, this is all just a metaphor. Technically, the runtime system creates space for all of a method's variables when the method begins execution.

Key Point: An object variable contains the address of an object, not the object itself. Putting `java.sun.com` in your address book is clearly not the same as putting the entire Sun Microsystems company itself in your address book.

Order of execution

When method X sends a message to an object O, X suspends all operations until O notifies X it is finished doing everything the message asked it to do. Specifically:

- X passes control of the execution of the program to O at the time the message is sent.
- does whatever it is supposed to do; X is doing nothing at this point, just waiting.
- returns control of execution back to X when O is finished.
- X executes the next operation after the messaging operation.

For instance, the message `florist.drawTwoFlowers()` in Listing 1.5 means the main method passes control of execution to `florist` and waits patiently until `florist` has drawn the two flowers. The main method does not send its next message, `florist.sleep(300)`, until after `florist` has drawn the two flowers and returned control to the main method.

If it were not pedagogically unsound to cascade metaphors, we would say you could think of it this way: X calls O up on the telephone and asks O to do something; O puts X on hold while he does it; O takes X off hold to report he has done it; X hangs up.

Variables versus values

There is a sharp difference between a variable and the value stored in it. You can have a variable refer to two different objects at different times, or you can have two different variables refer to the same object at the same time. This is illustrated by the following sequence of statements (although we normally try to avoid having two different variables refer to the same object within the same method):

```
Turtle pat;
pat = new Turtle();     // pat refers to Turtle object #1
pat.move (90, 50);      // that object is now north of the center

Turtle chris;
chris = pat;            // 2 variables refer to Turtle object #1
chris.move (90, 50);    // that object is now northwest of the center

pat = new Turtle();     // pat refers to a different Turtle object #2
pat.move (90, 50);      // Turtle object #2 is now north of the center
```

Exercise 1.25** Explain in your own words the difference between an object and a reference to the object.

1.7 Three Application Programs Using Other Kinds Of Objects

Future chapters present objects that are more complex to work with than Turtles. Several applications involving them are briefly described here so you can see how other kinds of objects are used. You should be able to get the general idea of what they do, although the details will have to wait until later chapters.

Evaluating the stock market

Suppose some investors want to know the range of possible outcomes if they invest in the stock market. That is, they are too smart to just bet on the average performance over the past few decades; they want to know reasonable estimates of the best and the worst possible outcomes for their investments over the next twenty years or so.

Listing 1.7 is an application program to model the financial markets and simulate their behavior over a period of twenty years. It is a slight variant of software you will learn to develop in Chapter Ten. The first statement declares an object variable named `wealth` for a portfolio of investments, a combination of stocks and bonds. The second statement creates a model of the portfolio and makes `wealth` refer to that model. The third statement prints out a description of the various mutual funds the portfolio is invested in.

Listing 1.7 An application program using a BuyAndHoldPortfolio object

```
public class InvestFor20
{
   public static void main (String[ ] args)
   {  BuyAndHoldPortfolio wealth;                          // 1
      wealth = new BuyAndHoldPortfolio();                  // 2
      wealth.describeInvestmentChoices();                  // 3
      wealth.waitForYears (20);                            // 4
      wealth.displayCurrentValues();                       // 5
   }  //=====================
}
```

The fourth statement simulates the action of the financial markets every day for the next twenty years and keeps track of the changes in the values of the portfolio. The last statement displays the results at the end of the twenty years.

You could run this program many times and write down the highest and lowest outcomes obtained. Or the program itself can be revised to run 100 simulations and then print out the lowest and highest end results obtained. (Note: You cannot actually run the three programs in this section until after you study the later chapters, because you do not have the object classes the programs need).

The investment program has the same structure as a Turtle program: You declare an object variable (`sam` or `wealth`), have it refer to a newly-created object, then send the object whatever messages will accomplish the task to be performed by the program.

Combining two files into one

Suppose you have two separate files on a hard disk that contain retail sales information from two different cash registers. You need to combine them for further computer processing. Specifically, you need an application program that combines two files on the hard disk into one file so that all the lines from the first file (named "firstData.txt") are

followed by all the lines from the second file (named "secondData.txt") in a new file named "combined.txt".

Listing 1.8 is an application program to do just that. You will learn how to write such programs in Chapter Twelve. The first two statements declare an object variable of the IO class, named `inputFile`, and have it refer to a representation of the physical file "firstData.txt". The IO class is built from Sun's standard library classes for disk files.

The next two statements declare an object variable of the FileOutputStream class, named `outputFile`, and have it refer to a representation of the physical combined file (replacing any existing file on the hard disk of the same name, creating a new one if needed). The FileOutputStream class is one of Sun's standard library classes that comes with the free download of JDK.

Listing 1.8 An application program using three disk file objects

```
public class Combiner
{
   public static void main (String[ ] args)
           throws FileNotFoundException    // explained in Ch 12
   {  IO inputFile;                                          // 1
      inputFile = new IO ("firstData.txt");                  // 2

      FileOutputStream outputFile;                           // 3
      outputFile = new FileOutputStream ("combined.txt");    // 4

      inputFile.copyTo (outputFile);                         // 5
      inputFile = new IO ("secondData.txt");                 // 6
      inputFile.copyTo (outputFile);                         // 7
      outputFile.close();                                    // 8
   }  //====================
}
```

The effect of the last sequence of four statements is as follows:

1. Statement #5 sends a message to the `inputFile` object to print all of its information to the file represented by the `outputFile` object.
2. Statement #6 changes the value of the `inputFile` variable so it refers to a representation of a completely different disk file named "secondData.txt".
3. Statement #7 sends the new object a message to print all of its information to the file represented by the `outputFile` object; this information is added after the information from "firstData.txt".
4. Statement #8 sends the combined file referred to by `outputFile` the message that it has been completed, which causes some clean-up activities.

The quotes around "firstData.txt" mean this is the name of a physical file on the hard disk. The runtime system uses the name to find an existing file. If you did not have the quotes, the compiler would interpret it as the name of a variable. That would not be acceptable, because you have not declared the variable or given it a value.

This program would be more useful if you could use it for any three files, not just the ones named. That could be done with the `String[] args` part: You would put something involving `args` in place of the three names in quotes, then run the program using

```
java Combiner firstData.txt secondData.txt combined.txt
```

(If it were not far too early to introduce this detail now, we would tell you to just put `args[0]` in place of "firstData.txt", put `args[1]` in place of "secondData.txt", and put `args[2]` in place of "combined.txt" within the main method.)

Finding the highest-paid worker

Suppose a company stores information about all the current employees in a hard-disk file named "workers.txt". The number of employees in the company fluctuates weekly, but it is not expected to exceed 2800 or so. The company needs an application program that prints out the name of the highest-paid employee and what he/she is paid.

Listing 1.9 is an application program to do just that. Chapter Seven shows how to define a WorkerList object, which is a list of all the workers in the company, and how to write this program. The first two statements create a WorkerList object that can store up to 3,000 employees, each represented as a separate Worker object, and have `company` refer to the WorkerList object. The third statement sends a message to the `company` object to read in the information about the employees from the hard-disk file named "workers.txt".

Listing 1.9 An application program using WorkerList and Worker objects

```
public class FindHighest
{
   public static void main (String[ ] args)
   {  WorkerList company;                                  // 1
      company = new WorkerList (3000);                     // 2
      company.addFromFile ("workers.txt");                 // 3

      Worker highestPaid;                                  // 4
      highestPaid = company.getWorkerWithHighestPay();     // 5
      System.out.println (highestPaid);                    // 6
   }  //=====================
}
```

The fourth and fifth statements ask the object named `company` to find the Worker object representing the employee with the highest pay and then store the object in a Worker variable named `highestPaid`. The last statement sends a message to an object to print a line in the terminal window describing the employee.

The object that prints a line is named `out`. Note that the program does not declare that object variable. That is because another class named System already declares the object variable named `out` and assigns it a value. Your class can use the variable as long as it makes it clear to the compiler where to find it. The compiler will not look for such variables outside of the class you are in unless you explicitly tell it to, by giving the name of the other class before the name of the variable.

A prototype application using only Sun library classes

Sometimes you want to test out part of a software system before you have the whole system completed. An executable program that does only a small part of what the final product is intended to do is a **prototype**. Many of its methods may be incomplete. A common technique is to have such methods simply print a message to the terminal window that it was called. Consider the following statement:

```
System.out.println ("whatever");
```

It prints whatever is within the quotes to the terminal window. Using this statement, you can make a BuyAndHoldPortfolio class for the InvestFor20 application that allows you to run the program. This class is in Listing 1.10. It, together with the InvestFor20 class in Listing 1.7, forms a complete working program without relying on anything other than the Sun standard library. But it does not do much. By the time you complete Chapter Four, you will know how to write complete programs that do much more interesting tasks.

Listing 1.10 A prototype of the BuyAndHoldPortfolio class of objects

```
public class BuyAndHoldPortfolio
{
   public void describeInvestmentChoices()
   {  System.out.println ("You have 5 ways to invest.");
   }  //=====================

   public void waitForYears (int years)
   {  System.out.println ("Wait for your money to grow.");
   }  //=====================

   public void displayCurrentValues()
   {  System.out.println ("You now have lots of money.");
   }  //=====================
}
```

Objects

You have seen four different concrete examples of object-oriented programming so far. Now you can look at what is common to them and to other similar situations.

Object-oriented software does its job primarily by sending messages to various objects. These messages call on the services provided by the objects. A class definition says what services an object of the class provides, defined as methods.

For the Turtle software, the objects represent the point of a pen that draws figures in colors. For the InvestFor20 software, the objects represent mutual funds and the portfolio as a whole. For the Combiner software, the objects represent physical files on a hard disk. For the WorkerList software, the objects represent the company as a whole and the individual workers.

The effect of a given message on different instances from the same class can vary. This is because each object stores individual information about itself. A Turtle object stores its own heading and coordinates. A Worker object stores its name, birth date, and week's pay. The values of the stored data affect what the object does in response to a message.

An analogy with CEOs

In olden times (thirty years ago), programmers developed software in a way that kept all of the information stored out in the open. A programmer had to keep track of perhaps hundreds of variables, not just a few Turtle or WorkerList variables. This was often overwhelming, and it limited the size and complexity of software that could be developed in a reasonable time. It is analogous to what happens in a private company run by a hands-on owner. When the owner supervises most of the details of everything that gets done, the company cannot grow past a certain modest size.

For object-oriented programming, the programmer creates objects that store the information. These objects supply or modify the information when requested via method calls. This gives the programmer much less to keep track of, and so larger and more

complicated software can be developed efficiently. Leaving most of the work to individual objects is a classic use of the principle of delegation of responsibility.

Such a programmer acts more like a CEO of a large company than a hands-on owner of a small company. For instance, the CEO may have an assistant do a substantial part of the job and report the result, which the CEO passes on to two other assistants who between them complete the job. The CEO is there only to determine the overall structure of the solution to a problem and make sure that individual subtasks are given to assistants who can do the task well.

A CEO who needs a task done looks for an employee with the right qualifications. Similarly, when you design larger programs, you will look for classes of objects you already have in the Sun standard library or in your own library that can do the task, i.e., you try to **reuse software**. The CEO who has no suitable employee for a given task sees if one can easily be trained in the needed skills. Similarly, you may find that you need to add new methods to existing classes of objects (i.e., subclass them) to get your tasks done. The CEO who has no employee who can be retrained goes out and hires a new one who can do the job. Similarly, you will often find you need to create new classes of objects to perform the tasks that a piece of software needs to do.

Models and simulation

Objects in software typically provide a **model** of a portion of the real world. For instance, a Turtle object is a model of a hand holding a pen or crayon. Software for operating a bank may have many Account objects, SafeDepositBox objects, Teller objects, and Customer objects that model part of the bank operations. And software for finding efficient ways of manufacturing furniture may have many WoodShaper objects, Assembler objects, and Finisher objects that model people with specific tasks.

Execution of the software typically provides a **simulation** of reality. Simulation is defined in Merriam Webster's Collegiate Dictionary as: "a. the imitative representation of the functioning of one system or process by means of the functioning of another; b. examination of a problem often not subject to direct experimentation by means of a simulating device." That device would be the computer, or more precisely, a program running on the computer.

The simulation may be profitable because it can be done by the computer hundreds of times faster than in reality, or because it avoids the destruction of material and the use of manpower that reality would require. The computer does not shape real wood into furniture, only virtual wood (objects) into virtual furniture (more objects).

An important way in which computer models and simulations affect us all is through the use of software to model the weather around the world and simulate its development over the next few days. This software is what makes your daily weather report possible so you can plan your activities in the near future. It also provides fairly reliable predictions of severe weather that can kill people and damage property if no one is prepared for it. The objects used are Storm objects, Cloud objects, JetStream objects, and the like.

In general, an application program often works with a model of a real situation. A model contains various elements that represent parts of the model. These elements (objects) can be grouped into classes according to the kind of behavior they exhibit -- objects with the same kinds of behaviors are grouped into the same class. For instance, Turtles draw pictures; BuyAndHoldPortfolios of stocks make money; FileOutputStreams store textual data; and Workers provide hours of work in return for their week's pay.

Exercise 1.26 Revise Listing 1.7 to have it print the status of the portfolio at the end of each five-year period in the overall twenty-year period.
Exercise 1.27 Revise Listing 1.9 to also print the information for the worker with the lowest pay. Assume the existence of one additional obvious method.
Exercise 1.28* Revise Listing 1.8 to have it combine four files into a single file.
Exercise 1.29* Think of another real-life problem a computer program could be used to solve. Write an application program to solve it on the level illustrated here (that is, with all the messy details left for the objects to carry out).
Exercise 1.30* Describe a different situation in which you know computer software models and simulates something. What does it model, what does it simulate, and what are the elements in the situation?

1.8 Program Development: Analysis, Logic Design, Object Design, Refinement, Coding

When you develop software, you need a plan. You cannot just read the statement of a problem and start coding. This section outlines a process for developing software that has been found to be very effective. If you use it, you will take somewhat longer to get to where you have Java code that appears to solve the problem, but you will take much less time to get to where you have Java code that really does solve the problem.

When you worked out some of the Turtle problems, surely you found that it was much easier to come up with the right answer if you first drew a picture of what you were trying to accomplish, putting in pixel measurements where needed. That was part of your plan. Other programming problems are not so strongly graphics-based, so you will generally need to have most of the plan in writing rather than in pictures.

The process presented here has five stages: Analysis, Logic Design, Object Design, Refinement, and then Coding. The presentation in this section must of necessity be rather general, since you have not seen much in the way of language features yet. You will see applications of this process to problems in Chapter Two and Chapter Three. Late in Chapter Three, we will go over these points more concretely, once you have seen Java language features that let you make choices and perform actions repeatedly. And we will discuss them further in Chapter Eight after you have seen larger examples.

Analysis

Before you can make a plan, you have to analyze the problem to see exactly what is required. Drawing a picture is often helpful. Thinking of many different situations in which the software will have to perform helps you decide how you will react to the situation. The statement of a problem is usually incomplete, so you cannot develop a solution until you have the answers to some questions about it. Your objective is to have a clear, complete, and unambiguous specification of the problem before you start working on the solution to the problem.

Choose some test data that the software will have to react to and decide what it should do for that particular situation. Repeat this several times with different data. This helps you more precisely determine what you are supposed to do. Make a record of the data and the expected result so you can test your program when you complete it. The time to develop a test plan is during the analysis and design stages, not after the coding stage.

Logic Design

Decide the order in which tasks will be done. As you do, think of the helpers you would like to have carry out those tasks for you, to make your job easy. For instance, the developer of the program in Listing 1.9 might have worked it out as follows:

"I need a records-clerk helper that can store information about thousands of workers. I will ask this records-clerk helper to get information about all the workers in the company from the company records. Then I will ask this records-clerk helper to produce a worker helper who knows who is the highest paid of all. I will then ask that worker helper to tell me the highest-paid worker's name and pay."

The process you develop should be written out entirely in English in this stage (or whatever natural language you prefer). Do not code anything in Java yet.

Object Design

Decide on a name for each of the types of helpers (known as objects in Java) you need. Then decide how you will phrase the messages you will send to them. In our example of Listing 1.9, the developer decided (a) to name the records-clerk helper a WorkerList, (b) to name the message that asks the records-clerk helper to get information about the workers `addFromFile`, and (c) to send along with that message the file name, because the records-clerk helper needs the file name to perform the task.

Refinement

Study the logic and the objects you have developed so far and make sure that everything is correct. Run through several sets of test data in your mind or on paper to see how the logic and the objects handle it. Make any modifications you see are needed. Do not go further until you are quite sure that the logic is correct.

Coding

Translate your design to Java (this is called **coding** the design). Since this is your first course in software development, you should use the following trick: Only code what you are fairly sure is not going to produce more than three or four errors, if that. If a method requires more than one or two statements, you can leave the method body empty for now or just have it print a simple message, as shown in Listing 1.10. Compile your partially-done program before adding more. That way, when errors occur, you will find it much easier to figure them out and correct them.

For a program of some complexity, you are not done yet. Your design calls on objects to perform subtasks to get the overall task done, but you need to develop the logic for those subtasks. That usually requires that you repeat the process just described, but this time for one subtask at a time. And you will have to repeat the process on subtasks of the subtasks if they are at all complex.

The waterfall model

Just because you are very careful in the steps so far, you cannot be sure that your program is correct. You need to test the program with the test data you developed during analysis and design. This usually leads to changes in coding, sometimes also to changes in the analysis and design. When the program appears correct, you can distribute it. Thereafter, the program will need maintenance as the way in which it is used changes or more faults are found.

The classic **waterfall model** emphasizes these steps: analysis, design, coding, testing, and maintenance. This book concentrates primarily on the first three.

1.9 Placing Java In History

Now that you have some idea of what Java is, you should know something about its historical background. Each computer chip has a set of numeric codes it understands directly, called **machine code**. For instance, 31 65 31 83 31 75 might mean to print the word ASK. Actually, machine code is in binary notation (base 2) rather than decimal notation (base 10), so the instruction would be 11111 1000001 11111 1010011 11111 1001011. Reminder: Base 2 uses powers of 2, e.g., 1111 means $2^3 + 2^2 + 2^1 + 1$, which is 8+4+2+1 = 15; and 100101 means $2^5 + 2^2 + 1$, which is 32+4+1 = 37.

Programming in these machine codes is prohibitively difficult. So **assembler languages** were invented, along with programs to translate the assembly instructions into the chip's machine code. In some assembler language, `PRNT 'A' PRNT 'S' PRNT 'K'` might be how you tell the chip to print the word ASK. This matches up one-for-one with the binary machine code instructions, but it is easier to remember and use. But still, each kind of chip has its own assembler language. If you want to write a program that runs on five different kinds of chips, you have to write it five times in five different assembler languages. And the language is still quite tedious.

High-level programming languages

High-level languages were invented to let you give a command such as `write('ASK')` to print the word ASK for any computer chip. FORTRAN (for science and engineering) and COBOL (for business) were developed in the late 1950s. BASIC (for students), Lisp (for artificial intelligence), and C (for operating systems) were developed in the 1960s and early 1970s. Pascal was developed in the 1970s and became the dominant language for teaching computer science in colleges and universities.

If you want to write a program that runs on five different chips, you just need to write it once if you use Pascal. Then you have it translated (compiled) to each chip's own machine code. You can do this because each of the chips has a compiler written in its own machine code that reads in a source code file written in Pascal and compiles it. The five compilers are written in different languages; but since the compilers have already been developed, additional programs can be written in just the one language Pascal.

Unfortunately, Niklaus Wirth, who invented Pascal, named its successor Modula instead of Pascal++, which is perhaps the primary reason Pascal lost its dominance in university instruction (sometimes marketing is everything). Ada (for government contracts) and C++ (a successor of C, for object-oriented programming) and others were developed in the 1980s. Then along came Java in the 1990s.

Java

Java is a high-level programming language that was originally developed to control electronic consumer appliances, such as CD organizers and home security systems. It is a simpler cleaner language than most other widely-used high-level languages. A key advantage is that it has only one compiled form regardless of the computer chip on which it is used. This is platform independence. The compiled form is called **bytecode**, and it runs on a **Java Virtual Machine** (**JVM**) (which is an "abstract design" of a machine, according to Sun Microsystems).

Sun Microsystems and others have created simulators of the JVM for all the common kinds of computer chips. A JVM simulator, called an **interpreter**, is written in the chip's native machine code. It takes one unit of bytecode at a time and executes it, then goes on to the next, etc. This is slower than executing the compiled form of e.g. Pascal, but chips today are fast enough that speed is not a problem in most cases.

The big advantage is that a program in compiled form (a `.class` file) can be retrieved over the internet and executed immediately. This is better than having to obtain the source code (as with Pascal) and have it compiled on your own machine before you can use it. Browsers have JVM simulators built in so they can execute applets on web pages. And security measures in the browsers prevent applets from harming your computer files. These security measures are not possible with programming languages such as Pascal.

Exercise 1.31 Convert to decimal notation: 1100 and 10110 and 101010.
Exercise 1.32 Convert to decimal notation: 0011,0001,0110 and 1000,0000,1000,1000.

*1.10 Fractal Turtles (*Enrichment)*

Lest you think that such a simple concept as the Turtle only allows simple-minded drawings, the program in Listing 1.11 illustrates the power of Turtles. The upper part contains the application program and the lower part contains the class of Turtle objects the program uses. The application begins by creating a FractalTurtle object, which knows how to draw a fractal. First it backs south by 240 pixels so as to have plenty of room to draw a tall tree. Then it draws a tree with a trunk of length 80 pixels.

This is a very special tree, called a Pythagorean tree, which is one of many fractal images Turtles can make. The directions for making the tree are in the `drawTree` method. It uses four language features you have not seen before (which is why this is an optional section). But you can probably make sense of the following step-by-step description.

Listing 1.11 A program that draws a fractal image

```
public class FractalApp
{
   // Draw a Pythogorean tree.

   public static void main (String[ ] args)
   {  FractalTurtle pythagoras;
      pythagoras = new FractalTurtle();
      pythagoras.move (90, -240);
      pythagoras.drawTree (80);
   }  //====================
}
//#############################################################


public class FractalTurtle extends Turtle
{
   public void drawTree (double trunk)
   {  paint (0, trunk);             // go to top of trunk
      move (30, 0);                 // face to the left
      if (trunk > 1)
         drawTree (trunk * 0.7);    // make branches on the left
      move (-60, 0);                // face to the right
      if (trunk > 1)
         drawTree (trunk * 0.7);    // make branches on the right
      move (30, -trunk);            // go to bottom of trunk
   }  //====================
}
```

How to draw a tree fractally

To draw a tree with a trunk of a given length, the Turtle first paints a straight line `trunk` pixels long. Second, it draws the branches that go off to the left. Specifically, it turns 30 degrees to its left and then draws a tree whose size is 70% of the tree it is in the midst of drawing. That is, the branches it is currently drawing are shaped like a somewhat smaller version of the tree it is currently drawing. This is indicated in the `drawTree` logic by multiplying the value of `trunk` by the number 0.7. Note that it is permissible for distances to be expressed as decimal numbers in the Turtle methods; the word `double` in the parentheses signals that the value to be used is a decimal. Note also that an asterisk is a multiplication sign.

After the Turtle finishes drawing the branches on the left side of the trunk, it swings back to its right 60 degrees, so it is now facing 30 degrees to the right of the trunk line. Then it draws the branches that go off to the right, which look like a smaller version of the tree it is drawing (70% of the size). Finally, the Turtle moves backward so that it is at the bottom of the trunk, where it was initially, and facing in the same direction it was initially.

The process just described goes on forever, which is too long for a computer program, even with a very fast chip. So the Turtle cheats a little. As soon as the tree it is drawing is so small that its trunk is less than a single pixel in length, the Turtle does not draw any of that tree's branches -- it figures you cannot see them on the screen anyway. Figure 1.7 shows the result. It is in the category of fractals which are primarily made up of smaller versions of themselves. The choices of 30 degrees and 70% are arbitrary, chosen after some experimentation to find a nice balance. You could try making different choices for the numbers.

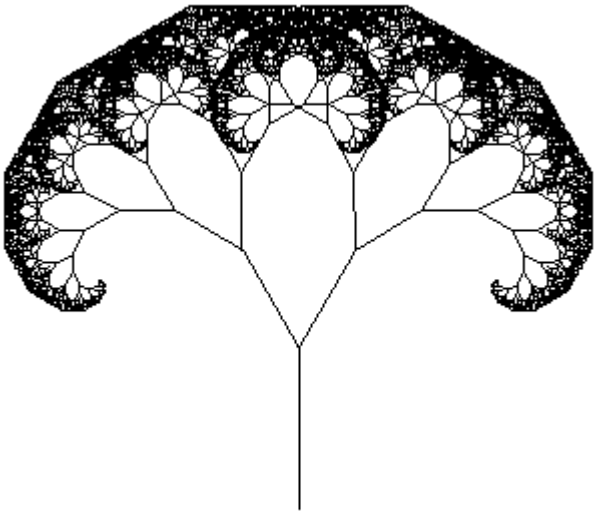


Figure 1.7 Result of executing FractalApp

Exercise 1.33** In the execution of the FractalApp program, how many points are there where the Turtle draws the trunk of a tiny tree that has no branches?

1.11 Review Of Chapter One

Listing 1.3 and Listing 1.4 illustrate all Java language features introduced in this chapter that you need to remember. `http://java.sun.com/docs` has much useful reference material and `http://java.sun.com/products` has the free compiler. Section 1.5 has details on using this free compiler.

About the Java language:

- `SomeClass sam` creates a variable of type SomeClass and declares `sam` as the name of that variable. The phrase `SomeClass sam` is a **variable declaration**. For instance, `Turtle sue` declares a variable for storing a Turtle object.

- Execution of a phrase of the form `sam = new SomeClass()` creates an **instance** (object) of SomeClass and puts a reference to that object in `sam`.
- Anything on a line after `//` is a **comment**; the compiler ignores it.
- A class named SomeClass is to be stored in a plain text file (the **source code**) named `SomeClass.java`. The **compiler** translates this file into a form the **runtime system** can use (the **object code**), stored in a file named `SomeClass.class`. The object code is expressed in **bytecode** which runs on a **Java Virtual Machine** (**JVM**).
- If a class is an **application program**, i.e., with a method whose heading is `public static void main (String[] args)`, then that **main method** can be executed by the runtime system with a command in the **terminal window** of the form `java SomeClass`. An **object class** (with instance methods and no main method) cannot be executed this way.
- When a **method call** has a variable before the dot (period), as in `sam.move(0,10)`, the variable refers to the **executor** of that call. If the statements within some method M include a method call whose executor is not stated, then the executor of the method called is by **default** M's executor.
- A class defined with the phrase `extends Turtle` in the heading is a **subclass** of the Turtle class, and Turtle is its **superclass** (and similarly for other classes besides Turtle). The subclass **inherits** each public method defined in the superclass, i.e., each instance of the subclass can use those methods as if they were defined in the subclass.
- See Figure 1.8 and Figure 1.9 for the remaining new language features. In Figure 1.8, `ArgumentList` stands for whatever values are required by the particular method, with commas separating them if you have more than one of them. Phrases in italics indicate optional parts -- sometimes they are present and sometimes not. A `DeclarationGroup` is zero or more declarations, and a `StatementGroup` is zero or more statements.

`ClassName VariableName;`	**statement** that creates a reference variable
`VariableName = new ClassName();`	**statement** that creates an object and assigns its reference to a reference variable
`VariableName.MethodName ( ArgumentList );`	**statement** that sends a message to the object referred to by the variable named
`MethodName ( ArgumentList );`	**statement** that sends a message to the executor of the method declaration it is in

Figure 1.8 Statements introduced in Chapter One

`public class ClassName extends SuperClassName`
`{ DeclarationGroup`
`}`
	declaration of a class; "extends SuperClassName" is optional

`public static void main (String[ ] args)`
`{ StatementGroup`
`}`
	declaration of a main method of an application

`public void MethodName()`
`{ StatementGroup`
`}`
	declaration of an instance method that is called as a stand-alone statement

Figure 1.9 Declarations introduced in Chapter One

Other vocabulary to remember:

- The programs you run and all the classes they use are **software**. The physical components of the computer (chip, RAM, disk, monitor, etc.) are **hardware**.
- The **method heading** is the first part of the method up through the parentheses; the **method body** is the following matched pair of **braces** `{ }` and its contents. The method body is a sequence of **statements**, most of which are a **command** followed by a semicolon.
- The absence of `static` in the method heading signals you must have an executor in order to call the method. Such a method is an **instance method**.
- A **declaration of a variable name** declares the type of value to be stored there. It is usually followed directly by a **definition of the variable**. You may define (give a value to) a variable many times but declare its name only once.
- The heading of a method **declares** its name to be the name of the method; the body **defines** what the method does. That is, the heading says how it is used and the body says what happens when you use it.
- The **Sun standard library** comes with with the installation of Java that you obtain from Sun Microsystems Inc. It includes the String class and hundreds of others.
- The **keywords** that occur in this chapter are all the words in Figure 1.8 and Figure 1.9 that begin with a small letter, except `args` is not. All non-keywords in a Java program that begin with a letter are **identifiers** (names) of classes, methods, or variables, except for three which you will see later (`true`, `false`, and `null`).
- When the runtime system executes a program, it **links in** all other class definitions required by the program, either directly or indirectly.
- A **prototype** is an executable program that does only part of what the final product is intended to do. The purpose of creating a prototype is to test parts of the system.

About the nine Turtle methods (developed for this book):

- `new Turtle()` creates a Turtle object in the center of a drawing surface 760 pixels wide and 600 tall. The Turtle initially faces east and carries a black paintbrush. In the following, `sam` is a Turtle variable to which you have assigned a Turtle object.
- `sam.paint(angle, dist)` tells `sam` to turn counterclockwise by `angle` degrees and then go forward by `dist` pixels, leaving a trail of the current drawing color.
- `sam.move(angle, dist)` tells `sam` to turn counterclockwise by `angle` degrees and then go forward by `dist` pixels, without leaving any marks.
- `sam.swingAround(dist)` tells `sam` to draw a circle of radius `dist` pixels with `sam` at the center.
- `sam.fillCircle(dist)` tells `sam` to draw a circle of radius `dist` pixels with `sam` at the center, and fill its interior with the current drawing color.
- `sam.fillBox(width, height)` tells `sam` to draw a rectangle `width` pixels wide and `height` pixels tall, with `sam` at the center, and fill its interior with the current drawing color.
- `sam.switchTo(col)` tells `sam` to change the current drawing color to `col`, which can be any of BLACK, GRAY, BLUE, GREEN, RED, YELLOW, ORANGE, PINK, MAGENTA, and WHITE. Put "Turtle." in front of the name of a color you use in a program, unless you use it in a subclass of the Turtle class.
- `sam.say("whatever")` tells `sam` to print whatever is within the quotes.
- `sam.sleep(milli)` tells `sam` to suspend action for `milli` milliseconds.

Answers to Selected Exercises

1.1
```
sue.paint (90, 60);
sue.paint (90, 120);
sue.paint (90, 60);
sue.paint (90, 120);
```

1.2
```
sam.paint (90, 12);
sam.move (180, 2);
sam.paint (135, 3);
sam.paint (-45, 6);
```

1.3
```
public class LetterB
{   public static void main (String[ ] args)
    {   Turtle cat;
        cat = new Turtle();
        cat.paint (0, 6);  // the top of the rounded part of the 'b'
        cat.paint (-90, 6);
        cat.paint (-90, 6);  // the bottom of the rounded part of the 'b'
        cat.paint (-90, 12);
    }
}
```

1.4
```
public class LetterM
{   public static void main (String[ ] args)
    {   Turtle cat;
        cat = new Turtle();
        cat.paint (90, 6);  // the left side of the 'm'
        cat.paint (-90, 4);
        cat.paint (-90, 6);  // the center part of the 'm'
        cat.move (180, 6);
        cat.paint (-90, 4);
        cat.paint (-90, 6);
    }
}
```

1.5
```
public class Hexagon
{   public static void main (String[ ] args)
    {   Turtle cat;
        cat = new Turtle();
        cat.paint (60, 50);
        cat.paint (60, 50);
        cat.paint (60, 50);
        cat.paint (60, 50);
        cat.paint (60, 50);
        cat.paint (60, 50);
    }
}
```

1.8
```
public class SquaresInSquares
{   public static void main (String[ ] args)
    {   SmartTurtle cat;
        cat = new SmartTurtle();
        cat.makeBigSquare ();
        cat.move (0, 50);   // this is arbitrary; anything over 40 would do
        cat.makeBigSquare();
        cat.move (90, 25);
        cat.move (90, 25);  // to upper-left corner of inner square
        cat.makeSmallSquare();
        cat.move (0, 50);   //  to upper-left corner of the other inner square
        cat.makeSmallSquare();
    }
}
```

1.9
```
public void goAndSquare()
{   move (90, 70);
    makeSmallSquare();
}
```
Then replace each of the last three pairs of statements in Listing 1.4 by:
```
sue.goAndSquare();
```

1.10
```
public void drawHexagon()
{   paint (60, 30);
    paint (60, 30);
    paint (60, 30);
    paint (60, 30);
    paint (60, 30);
    paint (60, 30);
}
```

1.11
```
public class ThreeHexagons
{   public static void main (String[ ] args)
    {   SmartTurtle cat;
        cat = new SmartTurtle();
        cat.drawHexagon();
        cat.move (120, 0);
        cat.drawHexagon();
        cat.move (120, 0);
        cat.drawHexagon();
    }
}
```
1.15 Put move (0, -20); right after the first drawFlower();.
Put move(0,20); right after the second drawFlower();.

1.16
```
public void drawSmallFlower()
{   paint (90, 7);
    switchTo (RED);
    fillCircle (3);
    switchTo (BLACK);
    move (180, 7);
}
```
1.17 First put the 8 statements of drawLeaf in drawFlower twice, so that it has 13+8+8 = 29 statements. So drawTwoFlowers will have 2+29+29 = 60 statements.

1.18
```
public class Target
{   public static void main (String[ ] args)
    {   Turtle cat;
        cat = new Turtle();
        cat.switchTo (Turtle.RED);
        cat.fillCircle (80);    // the 80 and other numbers are arbitrary
        cat.switchTo (Turtle.YELLOW);
        cat.fillCircle (60);
        cat.switchTo (Turtle.BLUE);
        cat.fillCircle (40);
        cat.switchTo (Turtle.BLACK);
        cat.fillCircle (20);
    }
}
```
1.26 Put the following eight statements in place of the last two statements of Listing 1.7:
```
wealth.waitForYears (5);
wealth.displayCurrentValues();
wealth.waitForYears (5);
wealth.displayCurrentValues();
wealth.waitForYears (5);
wealth.displayCurrentValues();
wealth.waitForYears (5);
wealth.displayCurrentValues();
```
1.27 Add the following three statements at the end of the main method:
```
Worker lowestPaid;
lowestPaid = company.getWorkerWithLowestPay();
System.out.println (lowestPaid);
```
1.31 8+4=12 and 16+4+2=22 and 32+8+2=42.

1.32 3*256+1*16+6=790 and 8*16*256+8*16+8 = 32768+128+8 = 32904.

2 Conditionals and Boolean Methods

Overview

This chapter introduces the Vic software to control electronic components. Vics provide a moderately realistic context in which you can learn about some new Java language features. These features let you ask your objects questions and then decide, based on the answers they give, what actions they should take.

- Sections 2.1-2.2 describe basic Vic commands and review main methods and subclassing.
- Sections 2.3-2.5 present conditional statements.
- Sections 2.6-2.8 discuss boolean operators, boolean variables, and methods that return boolean values.
- Sections 2.9-2.10 introduce additional useful topics including UML notation.

Some students like a more detailed preview of the language features they are about to learn. If you are such a student, you will find it helpful to skim the review at the end of each chapter before reading the chapter itself.

2.1 Using Vic Objects To Control Appliances

Your company sells a programmable machine that stores compact discs (CDs) and moves them around when the operator pushes certain buttons. The machine uses a computer chip to do this. Your job is to write the programs for this chip, using the Java programming language.

The primary component of this machine has a mechanical arm and one sequence of three or more slots for storing CDs (the higher-priced machines have more slots). At any given time, some slots contain a CD and some do not. The mechanical arm is positioned at one point in the sequence of slots, either right at a slot or just after all the slots.

Each programmable machine has one or more of these primary components. The machine also has a place where extra CDs can be stored, called its **stack** of CDs. This stack sometimes contains many CDs and sometimes contains none at all. When you put a CD on the stack, and later put another CD on the stack, then the later one is on top. That means that, the next time you take a CD off of the stack, you will get the later one, not the one that is underneath it.

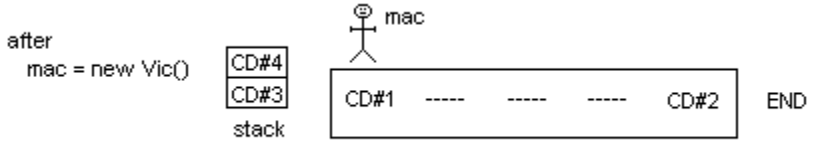
You can have one of these primary components perform one of four kinds of operations, named `takeCD`, `moveOn`, `putCD`, and `backUp`:

- The `takeCD` operation causes the mechanical arm to take a CD out of the slot at the current position and place it on top of the stack. This does not change the arm's position in the sequence. If there is no CD in the slot, the `takeCD` operation has no effect.
- The `moveOn` operation moves the mechanical arm down from its current position in the sequence of slots to the next position.
- The `putCD` operation causes the mechanical arm to remove a CD from the top of the stack and put it in the slot at the current position. This does not change the arm's position in the sequence. If a CD is already in the slot, or if no CD is in the stack, the `putCD` operation has no effect.
- The `backUp` operation moves the mechanical arm up from its current position in the sequence of slots to the position just before it.

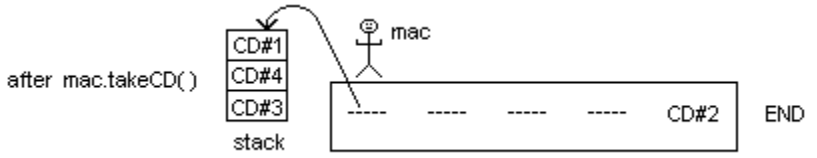
The phrase `Vic mac` in a program declares `mac` as the name for a variable, a part of the RAM's data area which can refer to a Vic object. The phrase `mac = new Vic()` in a program creates the object and puts a reference to the object in the variable named `mac`. Thereafter, a mention of `mac` in the program is an indirect mention of the object. The object specifies the component's sequence of slots and current position.

Figure 2.1 shows how the status of the Vic object changes after each operation. The stick figure indicates the position of the mechanical arm that shifts CDs. Initially, this particular Vic object has a sequence of five slots, with CDs in the first and last slots, as well as two CDs in the stack. The initial arrangement of CDs is whatever was left in the slots and stack when the machine was last turned off.

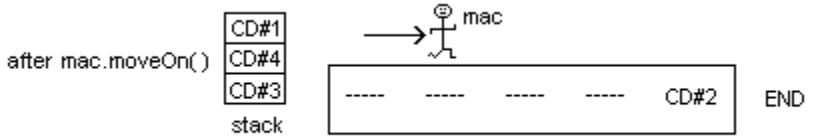
`mac = new Vic()`
creates the internal description for the Vic object, positioned at the beginning of the sequence of slots.



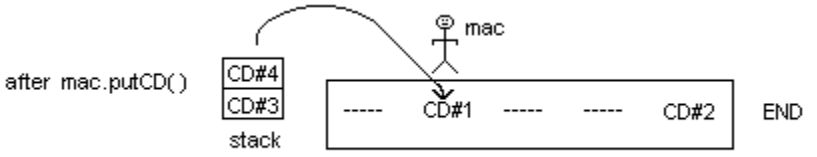
`mac.takeCD()`
asks the Vic to take a CD out of its current slot and put it on top of the stack of CDs for the machine.



`mac.moveOn()`
asks the Vic to move to the next slot after the one at its current position.



`mac.putCD()`
asks the Vic to put a CD in its current slot. This CD comes from the top of the stack of CDs.



`mac.backUp()`
asks the Vic to back up to the slot just before the one at its current position.

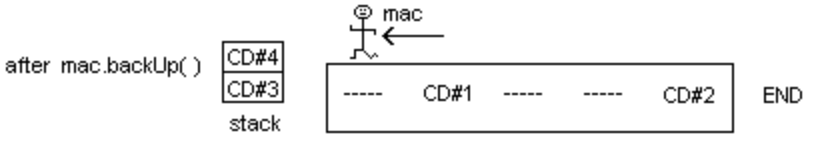


Figure 2.1 The meaning of the four basic Vic commands

Objects in Vic programs

When a program that operates the Programmable CD Organizer machine begins its work, it creates an internal description of each of these primary components and puts that data in RAM. This internal description is an object. An object is a virtual mechanism when it is a computer model of a physical mechanism. Since this particular object stores Virtual CDs, we can call it a Vic for short.

The four basic Vic operations of `putCD`, `takeCD`, `moveOn`, and `backUp` are what actually move the springs, gears, and grippers in the physical machine. You write a program to perform a complex task by performing these simple physical actions in an appropriate sequence. When `mac` refers to some Vic, `mac.takeCD()`, `mac.moveOn()`, `mac.putCD()`, and `mac.backUp()` are the ways you express these actions in a Java program.

An application program

Suppose you want a Vic object to take a CD from its third slot and put the CD in its second slot. You could have the computer chip execute the application program in Listing 2.1. The comments at the end of certain lines (signaled by the `//` symbol) explain what is happening. The class heading and the method heading (the two boldfaced lines) plus the corresponding two pairs of braces are what this book always uses for application programs, except that the name MoveOne is chosen to reflect what the specific program does.

Listing 2.1 An application program using one Vic object

```
public class MoveOne
{
   // Take a CD from the third slot; put it in the second slot.

   public static void main (String[ ] args)
   {  Vic sue;          // 1
      sue = new Vic();  // 2

      sue.moveOn();     // 3  move to the second slot
      sue.moveOn();     // 4  move to the third slot
      sue.takeCD();     // 5  take CD from slot 3, put on stack

      sue.backUp();     // 6  move back to the second slot
      sue.putCD();      // 7  put CD in slot 2, taken from stack
   }  //=====================
}
```

Figure 2.2 shows the status of the Vic object at three points during one execution of the MoveOne program. The first two statements of the main method create the Vic object, position it at the first slot in the first sequence of slots, and make `sue` a reference to that object. The top part of Figure 2.2 shows what things might look like at this point.

After the Vic object is created, the next three statements move the mechanical arm to the third slot in the sequence and then take CD#1 out of that slot. The middle part of Figure 2.2 shows the current status. Note that CD#1 is no longer in the slot; it is on the stack.

The last two statements of the main method move the mechanical arm back up to the second slot in the sequence and then have it transfer CD#1 from the stack into that second slot. The bottom part of Figure 2.2 shows the final status of the machine.

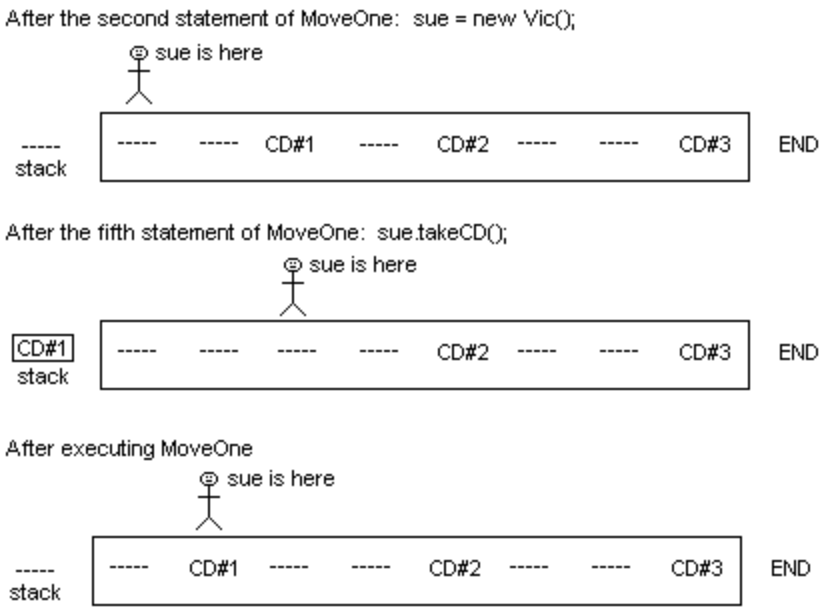


Figure 2.2 Stages of execution for MoveOne

The Vic simulator

The company engineers have not finished building the physical machine, so they have provided you with a simulation in software. It is the Vic class, available on this book's website. It lets you test the programs you write. When you run a program with this simulation, a graphical representation of the entire machine appears on the screen with CDs and slots. It carries out the commands you gave it, slowly enough for you to follow. If for some reason you do not have that Vic class available, you may type in the simpler implementation that is in Chapter Five and compile it.

To run an application program such as MoveOne in Listing 2.1, you must already have the Vic class compiled. You type the lines of the MoveOne program into a plain-text file named `MoveOne.java`, then compile it to produce the executable file named `MoveOne.class`. You enter `java MoveOne` at the prompt in the terminal window to run the program. The runtime system then executes the commands in the executable file.

For this particular Vic simulator, the graphics display shows one to four sequences of slots, representing actual components of the physical machine. Each sequence has at most eight slots. Figure 2.3 shows roughly what the graphics display looks like. The Vic software provides faked names for CDs consisting of a letter and a digit, so you can tell which CDs were originally in which slots. The CD names along the left side (b1 and a2 in the figure) are the CDs that are on the stack (b1 is on top of the stack).

Figure 2.3 shows three sequences with four slots in the first two and three slots in the third. The stick figure indicates that only one Vic object has been created (i.e., `new Vic()` has been executed one time), and it is currently positioned at the second slot.

Vic methods that perform an action print a record of the action in the terminal window. If the graphics window covers the terminal window, move the terminal window around to make at least its lower part visible so you can see this record.

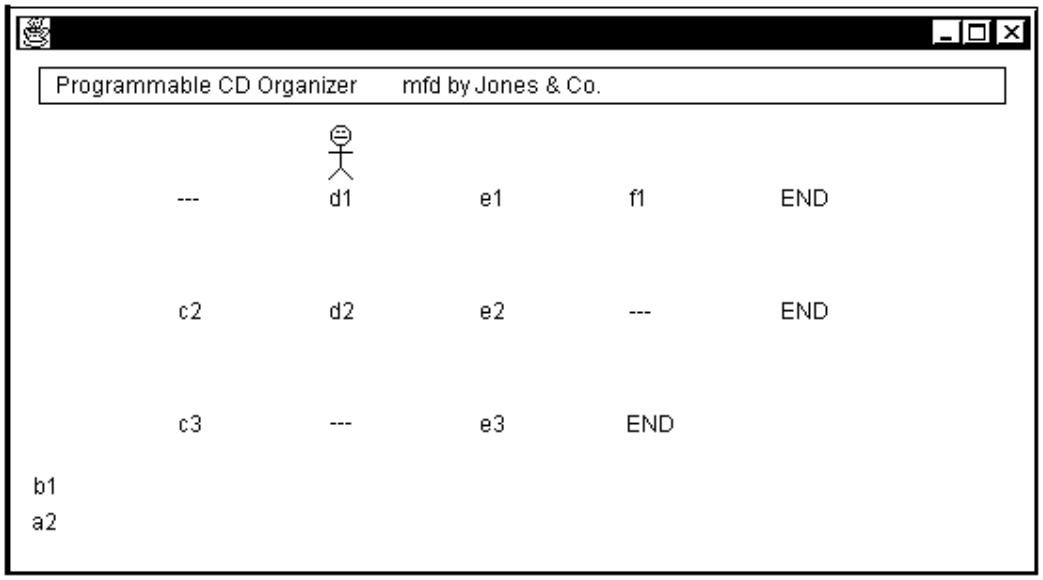


Figure 2.3 The graphics display for the Vic software

The reset command

You can test your programs by setting up the CDs in a particular order before a test run. The Vic simulator provides a `reset` command that puts CDs in whatever slots you choose at runtime; simply have `Vic.reset(args);` as the first statement of the main logic. For instance, if you want to have the arrangement of CDs shown in Figure 2.3, you execute the program by giving the command

```
java MoveOne 0111 1110 101
```

in the terminal window. The simulator then creates three sequences (because you gave three "words" after the basic command) with no CD wherever you have a 0 in a string and a CD wherever you have something other than a 0.

If you execute the same program using the command `java MoveOne 010 01100`, the simulator creates a machine with two sequences: The first sequence has three slots with a CD only in the second slot, and the second sequence has five slots with CDs only in the second and third slots.

Those extra words in the command line are called **command-line arguments**. The runtime system puts those words in `args`, and the `reset` command uses the information it receives in `args` to initialize the simulation. If no words are entered, or if the `reset` command is not used before any Vic object is created, the simulation creates an initial arrangement of CDs, slots, and sequences for you at random.

Language elements

A CompilableUnit can be:	public class ClassName { Declaration }	
A Declaration can be:	public static void main (String [ ] args) { StatementGroup }	
A StatementGroup is any number of Statements.		
A Statement can be:	ClassName VariableName ;	e.g., Vic sam;
or:	VariableName = new ClassName ( ) ;	e.g., sam = new Vic();
or:	VariableName . MethodName ( ) ;	e.g., sam.backUp();
or:	ClassName . MethodName (Expression) ;	e.g., Vic.reset(args);

Anything after // on a line in a program has no effect on the program.

Exercise 2.1 Write an application program that creates a Vic object and then moves a CD out of its third slot into its first slot. Assume the first slot is empty and the third is not.
Exercise 2.2 Write an application program that creates a Vic object and then puts a CD in each of its first three slots. Assume the stack has enough CDs.
Exercise 2.3 Write an application program that creates a Vic object and then takes a CD out of its second slot and its fourth slot. Use `reset` to be sure you have enough slots.
Exercise 2.4* Write an application program that creates a Vic object and then takes a CD from each of its second and fourth slots and puts one of them in its fifth slot. Use `reset` to be sure you have enough slots. Which one ends up in the fifth slot?
Exercise 2.5* Write an application program that creates a Vic object and then swaps the CD in its third slot with the CD in its second slot. Assume it has CDs in both slots.

2.2 Defining A Subclass Containing Only Instance Methods

When a command in a program refers to a Vic object before the dot, it is a message sent to the Vic object. For instance, `sue.putCD()` sends a message to `sue` requesting that `sue` put a CD into the current slot from the stack of CDs. So the main method in MoveOne (Listing 2.1) sends a message to a Vic requesting it to move forward to the third slot and take the CD from that slot. It then sends messages to the Vic requesting it to move back to the second slot and put the CD from the stack into that second slot.

Reminder: The stack and the sequences of slots exist independently of the program that controls them. The phrase `new Vic()` does not create a stack and a sequence; it only creates a description of them in RAM. You need the internal description (object) so you can send messages that cause changes in a physical stack and sequence.

Choosing statements to make into a method

You can expect to use the sequence of two messages

```
moveOn();
takeCD();
```

many times in many programs, with various Vic objects receiving that pair of messages. You saw `sue` doing these actions in Listing 2.1. Java allows you to invent new messages for Vics that are combinations of existing messages. This simplifies your programs. For instance, you can define a new message named `moveTake`: `sam.moveTake()` tells `sam` to execute those two commands in that order, and `sue.moveTake()` tells `sue` to execute those two commands in that order.

The sequence of two messages

```
backUp();
putCD();
```

will also be quite common, sent to various Vic objects (you saw it in Listing 2.1 with `sue` receiving the messages). You can define a new message named `backPut`: `sam.backPut()` tells `sam` to carry out those two messages in that order, and `sue.backPut()` tells `sue` to carry out those two messages in that order.

A simple Vic object does not know what the two words `moveTake` and `backPut` mean. They are not part of its vocabulary. You need a new class of objects that can understand these two messages plus all the messages a Vic understands. Let us call this new kind of Vic object a SmartVic. Then you could rewrite the main method in Listing 2.1 to do exactly the same thing but with a simpler list of statements, as follows:

```
   public static void main (String[ ] args)
   {  SmartVic sam;          // create variable named sam
      sam = new SmartVic();  // create object that sam refers to
      sam.moveOn();
      sam.moveTake();        // take CD in slot 3, put on stack
      sam.backPut();         // put that CD in slot 2
   }  //=====================
```

How to define a class of objects

The class definition in Listing 2.2 says that, if you create an object using the phrase `new SmartVic()` instead of `new Vic()`, you can send the `moveTake` and `backPut` messages to that object, as well as all of the usual Vic messages. In a sense, a SmartVic object is better educated than a basic Vic object, because it inherits all of the capabilities of a Vic object and adds two more.

Listing 2.2 The SmartVic class of objects

```
public class SmartVic extends Vic
{
   public void moveTake()
   {  moveOn();
      takeCD();
   }  //=====================

   public void backPut()
   {  backUp();
      putCD();
   }  //=====================
}
```

A class definition that extends the capabilities of a Vic object must have the heading

```
   public class WhateverNameYouChoose extends Vic
```

followed by a matched pair of braces that contain some definitions. This particular class definition contains two method definitions beginning `public void`. Within such a method definition, you do not name the variable that refers to the object that receives the message. This lets you use any variable name you like (such as `sue`, `sam`, or whomever) when you give the command outside the method. So the `backPut` definition says that for any `x`, if you have previously defined `x = new SmartVic()`, then `x.backPut()` has the same meaning as the following:

```
   x.backUp();
   x.putCD();
```

Listing 2.3 illustrates the use of these new commands in a program that moves three CDs backward one slot. The program creates a new SmartVic object and sends it messages to take the CD out of slot 2 and put that CD into slot 1. Then the object moves forward to slot 2 so it can repeat the actions, this time taking the CD out of slot 3 and putting it in slot 2. Finally, the object moves forward to slot 3 and repeats the actions, this time taking the CD out of slot 4 and putting it in slot 3. Figure 2.4 shows a sample run of this program.

Listing 2.3 An application program using one SmartVic

```
public class BringThreeBack
{
   // Move the CDs in slots 2, 3, and 4 back to slots 1, 2, 3,
   // respectively.  Presumes a reset with at least 4 slots.

   public static void main (String[ ] args)
   {  Vic.reset (args);     // 1
      SmartVic sue;         // 2
      sue = new SmartVic(); // 3
      sue.moveTake();       // 4  move to slot 2 and take CD
      sue.backPut();        // 5  back to slot 1 and put CD there

      sue.moveOn();         // 6
      sue.moveTake();       // 7  move to slot 3 and take CD
      sue.backPut();        // 8  back to slot 2 and put CD there

      sue.moveOn();         // 9
      sue.moveTake();       // 10 move to slot 4 and take CD
      sue.backPut();        // 11 back to slot 3 and put CD there
   }  //====================
}
```

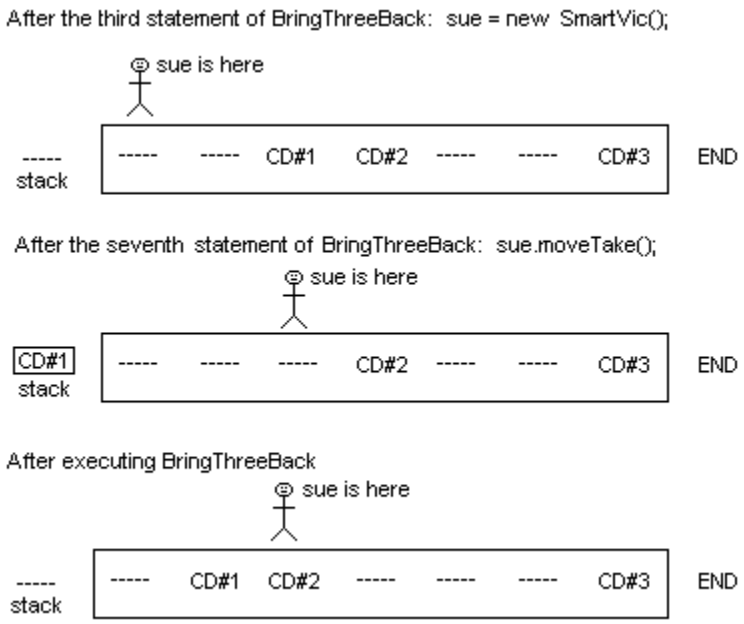


Figure 2.4 Stages of execution for BringThreeBack

Caution You will save yourself a lot of grief if you check the following three points before you compile a class definition: (1) No class heading or method heading has a semicolon at the end. (2) Each class heading and method heading has a left brace immediately below it. (3) Every left brace has a corresponding right brace several lines lower and aligned with it.

Language elements

A CompilableUnit can be: public class ClassName extends ClassName { DeclarationGroup }
A DeclarationGroup is any number of Declarations.
A Declaration can be: public void MethodName ( ) { StatementGroup }
A Statement can be: MethodName ( ) ; e.g., moveOn();

Terminology The phrase `sue.moveTake()` in the BringThreeBack class is a **method call**. The phrase `public void moveTake()` in the SmartVic class is the **method heading**. But the **method** itself is the process of moving the mechanical arm forward and taking a CD. If you misspell or miscapitalize the method call, the compiler will not be able to connect it to the method heading so that the runtime system can carry out the method.

Exercise 2.6 Rewrite the main method in the answer to Exercise 2.1 to use SmartVics instead of Vics, thereby shortening the logic.
Exercise 2.7 Rewrite the main method in the answer to Exercise 2.3 to use SmartVics instead of Vics, thereby shortening the logic.
Exercise 2.8 Write a SmartVic method `public void movePut()`: The executor moves forward one slot and puts a CD there from the stack.
Exercise 2.9 Write an application program that uses a SmartVic object, as augmented by the preceding exercise, to move a CD from the third slot to the fifth slot.
Exercise 2.10* Write a SmartVic method `public void backTake()`: The executor moves backward one slot and takes a CD from there to go on top of the stack.
Exercise 2.11* Write an application program that uses a SmartVic object, as augmented by `movePut` and `backTake` as just described, to swap the CD in the second slot with the CD in the first slot using only four message commands.
Exercise 2.12* Write an application program that uses a SmartVic object, augmented by `movePut` and `backTake` as just described, to move the CD in the first slot into the second, the CD in the second slot into the third, and the CD in the third slot into the first.

2.3 The If Statement

A program that is given bad input can produce bad results. For instance, if some program is set up to accept only a numeric input at a certain point, and the input has letters in it, the program may fail. Or if a Vic program is set up to take a certain action at the fourth or fifth slot of a sequence and the sequence only has three slots, the program may fail. In both of these cases the program is not **robust**: It does not handle unexpected input well. For instance, Listing 2.3 is not robust because it fails if the sequence has only three slots.

You can make your Vic programs robust if you learn to expect the unexpected and adjust for it. A program fails if there is no slot at a point where a Vic tries to `putCD`, `takeCD`, or `moveOn`. To prevent failure, you need to be able to test whether a certain condition is true. Fortunately, you can ask a Vic object either of the two kinds of questions shown in Figure 2.5; the Vic object (referred to here as `aVic`) gives the answers in the form of a **condition** (an expression that is either true or false).

`aVic.seesSlot()`
is true if aVic is not past its last slot and is false otherwise. So it means aVic actually has a current slot.

`aVic.seesCD()`
is true if aVic's current slot has a CD in it and is false otherwise. Evaluation of this condition causes the program to fail if aVic.seesSlot() is false.

Figure 2.5 Two Vic methods that answer questions

Examples of if statements

In order to use these conditions, you need to have a kind of statement that will take an action if a certain condition is true but will skip the action if the condition is false. The if-statement is that kind of statement. It comes in two forms, illustrated in Listing 2.4.

Listing 2.4 An application program using one Vic object, with if-statements

```
public class TwoToFour
{
   // If a CD is in the second slot, take it out and then
   // put it in the fourth slot if possible (robustly).

   public static void main (String[ ] args)
   {  Vic.reset (args);       // 1
      Vic sue;                // 2
      sue = new Vic();        // 3
      sue.moveOn();           // 4

      if (sue.seesCD())       // 5  if sue sees a CD in slot 2,
      {  sue.takeCD();        // 6     then sue takes that CD
         sue.moveOn();        // 7
         sue.moveOn();        // 8
         if (sue.seesSlot())  // 9  if sue has a slot number 4,
            sue.putCD();      // 10    then sue puts the CD there
      }                       // 11
   }  //======================
}
```

Listing 2.4 creates a Vic object and has it move to the second slot in its sequence (lines 1-4). If it does not see a CD in that slot, nothing else happens in the program. This is because the if-statement (line 5) tests the condition `sue.seesCD()` and, if the condition is false, skips execution of all the statements between the matched braces that follow the condition (lines 6-11).

Those statements to be executed if the second slot has a CD are to take the CD from the slot, move on to the fourth slot, and put the CD there. However, a sequence might have only three slots. If so, trying to put a CD in the fourth slot would make the program fail (specifically, the Vic software prints a warning message and stops the program). So this program tests the condition `sue.seesSlot()` at the fourth slot (line 9) and, only if that condition is true, executes the one statement that follows the condition (line 10).

An **if-statement** has two parts for you to fill in: the condition it tests and the subordinate statement it executes only if the condition is true. This book boldfaces the word `if` in listings to signal it needs a subordinate statement. The two general forms of the basic if-statement are as follows:

```
if (Condition)          if (Condition)
   Statement            {  Statement
                           Statement...
                        }
```

If you want a group of <u>two or more</u> statements to be executed <u>only</u> when the given condition is true, you must put the matched pair of braces around the group. If you want only <u>one</u> statement conditionally executed, you do not need the braces around it. However, you may use them if you wish. As Listing 2.4 illustrates, you are allowed to have any statements inside the braces of an if-statement, even another if-statement.

Continuing the email message metaphor of Section 1.6, when you write `sam.seesCD()` in a program, it sends a message to the person whose email address is stored in `sam`. The message says, "Is it true or false that you see a CD in your current slot?" The person sends back to you a response of `true` or `false`. You then look at the response to decide what to ask the person to do next.

Using several Vic objects in a program

Suppose you want to move the CD from slot 3 into slot 5 for each of the first three sequences of slots. This is a very straightforward thing to do except for two possible problems: One is that you might not have as many as three sequences, and the other is that one or more of those sequences might not have as many as five slots.

The `seesSlot()` method call can be used to solve both of those problems. The first time your program creates a new Vic object, you get the first sequence of slots. The next time your program creates a new Vic object, you get the second sequence of slots, if there is one. If you then immediately test `seesSlot()`, it will be false if there was no second sequence. Otherwise, it will have at least three slots, since all sequences do. This all leads to the structured plan shown in the accompanying design block.

DESIGN for moving the CD from slot 3 into slot 5 for each of three sequences

1. Create the first Vic object.
2. If its third slot contains a CD, then...
   Move it into the fifth slot, checking first that the Vic has a fourth and fifth slot.
3. Create a second Vic object.
4. If it has at least one slot, it has at least three (by definition of Vics), so...
   4a. Do what you did for the first sequence, as described in step 2.
   4b. Create a third Vic object.
   4c. If it has at least one slot, it has at least three (by definition of Vics), so...
   Do what you did for the first sequence, as described in step 2.

Listing 2.5 (see next page) implements this plan as a Java program. Since Step 2 requires a method for accomplishing a subtask that is used in two more places, it is best to make a Java method out of it, the `shiftThreeToFive` method in the Shifter class (Note: You do not need to memorize any of these Vic subclasses for later in this book).

Java requires you to put each class with the heading `public class Whatever` in a separate file to be compiled, with the name `Whatever.java`. Listing 2.5 is the contents of two separate compilable files, `ThreeSequences.java` and `Shifter.java`.

Note particularly the advantage of defining a method without mentioning which object is to carry out the task. That allows you to have three different Vic objects execute the sequence of statements in the `shiftThreeToFive` method.

The ThreeSequences program creates three different sequences, so it uses the three different descriptive variable names `one`, `two`, and `three`. However, since `one` is never mentioned after `two` is created, nor `two` after `three` is created, you could use just one Vic variable throughout. For instance, you could write `sam` in place of each of the three variable names in that class. But then you would have to omit the second and third declarations of Vic variables -- you can only declare a variable one time in a method.

For all exercises from now on, unless otherwise stated, write your answer so that the application programs cannot fail. You <u>are</u> doing all the unstarred exercises, are you not? You cannot learn this material well if you do not do them. You may compile a method you write as an exercise by enclosing it in a matched pair of braces with an appropriate class heading.

Language elements

A Statement can be:	if (Condition) Statement	
or:	if (Condition) { StatementGroup }	
A Condition can be:	MethodName ( )	e.g., seesCD()
or:	VariableName . MethodName ( )	e.g., sue.seesCD()

Listing 2.5 An application program using three objects of the Shifter class

```
public class ThreeSequences
{
   // For each of the first three sequences, move a CD from the
   // third slot to the fifth slot, insofar as possible.

   public static void main (String[ ] args)
   {  Vic.reset (args);
      Shifter one;
      one = new Shifter();                  // design step 1
      one.shiftThreeToFive();               // design step 2

      Shifter two;
      two = new Shifter();                  // design step 3
      if (two.seesSlot())                   // design step 4
      {  two.shiftThreeToFive();            // design step 4a
         Shifter three;
         three = new Shifter();             // design step 4b
         if (three.seesSlot())              // design step 4c
            three.shiftThreeToFive();
      }
   }  //====================
}
//#################################################################

public class Shifter extends Vic
{
   // take the CD from slot 3 (if any) and put it in slot 5.

   public void shiftThreeToFive()
   {  moveOn();
      moveOn();                             // now in slot 3
      if (seesCD())
      {  takeCD();
         moveOn();                          // now in slot 4
         if (seesSlot())
         {  moveOn();                       // now in slot 5
            if (seesSlot())
               putCD();
         }
      }
   }  //====================
}
```

Exercise 2.13 Revise the program in the earlier Listing 2.3 so it cannot fail even with bad input (from the `reset` method).
Exercise 2.14 Write an application program that creates a Vic and then has it take a CD from each of its fourth and fifth slots. Be sure the program cannot fail.
Exercise 2.15 Write a method `public void swapTwo()` for a subclass of Vic: The executor swaps the CD in the current slot with the CD in the following slot, except that it moves no CD at all if either CD is missing. Leave the executor at its original position.
Exercise 2.16* Write an application program that creates a Vic and then has it take a CD out of its fifth slot and put it into its seventh slot. Be sure the program cannot fail.
Exercise 2.17* Write an application program that moves a CD from slot 3 to slot 2 of the first sequence and also from slot 2 to slot 3 of the second sequence. Use SmartVic objects (as defined in the earlier Listing 2.2). Be sure the program cannot fail.
Exercise 2.18* Write an application program that takes the CD out of the second slot of each of the first four sequences. Use SmartVic objects. Be sure the program cannot fail.

2.4 Using Class Methods And Javadoc Comments In A Program

Some methods can be called with the class name in place of an executor. An example is `Vic.reset(args)`. Such a method is called a **class method.** You are sending a message to the class as a whole, not to an individual instance of the class. By contrast, a method that requires an instance (object) of the class for its executor is an **instance method**.

Two new Vic class methods

The Vic class provides a class method for you to print messages on the display. If you put `Vic.say("Hello world")` in your program, `Hello world` will appear on an LCD screen on the front of the machine. `Vic.say("No CDs")` will cause `No CDs` to appear on the front of the machine. You can put whatever you want inside the quotes except a backslash `\` or quotes themselves (this caveat is explained in Chapter Six).

The Vic class also provides a class method for you to find out whether the stack has any CDs on it (as opposed to being empty). If you execute `sam.putCD()` and the stack is empty, nothing happens. Often you need to know whether something happens. So you test the condition `Vic.stackHasCD()`. Figure 2.6 describes these two new methods more precisely.

`Vic.say("whatever")`
displays the given message on an LCD screen the machine has, so the operator of the Programmable CD Organizer can read it. The command first erases whatever might have been on the LCD screen before.

`Vic.stackHasCD()`
is true if the stack contains at least one CD and is false otherwise.

Figure 2.6 Two new Vic class methods

A **parameter** or **argument** is a value you put in the parentheses after a method name to make a method call. It gives information to the method that it needs so it can do its job. The parameter of `Vic.say` is always a string of characters. The phrase `Vic.say("Hello")` is a method call with a parameter, namely `"Hello"`. `Vic.reset(args)` is another method call with a parameter `args`, which contains whatever strings of characters might have been typed on the command line.

Programming Style Java allows you to call a class method by using an instance of the class in place of the class name. For instance, if you declare `Vic sam`, then `sam.say("Hi")` is a legal method call and does the same thing as `Vic.say("Hi")`. However, this can be deceptive, so it should be avoided.

The `switchTo` method in the Turtle class (Section 1.4) is actually a class method, since the color applies to the drawing surface rather than to the individual Turtle. So it is better to use `Turtle.switchTo(aColor)` than `sam.switchTo(aColor)`. This was not mentioned in Chapter One only because it contained enough material as it was. Note: You may use `say` or `stackHasCD` within a method in a subclass of Vic without having "Vic." before it -- the compiler will know what you mean.

Example using the three class methods

Suppose you would like to have a program to fill in the first three slots of the first sequence. The program begins with an unknown number of CDs in the stack. It makes sense to stop trying to fill slots as soon as the stack runs out of CDs. So begin by seeing if the stack has any CDs. If so, you create a new Vic object and fill the first slot, then check to see if the stack has more CDs. If so, you move on to the second slot and put a CD there. You check one more time to see if the stack still has at least one CD. If so, you move on to the third slot and put a CD there.

This logic is implemented in Listing 2.6. The condition in line 3 causes the executor to skip lines 4-15 if the stack is empty. The condition in line 7 causes the executor to skip lines 8-14 if the stack is empty at that point. Of course, if a slot already has a CD in it, the `sue.putCD()` message has no effect, so the CD on the stack will be available for the next slot.

Listing 2.6 An application program using the three Vic class methods

```
public class FillThreeApp
{
   /** Put a CD in each of the first three slots.
    *  Stop as soon as you run out of CDs on the stack. */

   public static void main (String[ ] args)
   {  Vic.say ("This program fills the first three slots.");
      Vic.reset (args);                  // 2
      if (Vic.stackHasCD())              // 3
      {  Vic sue;                        // 4
         sue = new Vic();                // 5
         sue.putCD();                    // 6    in slot 1
         if (Vic.stackHasCD())           // 7
         {  sue.moveOn();                // 8    to slot 2
            sue.putCD();                 // 9
            if (Vic.stackHasCD())        // 10
            {  sue.moveOn();             // 11   to slot 3
               sue.putCD();              // 12
            }                            // 13
         }                               // 14
      }                                  // 15
      Vic.say ("All done!");             // 16
   }  //======================
}
```

Elementary javadoc comments

Listing 2.6 illustrates a second kind of Java comment: `/*` causes everything to be ignored until `*/` is seen, even if several lines later. This book henceforth uses the **javadoc standard for commenting methods**, which is the following:

- Put the description of the method before the method heading, beginning with `/**`.
- Put an asterisk at the beginning of each additional line (this asterisk is optional).
- Put `*/` at the end of the comment.

The javadoc formatting tool uses these special comments. You execute it by issuing in the terminal window the command `javadoc SomeClass.java` for a class declared in the `SomeClass.java` file. Then the javadoc formatting tool produces a webpage in a

file named `SomeClass.html` which displays documentation for the class very nicely. Each comment that comes right before a public class, method, or variable declaration and begins with `/**` appears in this documentation (you will learn about public variables in Chapter Five).

Language elements

A Condition can be:	ClassName . MethodName ( )	e.g., Vic.stackHasCD( )
A Statement can be:	ClassName . MethodName (someString)	e.g., Vic.say ("Hi");

Anything between /* and */, even over several lines, has no effect on the program.

Exercise 2.19* Write an application program that fills the third and fourth slots in the second sequence. Stop as soon as you run out of CDs in the stack. Print an appropriate message at the beginning of the program. Be sure the program cannot fail.
Exercise 2.20* Write a method `public void fillThree()` for a subclass of Vic: The executor puts a CD in each of its first three slots, but stopping when the stack is empty (hint: similar to part of Listing 2.6). Then revise Listing 2.6 to use that method to have both the first and the second sequence fill their first three slots.

2.5 The If-Else Statement And The Block Statement

Listing 2.7 is a Java program for a machine with at least two sequences. It creates one Vic for each sequence. Then it sends messages to each Vic asking it to take a CD if one is in its first slot, but to put a CD there if no CD is in its first slot. The program uses the **if-else statement** to decide which of two messages to send, `takeCD` or `putCD`. The word `else` in this context essentially means "otherwise" or "if that condition wasn't true". You should compile this program and run it several times to watch it work.

Listing 2.7 An application program using two Vics

```
public class TwoVics
{
   /** Take the CD in slot 1 if there is one, otherwise put a
    *  CD there; repeat for the second sequence, if any. */

   public static void main (String[ ] args)
   {  Vic first;                        // 1
      first = new Vic();                // 2
      if (first.seesCD())               // 3
         first.takeCD();                // 4
      else                              // 5
         first.putCD();                 // 6

      Vic second;                       // 7
      second = new Vic();               // 8
      if (second.seesSlot())            // 9
      {  if (second.seesCD())           // 10
            second.takeCD();            // 11
         else                           // 12
            second.putCD();             // 13
      }                                 // 14
   }  //=====================
}
```

The general form of an if-else statement is as follows. If you want to replace either of these Statements by a group of two or more statements, you put a matched pair of braces around that group. The compiler will then treat the braces plus the group of statements within it as one statement, called a **block statement**:

```
if (Condition)
   Statement
else
   Statement
```

When an if-else statement is executed, the runtime system begins by evaluating the if-condition. If the condition is true, the statement after it is executed and the statement after the `else` is ignored. If the condition is false, the statement after it is ignored and the statement after the `else` is executed. That is, exactly one of them is executed. Figure 2.7 illustrates the meaning of the two varieties of conditional statements you have.

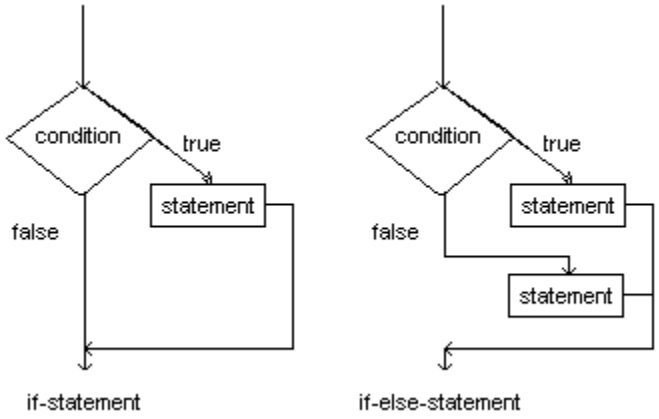


Figure 2.7 Flow-of-control for if and if-else statements

Examples of block statements

The following are some examples that use block statements in an if-else statement. The general principle is that, if the action to be taken when the if-condition is false consists of two or more statements in sequence, you need to put a matched pair of braces around those statements:

```
if (Vic.stackHasCD())
{  putCD();
   moveOn();
}
else
{  moveOn();
   takeCD();
}
```

```
if (seesSlot())
   takeCD();
else
{   backUp();
    takeCD();
    moveOn();
}
```

```
if (seesCD())
{  takeCD();
   moveOn();
   putCD();
}
else
   moveOn();
```

Caution If you forget a pair of braces around a group of statements after `else` or after the if-condition, the program usually produces the wrong result. Even if you indent properly, the program still produces the wrong result. It may not even compile without error. Indentation makes the logic easier for humans to understand; the compiler ignores it.

The physical mechanism

You are perhaps puzzled about how the physical mechanism can do a `moveOn` operation to a non-existent position where `seesSlot()` is false, e.g., when it is already at the seventh slot in its sequence of seven. The physical mechanism has a gripper part that moves in a groove beside the sequence, stopping next to the appropriate slot. Slots are 3/8" apart, so the gripper moves 3/8" each time. The groove extends 3/8" beyond the last slot. So it can move in the groove though it is not to a slot.

In the virtual mechanism, there is no gripper and no groove. The `moveOn` operation adds 1 to the numeric value tracked by its position. So even if there are only seven slots, its position can still be the number 8.

Lab Practice You are probably writing most of the unstarred exercises on a sheet of paper and checking them against the answers. But you should from time to time type up at least a few of the exercises and compile them without first looking at the answers (the answer to an exercise saying "Write a method...for a subclass of Vic" should be put inside your own class that extends Vic and compiled). This tests to make sure you know where the parentheses, braces, and semicolons go, and that you are using correct capitalization, too. These things are not hard, but they do take some practice: **You can't learn stuff if you don't do stuff.**

Language elements

A Statement can be:	if (Condition) Statement else Statement
or:	if (Condition) Statement else { StatementGroup }
or:	if (Condition) { StatementGroup } else Statement
or:	if (Condition) { StatementGroup } else { StatementGroup }

Exercise 2.21 Write a statement that puts a CD in `sam`'s current spot if there is no CD there and the stack has a CD, but otherwise moves on to the next slot.
Exercise 2.22 Write a method `public void shiftForward()` for a subclass of Vic: The executor moves the CD from the next slot into the current slot, but only if the current slot is empty and the next slot exists and is not empty. In either case, the executor is at the next slot after the method finishes executing.
Exercise 2.23** Write an application program that only does things with the second sequence: First take a CD from its first slot and one from its fourth slot; then if at least one of those two slots was not empty, put a CD in its second slot.

2.6 Boolean Methods And The Not-Operator

You have seen two kinds of Vic instance method calls: four actions (`takeCD`, `putCD`, `moveOn`, and `backUp`) and two conditions (`seesSlot` and `seesCD`). And you have seen how to define new actions in an extension of the Vic class. In this section you will see how to define new conditions in an extension of the Vic class.

Preconditions

Programming Style It is good style to have a comment at the beginning of each method definition to describe what happens when the commands in the method are carried out. This header comment should include the precondition of the method, if any. The **precondition** of method M is what each method that calls M must verify is true before making the call, in order that method M produce the expected result (i.e., the result described by its header comment).

This book will henceforth include the precondition in all listings where they are needed. The methods earlier in this chapter that had an unstated precondition are as follows:

- `moveTake` in Listing 2.2. Precondition: There is a slot after the current slot.
- `backPut` in Listing 2.2. Precondition: There is a slot before the current slot.
- `shiftThreeToFive` in Listing 2.5. Precondition: The sequence has at least two slots after the current slot.

The main method in Listing 2.3 fails if its sequence does not have four slots, but that is not a precondition -- `main` is called by the operating system, not by another method.

The not-operator

Sometimes you need to test to see if a slot is empty. The Vic class does not provide a separate condition to do that. But in Java, if you put an exclamation mark in front of a condition, it means the opposite of the condition. `! sam.seesCD()` means "**it is false that** sam sees a CD", i.e., the current slot is empty. (When you read a phrase involving `!` aloud, you could say "not" for the "!").

The first if-else statement in Listing 2.7, repeated below on the left, could be written instead as shown on the right to use this **not-operator** and have the same effect:

```
if (first.seesCD())
   first.takeCD();
else
   first.putCD();
```

```
if ( ! first.seesCD())
   first.putCD();
else
   first.takeCD();
```

Development of the hasTwoOnStack method

The first method defined here is `hasTwoOnStack`, a method that has the executor answer the question, "Are at least two CDs on the stack?" This particular question is to be asked only when the executor is at an empty slot (the precondition). An example of a statement that asks this question and makes use of the answer is the following:

```
if (sue.hasTwoOnStack())
   sue.moveOn();
```

The task is not simple to do, so you should design a plan in ordinary English to do it (or whatever natural language you think in most comfortably) before you try to implement it in Java. Write as if explaining it to a literal-minded not-very-bright person. A reasonable plan is shown in the accompanying design block.

DESIGN of the hasTwoOnStack method
If you do not have any CD at all on the stack, then...
    The answer is `false`; all further analysis is terminated.
Otherwise...
    Put a CD from the stack into the empty slot.
    Make a note of whether you have a CD on the stack at this point.
    Take the CD back onto the stack.
    The answer is whatever you made a note of; no further analysis is needed.

Return statements

When you define a new method asking a true-false question, such as `hasTwoOnStack`, you have to use a special statement to say whether the answer to the question is `true` or `false`. The statement `return false;` means execution of the method gives the answer `false` at that point and ignores the rest of the statements in that method. Similarly, the statement `return true;` means execution of the method gives the answer `true` at that point and ignores the rest of the statements. The key point: Executing a `return` means the rest of the statements in the method are skipped.

A Java implementation of this `hasTwoOnStack` method is in the upper part of Listing 2.8. It looks different from earlier methods in two ways: It has the word `boolean` and it has `return` statements (lines 2, 6, and 10). The word `boolean` in the heading says this method returns a true-false value ("boolean" commemorates a 19th-century logician named George Boole). So you may put a call of this **boolean method** inside the parentheses of an if-statement, as in `if (sam.hasTwoOnStack()) sam.putCD()`.

Listing 2.8 The Checker class of objects, with two boolean methods

```
public class Checker extends Vic
{
   /** Tell whether the stack has at least two CDs.
    *  Precondition: the executor is not at the end
    *  of its slots and the current slot is empty. */

   public boolean hasTwoOnStack()
   {  if ( ! stackHasCD())           // 1
         return false;               // 2
      putCD();                       // 3
      if (stackHasCD())              // 4
      {  takeCD();                   // 5
         return true;                // 6
      }                              // 7
      else                           // 8
      {  takeCD();                   // 9
         return false;               // 10
      }                              // 11
   }  //=====================

   /** Tell whether the executor's current slot and the one
    *  after it exist and have no CD.  No precondition. */

   public boolean seesTwoEmpty()
   {  if ( ! seesSlot())             // 12
         return false;               // 13
      if (seesCD())                  // 14
         return false;               // 15
      moveOn();                      // 16
      if (seesSlot())                // 17
         if ( ! seesCD())            // 18
         {  backUp();                // 19
            return true;             // 20
         }                           // 21
      backUp();                      // 22
      return false;                  // 23
   }  //=====================
}
```

Programming Style Surely you wondered why the executor should bother to take the CD back to the stack (using the if-else statement) before it returns `true` or `false`. The reason is, `someVic.hasTwoOnStack()` is a question you ask some Vic about its current state, and it is not good style to have the process of answering the question alter that state in any way. Quite often, alteration would make the answer useless.

Development of the seesTwoEmpty method

The second boolean method defined here is `seesTwoEmpty`, asking the executor whether both the current slot and the next slot exist and are empty. The task is not simple to do, so you should design a plan in ordinary English to do it before you try to implement it in Java. A reasonable plan is shown in the accompanying design block.

DESIGN of seesTwoEmpty
If you do not have a slot or if you see a CD in your current slot, then...
    The answer is `false`; all further analysis is terminated.
Move on to the next slot (if any).
If you have a slot there and if you do not see a CD in your current slot, then...
    The answer is `true`.
Otherwise...
    The answer is `false`.
Return the answer determined in the previous step, after backing up to the original position.

The definition of the `seesTwoEmpty` method is in the lower part of Listing 2.8. Note that, after the answer to the question is determined, the executor should back up to its original position (lines 19 and 20) so answering the question about its state does not change that state. Suppose you use these methods in some instance method as follows:

```
if (seesTwoEmpty())
   if (hasTwoOnStack())
   {  putCD();
      moveOn();
      putCD();
   }
```

This logic makes sure you can safely put a CD in each of the next two slots. Imagine your surprise if the program were to fail because verifying the safety of the actions was what made those actions unsafe. That is why the executor should restore its state.

Note: Execution of a `return` statement terminates all action in the current method. Therefore, the `else` in the `hasTwoOnStack` logic can be omitted (and its matching braces as well) without affecting what the method does.

Design before you implement in Java

The written designs you have seen for `hasTwoOnStack` and `seesTwoEmpty` give a structure to the ordinary English sentences when a selection of several alternatives is to be made. Specifically, they show the alternatives indented below the conditions that decide which alternative is to be chosen. This kind of design is highly effective in helping you create individual methods that work correctly the first time.

Whenever you have to develop a method to accomplish a task, and the logic is not immediately obvious, you should first design a plan as illustrated here, in accordance with the general programming principle, "If you do not know where you are going, you are not likely to get there."

The translation of a design into a particular programming language is called **coding**. The translation of "Look before you leap" into programming is, "Design before you code."

Language elements	
A Declaration can be:	public boolean MethodName ( ) { StatementGroup }
A Condition can be:	! Condition e.g., !seesCD()
or:	true
or:	false
A Statement can be:	return Condition ;

Exercise 2.24 Write a method `public boolean hasNoSlot()` for a subclass of Vic to mean the opposite of `seesSlot`.
Exercise 2.25 Write a method `public boolean canTakeCD()` for a subclass of Vic: The executor tells whether it could take a CD from its current position if told to do so.
Exercise 2.26 Write a method `public boolean canPutCD()` for a subclass of Vic: The executor tells whether it could put a CD in its current position if told to do so.
Exercise 2.27* Write a method `public void putNextAvailable()` for a subclass of Vic: The executor puts a CD in the next available slot. Precondition: The stack is not empty and either the current slot or the next slot is empty.
Exercise 2.28* Write a method `public boolean seesTwoFilled()` for a subclass of Vic: The executor tells whether its current slot and the next slot exist and both have CDs. Leave the executor in its original state.
Exercise 2.29* Write a method `public boolean hasJustOneOnStack()` for a subclass of Vic: The executor tells whether exactly one CD is on the stack. Precondition: Its current slot exists and is empty. Leave the executor in its original state.

2.7 Boolean Variables And The Assignment Operator

A method that effects a change in the state of one or more objects and does not return a value is an **action method**. Commands such as `sam.moveOn()` and `sue.putCD()` call action methods in the Vic class. These commands tell their executor to do something.

A method that returns a value and does not effect a change in the state of any object (executor or otherwise) is a **query method**. Commands such as `Vic.stackHasCD()` and `sam.seesSlot()` call query methods in the Vic class. These commands ask their executor a question and get an answer in return, without changing the state of the executor.

Technical Note This is not official Java vocabulary, just words to help you see patterns and categories: A method call can be a message to the "executor" to answer a "query" or perform an "action". An action method is a method with "void" just before the method name in the heading. But "void" is not a description of the method, it is just the signal to the compiler that no value is to be returned.

The `hasTwoOnStack` method in Listing 2.8 is a query method because care was taken to put the CD back on the stack before returning. This restored the original state. A true query method answers a question without making any change in any object. Similarly, `seesTwoEmpty` is a query method because care was taken to restore the original position in the sequence. However, the logic in those two methods is clumsy and a bit difficult to follow. It is time you learned about boolean variables.

You have seen how to declare the name of a variable that holds a Vic object. You can also declare the name of a variable that holds a boolean value (i.e., a value that is either `true` or `false`). For instance, `boolean theAnswer` declares `theAnswer` as the name of a place where either `true` or `false` can be stored. You can then have the command `return theAnswer` in a boolean method that has declared `theAnswer` and given it a value of `true` or `false`.

Declaring a variable name within a method does not by itself give the variable a value. You have to **assign** a value to it using the assignment symbol =, which you have seen used in statements such as `sue = new Vic()`. You should almost always assign a value to a variable by the very next statement after you declare it. You can later change the value assigned to the variable if you wish.

Rewrite of the hasTwoOnStack method to use a boolean variable

For the `hasTwoOnStack` method definition in Listing 2.8, after `putCD()` is executed, the value of `stackHasCD()` is either `true` or `false`. And that is the value to be returned. It is clearer to store the value of `stackHasCD()` in a boolean variable (named perhaps `result`), then execute `takeCD()` before returning the value of `result`.

The upper part of Listing 2.9 shows how the `hasTwoOnStack` method can be rewritten using a boolean variable. The assignment to `result` in the method (line 5) could be written instead as follows, with exactly the same effect as `result = stackHasCD()`, but it is wasteful to do so:

```
if (stackHasCD())
   result = true;
else
   result = false;
```

Listing 2.9 Rewrites of two boolean methods in Checker

```
/** Tell whether the stack has at least two CDs.
 *  Precondition: the executor is not at the end
 *  of its slots and the current slot is empty. */

public boolean hasTwoOnStack()
{  if ( ! stackHasCD())                  // 1
      return false;                      // 2
   putCD();                              // 3    modify state
   boolean result;                       // 4
   result = stackHasCD();                // 5
   takeCD();                             // 6    restore state
   return result;                        // 7
}  //=====================

/** Tell whether the executor's current slot and the
 *  one after it exist and have no CD.  No precondition. */

public boolean seesTwoEmpty()
{  if ( ! seesSlot())                    // 8
      return false;                      // 9
   if (seesCD())                         // 10
      return false;                      // 11
   moveOn();                             // 12   modify state
   boolean valueToReturn;                // 13
   if ( ! seesSlot())                    // 14
      valueToReturn = false;             // 15
   else                                  // 16
      valueToReturn = ! seesCD();        // 17
   backUp();                             // 18   restore state
   return valueToReturn;                 // 19
}  //=====================
```

Rewrite of the seesTwoEmpty method to use a boolean variable

In the earlier `seesTwoEmpty` method definition, after `moveOn()` is executed, the value to be returned is `true` if `seesSlot()` is `true` and `seesCD()` is `false`, otherwise the value to be returned is `false`. It is clearer to store that true/false value in a boolean variable (named perhaps `valueToReturn`), then execute `backUp()` before returning the value of `valueToReturn`. This is done in the lower part of Listing 2.9.

You should carefully compare the coding in Listing 2.9 with the logic in the earlier Listing 2.8 to see the difference. In each case, the first few lines are the same, down to where the state of the executor changes. The line numbers are so future discussion can refer to those lines.

The crucial difference between Listing 2.8 and Listing 2.9 for the `seesTwoEmpty` method occurs after `moveOn`: In Listing 2.8, if `seesSlot()` is `true` and `seesCD()` is `false`, the executor backs up and returns `true`, otherwise the executor backs up and returns `false`. By contrast, in Listing 2.9, if `seesSlot()` is `true` and `seesCD()` is `false`, the executor simply stores `true` in the variable, otherwise the executor stores `false` in the variable (lines 14-17). Then, in either case, the executor backs up and returns whatever value it previously stored in the variable (line 19).

Programming Style All methods that return a value are to restore the original state of all objects before returning, unless explicitly stated otherwise (and then only if there is a very good reason). This is an important point of good style.

Pictorial representation of values in variables

Figure 2.8 shows a picture of what might be stored in RAM after you (a) create an object for `pam` to refer to (indicated by an arrow from `pam` to the internal description of the object), and (b) assign `true` to a boolean variable named `result`. The Vic object keeps track of the stack of CDs, its sequence of slots, and its position in the sequence. A newly-created Vic always starts at position number 1.

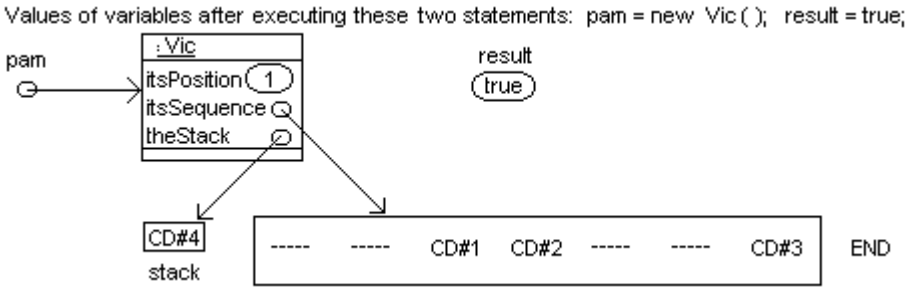


Figure 2.8 Object references as arrows to the objects

Each rectangle in the figure represents one variable. An arrow is shown leaving each variable which refers to an object. A standard notation for an object is a box with three parts, where the top part has the class of the object underlined, the middle part lists variables which are part of the object, and the bottom part is often left blank.

Are you skipping many of the unstarred exercises? That is not a good idea, you know. You cannot just read this material as if it were a novel. You will start to become confused by so many new concepts. At a minimum, spend just two minutes on each unstarred exercise and then, done or not, check out its answer at the end of the chapter. This will take less than ninety minutes per chapter, you will learn a lot more, and that learning will help you complete your graded course assignments faster and better.

Language elements		
A Statement can be:	boolean VariableName ;	e.g., boolean valueToReturn;
A Condition can be:	VariableName	e.g., valueToReturn

Exercise 2.30 Rewrite the `hasTwoOnStack` method of Listing 2.9 to have just one return statement. Precondition: Its current slot exists and is empty. Hint: You will need `else` followed by braces.
Exercise 2.31 Write a query method `public boolean atMostOneOnStack()` for a subclass of Vic: The executor tells whether the stack has either zero or one CD on it. Precondition: Its current slot exists and is empty. Only have one return statement.
Exercise 2.32* Write a query method `public boolean hasOneBefore()` for a subclass of Vic: The executor tells whether a CD is in the slot before the current slot. Have only one return statement. Precondition: The executor is not at the first slot in its sequence.
Exercise 2.33** Write an application program to move the first two available CDs in the first sequence of slots to the stack. Precondition: That sequence has at least two CDs in its first four slots. Hint: First create a boolean variable `alreadyTookCD`. Make it `true` if you take the first CD and `false` if not. Use it to decide what to do at later slots.

2.8 Boolean Operators And Expressions; Crash-Guards

If you put **&&** between two true-false expressions it means "and". That is, the combined expression is true only when both of its two parts are true. The two parts are called the **operands** of `&&`. Now you can rewrite these lines 14-17 of Listing 2.9

```
if ( ! seesSlot())
   valueToReturn = false;
else
   valueToReturn = ! seesCD();
```

more compactly and more clearly, but with exactly the same effect, as follows:

```
valueToReturn = seesSlot() && ! seesCD();
```

If you put **||** between two true-false expressions, it means "or". That is, the combined expression is true when either the first operand of `||` is true or the second operand of `||` is true, or both. Now you can rewrite these lines 8-11 of Listing 2.9

```
if ( ! seesSlot())
   return false;
if (seesCD())
   return false;
```

more compactly and more clearly, but with exactly the same effect, as follows:

```
if ( ! seesSlot() || seesCD())
   return false;
```

The three symbols `&&` and `||` and `!` are called **boolean operators** because they operate on boolean (true-false) expressions to produce a boolean expression. The ! operator has only one operand.

Short-circuiting boolean expressions

In the expression `seesSlot() && ! seesCD()`, if it happens `seesSlot()` is `false`, the runtime system does not look at the operand after the `&&` and the result

`false` is obtained. This is good, because otherwise the program would fail. And if `seesSlot()` is false in the expression `! seesSlot() || seesCD()`, the operand after the `||` is not looked at and the result `true` is obtained, so again the program does not fail.

In each case, the first operand of the expression is a "crash-guard" for the second operand, in that it prevents evaluation of the second operand in precisely those situations where the evaluation would crash the program. The `&&` and `||` operators **short-circuit** the condition they form: The second operand of an `&&` or `||` expression is not evaluated if the first operand by itself determines its truth or falseness. From this discussion you can see that the `seesTwoEmpty` method of Listing 2.9 can be written more compactly as shown in Listing 2.10.

Listing 2.10 Improved replacement for the seesTwoEmpty method in Checker

```
public boolean seesTwoEmpty()
{  if ( ! seesSlot() || seesCD())
      return false;
   moveOn();
   boolean valueToReturn;
   valueToReturn = seesSlot() && ! seesCD();
   backUp();
   return valueToReturn;
}  //=====================
```

Caution Put parentheses around an expression formed with `||` or `&&` if the expression does not stand alone as an if-condition or as something assigned to a boolean variable. That is, it should have parentheses around it when it is an operand of `||` or `!` or `&&`. `(x || y) && z` is different from `x || (y && z)`. Also, if you want `!` to apply to an expression that is not a simple method call or variable, you should put parentheses around that expression. There are times when the parentheses are not needed, but it is safer to always use them in such cases.

Language elements	
A Condition can be:	Condition && Condition
or:	Condition || Condition

Exercise 2.34 Rewrite the answer to Exercise 2.25 (`canTakeCD`) so that the body is a single return statement.
Exercise 2.35 Rewrite the answer to Exercise 2.26 (`canPutCD`) so that the body is a single return statement.
Exercise 2.36 Explain why the following causes compilation errors:

```
if(sam.seesSlot)
   sam.putCD();
   sam.moveOn();
else
   sam.takeCD();
```

Exercise 2.37* Rewrite `seesTwoEmpty` in Listing 2.10 so it contains only one return statement.
Exercise 2.38** Write a method `public void shiftDown()` for a subclass of Vic: The executor moves the CD in its current slot into the following slot, except no change is to be made at all if there is no empty following slot or if there is no CD in the current slot. Avoid all program failures. Use boolean operators where appropriate.

2.9 Getting Started With UML Class Diagrams And Object Diagrams

The standard method for drawing a model of a program uses the Unified Modeling Language, **UML** for short. A **class diagram** is a picture that shows the classes the program uses and some relations between them. For instance, Figure 2.9 is a class diagram for the TwoToFour application program in the earlier Listing 2.4.

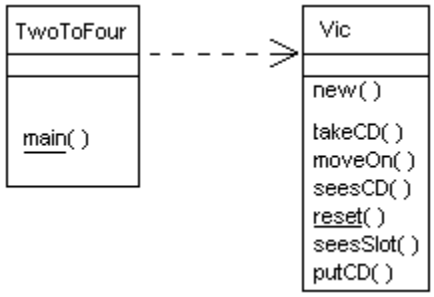


Figure 2.9 UML class diagram for TwoToFour

This diagram has two class boxes with a dependency indicated by a dotted-line arrow. A **class box** is a rectangle with three parts separated by horizontal lines. A **dependency** means the one class uses the methods in the class pointed to.

The top part of the class box contains the name of the class (not underlined, to distinguish it from an object box as shown in Figure 2.8). The middle part (between the two lines) is for attributes. It is blank in this diagram. The bottom part lists the method calls used in the program. The names of class methods are to be underlined.

UML generalization

Figure 2.10 shows the class diagram for the program BringThreeBack in the earlier Listing 2.3. The diagram shows the two classes mentioned in this program, namely, BringThreeBack itself plus the SmartVic class. Since the `moveOn` and `reset` methods that BringThreeBack uses are inherited from the Vic class, the class diagram shows the Vic class too. The solid line with the triangular head signals that SmartVic inherits from Vic. UML calls it a **generalization**.

The three arrows in the class diagram can be read as follows:

1. BringThreeBack uses Vic.
2. BringThreeBack uses SmartVic.
3. SmartVic is a kind of Vic.

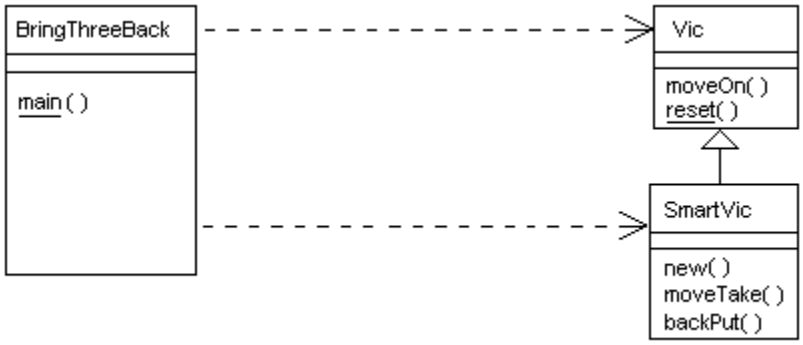


Figure 2.10 UML class diagram for BringThreeBack

UML allows you a great deal of choice in how much detail you choose to show in a class diagram. The class diagrams shown in this section are at the minimum level of detail. Chapter Three describes some additional choices you have, but you should first practice this minimum-level kind of class diagram for a while, defined as follows:

1. Write down a list of all of the methods mentioned in the class or classes you want to diagram (in the examples, we only want to diagram one application program).
2. Draw a rectangle for each class in which one or more of those methods are defined.
3. Divide the rectangle horizontally into three parts, with the name of the class in the top part.
4. List each method from step 1 in the bottom part of the rectangle for the class that contains its definition. Underline `main` and other class methods (such as `reset`).
5. Draw a solid-line-triangle-arrowhead from a class to its superclass if both appear in the diagram.
6. Draw a dotted-line-arrow from a class to each other class (besides its superclass) whose methods or instances it mentions.

UML is also used to diagram individual objects. The earlier Figure 2.8 is a **UML object diagram**. For that diagram, an alternate way of representing the object `pam` refers to is to put `pam:Vic` in the top part of the box and omit the arrow from `pam`. That is, a rectangle that represents a particular Vic object has `:Vic` or `variable:Vic` underlined in the top part of the rectangle; a rectangle that represents a class just has the class name and it is not underlined.

Client-server relations

When one class provides services (generally methods) that another class uses, the first class is said to be a **server** for the second class, and the second is said to be a **client** of the first. In this client-server relation, for instance, the client may have parameters that are instances of the server class, or may return instances of the server class, or may use the methods of the server class locally within its coding. Both dependencies and generalizations are kinds of client-server relations.

Each method that provides a service should have a descriptive comment saying what its effect is in all situations in which the method can be called (except when the name of the method alone makes it quite clear). This is the **specification** for the method. A **fault** in a program (colloquially known as a **bug**) is a failure of a method to do in some situation what its specification says it does in that situation.

A precondition on a method states a condition that must be true when the method is called. If some method M calls a method P when P's precondition is false, then it is a bug in M, not in P. Essentially, a precondition shifts the responsibility for verifying something from the called method to the calling method. So it is a deficiency in P to verify its preconditions and a fault in M to not verify P's preconditions. An implied part of each method's specification is that it does not call any method in a situation when the called method's precondition is false.

For example, if the coding for a method begins with `if ( ! seesSlot()) say...` then its specification should not say "Precondition: its current slot exists"; it should instead say "If its current slot does not exist, say... otherwise...".

Exercise 2.39 Draw the UML diagram for the program in Listing 2.1.
Exercise 2.40* Draw the UML diagram for the program of Exercise 2.9.

2.10 Analysis And Design Example: Complex Conditionals

Perhaps you have a need to write a method to add to a subclass of Vic, to remove the first two CDs from a sequence known to have at least two CDs in the next three slots. Since this is not a simple thing to do, you should begin with an analysis of the situation:

Question 1: What do you do first? *Answer:* It depends.
Question 2: Depends on what? *Answer:* On whether this first slot contains a CD.
Question 3: What do you do first if it does? *Answer:* Take the CD from there and then move on to the second slot.
Question 4: What do you do first if it does not? *Answer:* Move on to the second slot.

This gives you the following overall if-else design for your logic:

```
If the first slot contains a CD then...
   Take that CD out of the slot and move to the second slot.
   // do more in Case 1
Otherwise...
   Move to the second slot.
   // do more in Case 2
```

In both cases, you end up in the second slot. Next decide whether what you do in Case 1 is the same as what you do in Case 2. If that were so, you would handle that common case after the if-else statement (change the design by removing the two comments and putting `// do more either case` at the end). But that is not so.

Now you figure out what comes next in each case. Case 2 is the easier situation, since at this point you know a CD is in each of the next two slots (remember, the precondition is that you have at least two CDs in the three slots, and there was no CD in the first slot). So you know in Case 2 what actions to take without testing any more conditions:

```
// do more in Case 2 expands to:
Take the CD from the second slot.
Move to the third slot and take the CD from that slot.
```

In Case 1, you also have to move on to the second slot, because you have one CD left to find and it is in one of the next two slots. After you move on to the second slot, you realize what you do again depends on what you find. Analyze it with the same four key questions:

Question 1: What do you do first? *Answer:* It depends.
Question 2: Depends on what? *Answer:* On whether this second slot contains a CD.
Question 3: What do you do first if it does? *Answer:* Take the CD from there.
Question 4: What do you do first if it does not? *Answer:* Move on to the third slot and take the CD from that slot.

From this you can develop a design of the logic. Note that the design is in English, not in Java. You need to review the design and make sure it is logically correct and covers all of the possibilities. This is much easier to do in a language you know well, e.g., English.

```
// do more in Case 1 expands to:
If the second slot contains a CD then...
   Take the CD from that slot.
Otherwise...
   Move to the third slot and take the CD from that slot.
```

You should combine the parts of this design into one unit so you can study it further before you try to translate it to Java. The accompanying design block does this. One

small change is made: Since Case 2 turned out to be shorter and simpler than Case 1, it is presented first in the design. Putting the simpler alternative first is normally clearer.

DESIGN of takeTwoOfThree

1. If you do not see a CD in the current slot then...
    1a. Move on to the second slot and take that CD.
    1b. Move on to the third slot (there must be a CD in that slot).
2. Otherwise...
    2a. Take the CD in the current slot.
    2b. Move on to the second slot.
    2c. If you do not see a CD in the second slot, then...
        Move on to the third slot (there must be a CD in that slot).
3. Take the CD in the current slot (which is either the second or the third slot).

After you work out the design in this form, you need to read it over several times, even recite it out loud and listen to it, to make quite sure there are no errors in the logic. Read again the specification of what the method is to do, to be sure your design fulfills the requirements. This can then be translated into Java as the method shown in Listing 2.11. Figure 2.11 shows which statements are actually executed for the three possibilities.

Listing 2.11 An instance method for a subclass of Vic

```
/** Take the first two available CDs.
 *  Precondition: the first 3 slots contain at least 2 CDs. */

public void takeTwoOfThree()
{  if ( ! seesCD())                          // design step 1
   {  moveOn();                              // design step 1a
      takeCD();
      moveOn();                              // design step 1b
   }
   else                                      // design step 2
   {  takeCD();                              // design step 2a
      moveOn();                              // design step 2b
      if ( ! seesCD())                       // design step 2c
         moveOn();
   }
   takeCD();                                 // design step 3
}  //======================
```

// THE CODING	CD IN 1ST AND 2ND	CD IN 1ST AND 3RD	CD IN 2ND AND 3RD
if (! seesCD())	! seesCD() is false	! seesCD() is false	! seesCD() is true
{ moveOn();			moveOn();
takeCD();			takeCD(); // slot 2
moveOn();			moveOn();
}			
else			
{ takeCD();	takeCD(); // slot 1	takeCD(); // slot 1	
moveOn();	moveOn();	moveOn();	
if (! seesCD())	! seesCD() is false	! seesCD() is true	
moveOn();		moveOn();	
}			
takeCD();	takeCD(); // slot 2	takeCD(); // slot 3	takeCD(); // slot 3

Figure 2.11 Three possible sequences of actions for Listing 2.11

Now read this section down to here one more time and note well the process described here. For all but the simplest Java methods, if you apply this process to code a method to perform a given task, you will find that you spend somewhat longer before you get to the coding part but that you spend far less time getting the coding part right.

The Dangling Else Trap

Look at the following two methods that could be in a subclass of Vic. The first three lines of the bodies are the same in both cases; they differ beginning with `else`:

```
public void sayWhenSlotEmpty()
{  if (seesSlot())
      if (seesCD())
         takeCD();
      else
         Vic.say ("No CD!");
}  //======================
```

```
public void sayWhenNoSlot()
{  if (seesSlot())
      if (seesCD())
         takeCD();
   else
      Vic.say ("No slot!");
}  //======================
```

From the indentation and the method names, it is clear that the first method is to print a message only when there is a slot that contains a CD. And the second method is to print a message only when there is no slot at all (i.e., the Vic is at the end of its sequence). The first method behaves as it is intended. But the second method behaves just like the first method except for the words in the message.

General principle: The compiler pays no attention to indentation. Indentation is only to make it easy for you and other people to understand your logic. And the compiler does not understand the words inside the calls of `Vic.say`. So the compiler has no choice but to interpret these two methods the same; they have the same structure. The compiler's rule in such a case is to match the `else` with the most-recently occurring `if`.

Caution It is best never to write `if (Condition) if` in your logic. Rephrase it; if nothing else, put braces around the subordinate if-statement. Doing this guarantees you will never fall prey to this **Dangling Else Trap**. Applying this rule to the two preceding examples gives the following (mark ye well the position of the braces):

```
public void sayWhenSlotEmpty()
{  if (seesSlot())
   {  if (seesCD())
         takeCD();
      else
         Vic.say ("No CD!");
   }
}  //======================
```

```
public void sayWhenNoSlot()
{  if (seesSlot())
   {  if (seesCD())
         takeCD();
   }
   else
      Vic.say ("No slot!");
}  //======================
```

Multiway selection format

The `verbosePutCD` method shown in Listing 2.12 has an if-else structure that occurs with some frequency in developing methods. This logic prints a different message in each of four different cases. Only in the fourth case does it actually put a CD in the slot.

The indentation used in this method is the **Multiway Selection** format. It makes it clear to the reader that exactly one of the four cases will be selected. Standard indentation would require two more lines and indent two additional levels. That is not as readable.

Listing 2.12 An instance method illustrating multiway selection

```
/** Try putCD().  Print a description of the situation. */

public void verbosePutCD()   // in a subclass of Vic
{  if ( ! seesSlot())
      Vic.say ("There is no slot");
   else if (seesCD())
      Vic.say ("The slot is full");
   else if ( ! stackHasCD())
      Vic.say ("I have no CD");
   else
   {  putCD();
      Vic.say ("Mission accomplished!");
   }
}  //======================
```

Some programming languages offer a special multiway selection statement using e.g. the special word ELSIF. The designers of Java apparently felt programmers could just use the normally-two-way if-else with the modified indentation -- form follows function.

Programming Style There are two commonly-used styles in the placement of braces after an if- or while- condition, or method heading, or the equivalent. One is the Allman style, used in this book. The other is the Kernighan & Ritchie style, where the opening brace is placed at the end of the line that has the `if` or `while` or method name. It is a matter of personal preference which one you use. It is also a matter of preference whether you have a blank line (other than perhaps the beginning brace) between the controlling phrase and the subordinate part. However, it is wise to use the Allman style until late in your first course of programming, because beginners find it easier to have their braces match up when they are vertically lined up with each other and everything between is indented.

Exercise 2.41 Write a method `public void downShift()` for a subclass of Vic: The executor takes the CD from the current slot and puts it in the very next slot. However, do not do anything if there is no current slot or no CD in it, and leave the CD on the stack if there is no following slot or it has a CD in it already.
Exercise 2.42 Rewrite the answer to Exercise 2.21 in the multiway selection format. Pay close attention to indenting.
Exercise 2.43** Write an application program that moves the first available CD in the first sequence of slots to the stack. Do not go any further in the sequence than necessary. Precondition: The sequence is known to have at least one CD in its first four slots.
Exercise 2.44** Revise the `takeTwoOfThree` method in Listing 2.11 to remove all preconditions.
Exercise 2.45* Look ahead to Listing 3.10 and rewrite the body of the lengthy while-statement in the multiway selection format. Pay close attention to indenting.

2.11 Review Of Chapter Two

Listing 2.5, Listing 2.9, and Listing 2.10 illustrate almost all Java language features introduced in this chapter.

About the Java language:

- The compiler ignores anything between `/*` and `*/`, even over several lines. By contrast, the compiler ignores anything after `//` up to the end of the physical line.
- You cannot "run" an object class that has no main method. You can only "run" a class with `java ClassName` when the class has a main method to run. You may test out an object class X (i.e., a class with instance methods) by executing some other class's main method that creates objects of type X and sends them messages.
- The words `true` and `false` are constants that indicate the only two possible boolean values. They are called "reserved words" in Java: You cannot use them to name a method, variable, or class, just as with the keywords `boolean` and `return`. A **condition** is an expression whose value is either `true` or `false`.
- The symbol for "not" is an exclamation point, for "and" is `&&`, and for "or" is `||`. In expressions involving two or more **boolean operators**, you should use parentheses liberally to make sure the meaning is clear. If the first **operand** `x` is true in `x || y`, or is false in `x && y`, then the second operand `y` is not evaluated, and the result of the expression is the value of `x`. This is **short-circuiting**.
- See Figure 2.12 and Figure 2.13 for the remaining new language features. `StatementGroup` means zero or more statements in sequence.

`if (Condition)` `  Statement`	**statement** that executes the subordinate Statement if the Condition is true

`if (Condition)`
`  Statement`
`else`
`  Statement`
	statement that executes only the first Statement if the Condition is true, but executes only the second Statement if the Condition is false

`{  StatementGroup` `}`	**statement** that can replace the one Statement in ifs and if-elses and others
`ClassName.MethodName (Expression);`	**statement** that calls a class method with one parameter
`boolean VariableName;`	**statement** that creates a true-false variable
`VariableName = Condition;`	**statement** that assigns a value of true or false to a boolean variable
`return Condition;`	**statement** that returns a true-false value; it terminates the method it is in

Figure 2.12 Statements introduced in Chapter Two

`public boolean MethodName()`
`{  StatementGroup`
`}`
	declaration of a method that returns a true-false value, i.e., a Condition

Figure 2.13 Method declaration introduced in Chapter Two

Caution Do not put any statement directly after a `return` statement if it is appropriately indented the same (i.e., is subordinate to the same condition). Such a statement can never be executed.

Other vocabulary to remember:

- The expression you put in the parentheses of a method call is a **parameter**, also known as an **argument** of the method call.
- The parts of an if-statement or if-else statement are named the **condition** (in parentheses after `if`), the first **subordinate part** (directly after those parentheses), and the second subordinate part (after `else` if present). If one of the subordinate parts has two or more statements, it should be enclosed in braces (to make it a **block statement**).
- The **javadoc** standard for commenting methods puts a description of each public method just before the method heading, beginning with `/**` and ending with `*/`.
- The **precondition** for a method is what has to be true when the method begins execution, in order that the method produce the expected result.
- The translation of a particular design into a compilable class is called **coding**. "Design before you code" is as basic a principle as "Think before you act."
- An **action method** changes the state of one or more objects but does not return a value. A **query method** returns a value but does not change the state of any object. Try to avoid methods that do both.
- You can convey the information that a choice is being made from among three or more alternatives by putting an `if` immediately after `else` on the same physical line, rather than indented on the following line. This **multiway selection format** gets around the limitations of using a language feature which offers only a choice from two alternatives.

About Vic methods (developed for this book):

- `new Vic()` creates a Vic object.
- You can send four action messages to a Vic: `sam.putCD()`, `sam.takeCD()`, `sam.moveOn()`, and `sam.backUp()`.
- You can ask two questions of a Vic: `sam.seesCD()` and `sam.seesSlot()`.
- You have three Vic class methods: `Vic.stackHasCD()`, `Vic.reset(args)`, and `Vic.say("whatever")`. The `reset` method uses whatever strings of characters follow `javac ProgramName` on the command line to initialize the sequences of CDs. But a random arrangement is used if there are no such **command-line arguments** or the `reset` method is not executed until after some Vic object has been created.
- If `sam.seesSlot()` is `false`, then `sam.putCD()`, `sam.takeCD()`, `sam.moveOn()`, and `sam.seesCD()` all cause the program to fail (i.e., gracefully terminate execution). Also, the `backUp` message causes the program to fail if the Vic object is positioned at its first slot. You should avoid letting this happen. If you let it happen, your program is not **robust**, since it does not handle unexpected input well. Ideally, all application programs should be written so that they cannot fail.

About UML notation:

- A **UML class diagram** uses **class boxes** -- rectangles divided into three parts. The top part has the class name and the bottom part lists any of its method calls you wish to mention, with `main` and other class methods underlined.
- A **dependency** of the form `X uses Y` is indicated by an arrow with a dotted line.
- A **generalization** of the form `X is a kind of Y` is indicated by an arrow with a solid line and a big triangular head.
- A **UML object diagram** also uses three-part rectangles. The top part has a colon followed by the class name, optionally with the name of the variable that refers to it before the colon. The middle part generally lists attributes of the object.

Answers to Selected Exercises

2.1
```
public class ThirdToFirst
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        Vic cat;
        cat = new Vic();
        cat.moveOn();       // to slot 2
        cat.moveOn();       // to slot 3
        cat.takeCD();
        cat.backUp();       // to slot 2
        cat.backUp();       // to slot 1
        cat.putCD();
    }
}
```
2.2
```
public class PutThree
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        Vic cat;
        cat = new Vic();
        cat.putCD();
        cat.moveOn();       // to slot 2
        cat.putCD();
        cat.moveOn();       // to slot 3
        cat.putCD();
    }
}
```
2.3
```
public class TakeSecondAndFourth
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        Vic cat;
        cat = new Vic();
        cat.moveOn();       // to slot 2
        cat.takeCD();
        cat.moveOn();       // to slot 3
        cat.moveOn();       // to slot 4
        cat.takeCD();
    }
}
```
2.6
```
public static void main (String [ ] args)  // rewrite of Exercise 2.1
{   Vic.reset (args);
    SmartVic cat;
    cat = new SmartVic();
    cat.moveOn();
    cat.moveTake();     // to slot 3
    cat.backUp();
    cat.backPut();      // to slot 1
}
```
2.7 Put "cat.moveTake();" in place of the two lines beginning at the comment // to slot 2 and also in place of the two lines beginning at the comment // to slot 4

2.8
```
public void movePut()
{   moveOn();
    putCD();
}
```
2.9
```
public class ThirdToFifth
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        SmartVic cat;
        cat = new SmartVic();
        cat.moveOn();       // to slot 2
        cat.moveTake();     // to slot 3
        cat.moveOn();       // to slot 4
        cat.movePut();      // to slot 5
    }
}
```
2.13 Replace the last two statements by the following:
```
sue.moveOn();
if (sue.seesSlot())
{   sue.takeCD();
    sue.backPut();
}
```

2.14
```
public class TakeFourthAndFifth
{   public static void main (String [ ] args)
    {   Vic.reset (args);
        Vic cat;
        cat = new Vic();
        cat.moveOn();
        cat.moveOn();
        cat.moveOn();           // to slot 4
        if (cat.seesSlot())
        {   cat.takeCD();
            cat.moveOn();       // to slot 5
            if (cat.seesSlot())
                cat.takeCD();
        }
    }
}
```

2.15
```
public void swapTwo()
{   if (seesSlot())
        if (seesCD())
        {   moveOn();
            if (seesSlot())
                if (seesCD())
                {   takeCD();
                    backUp();
                    takeCD();
                    moveOn();
                    putCD();
                    backUp();
                    putCD();
                }
        }
}
```

2.21
```
if (sam.seesCD())
    sam.moveOn();
else
    if (Vic.stackHasCD())
        sam.putCD();
    else
        sam.moveOn();
```

2.22
```
public void shiftForward()
{   if (seesCD())
        moveOn();
    else
    {   moveOn();
        if (seesSlot())         // this test must be made first
            if (seesCD())
            {   takeCD();
                backUp();
                putCD();
                moveOn();
            }
    }
}
```

2.24
```
public boolean hasNoSlot()
{   if (seesSlot())
        return false;
    else
        return true;
}
```

2.25
```
public boolean canTakeCD()
{   if (seesSlot())
        if (seesCD())
            return true;
    return false;
}
```

2.26
```
public boolean canPutCD()
{   if ( ! seesSlot())
        return false;
    if (seesCD())
        return false;
    if (stackHasCD())
        return true;
    else
        return false;
}
```
2.30
```
public boolean hasTwoOnStack()
{   boolean result;
    if ( ! stackHasCD())
        result = false;
    else
    {   putCD();
        result = stackHasCD();
        takeCD();
    }
    return result;
}
```
2.31 This is just the opposite of the preceding exercise, so replace its two assignments by:
```
result = true;
result = ! stackHasCD();
```
2.34
```
public boolean canTakeCD()
{   return seesSlot() && seesCD();
}
```
2.35
```
public boolean canPutCD()
{   return (seesSlot() && ! seesCD()) && stackHasCD();
}
```
2.36 The empty parentheses are missing at the end of the sam.seesSlot method call.
Also, you must put the two statements subordinate to if inside matching braces.

2.39 It is the same as Figure 2.9 except that you remove three items from the Vic box (reset, seesSlot, seesCD) and add backUp(). Also replace TwoToFour by MoveOne.

2.41
```
public void downShift()
{   if (seesSlot() && seesCD())
    {   takeCD();
        moveOn();
        if (seesSlot())
            putCD();
    }
}
```
2.42
```
if (sam.seesCD())
    sam.moveOn();
else if (Vic.stackHasCD())
    sam.putCD();
else
    sam.moveOn();
```

3 Loops and Parameters

Overview

The commands in the Vic class operate physical clamps and springs to move the CDs around. In the context of this Vic software, you will learn about several new Java features:

- Section 3.1 presents the while-statement, which can be used to send a Vic to the end of its sequence. In general, a while-statement allows you to repeat an action many times.
- Section 3.2 explains the `equals` method in the Sun standard library String class.
- Sections 3.3-3.6 introduce more language features: private methods, the default executor, and parameters for passing extra information to a method. You only need study through Section 3.6 to understand the material in the rest of this book.
- Sections 3.7-3.8 describe and illustrate a highly reliable method for developing the logic of complex methods relatively quickly and with few errors.
- Sections 3.9-3.10 cover enrichment topics: Turing machines and Javadoc tags.

3.1 The While Statement

The `fillFourSlots` method that follows asks the executor to put a CD in the next four available slots, except that the four if-statements make the method stop early if the executor runs out of slots:

```
public void fillFourSlots()     // in a subclass of Vic
{  if (seesSlot())
   {  putCD();
      moveOn();               // move to slot 2
      if (seesSlot())
      {  putCD();
         moveOn();            // move to slot 3
         if (seesSlot())
         {  putCD();
            moveOn();         // move to slot 4
            if (seesSlot())
               putCD();
         }
      }
   }
}  //======================
```

Three of the four if-statements have a block statement for the subordinate part, i.e., several statements enclosed in a matching pair of braces. When an if-condition is false, none of the statements within its block will be executed. Clearly, the method would be quite lengthy if you wanted to put a CD in each of the first six slots. And what if you wanted to put a CD in every single slot?

The following method contains a new Java statement that repeats its two inner statements any number of times, until the condition `seesSlot()` becomes false. It fills every slot possible. Note that it is far shorter than `fillFourSlots`, even though it does far more:

```
public void fillSlots()     // in a subclass of Vic
{  while (seesSlot())
   {  putCD();
      moveOn();
   }
}  //======================
```

The usual format of a **while-statement** is:

```
while (Condition)
{  Statement...
}
```

Subordinate statements

The statements in the block are **subordinate** to the while. If you have only one subordinate statement in the block you may omit the braces, as for an if-statement. Note that we boldface the word `while` as a signal it requires a subordinate statement.

The condition in the parentheses after `while` is the **continuation condition**. The meaning of a while-statement can be expressed as follows:

1. If the continuation condition is true, then
   1a. Execute all subordinate statements in sequence.
   1b. Repeat this process from Step 1.

Figure 3.1 gives a pictorial description of this action.

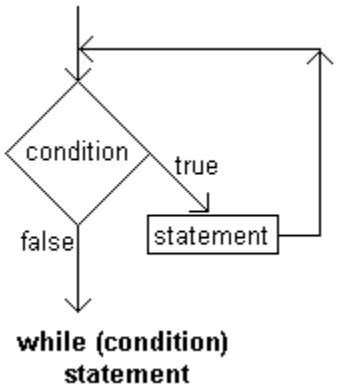


Figure 3.1 Flow-of-control for the while-statement

Examples of the while-statement

The `toggleCDs` method in Listing 3.1 gives another illustration of the while-statement. While moving to the end of the sequence, it switches the status of each slot by (a) removing a CD if the slot contains one, otherwise (b) putting a CD in the slot if the stack contains one.

Listing 3.1 A method that uses the while statement

```
public void toggleCDs()      // in a subclass of Vic
{  while (seesSlot())
   {  if (seesCD())
         takeCD();
      else
         putCD();
      moveOn();
   }
}  //======================
```

Programming Style The following statement as the subordinate part of the while-statement in `toggleCDs` would do exactly the same thing, but it is not good style to have the same statement at the end of both alternatives of an if-else statement. Such a statement should be "factored out" as in Listing 3.1:

```
if (seesCD())   // unfactored if-else statement
{  takeCD();
   moveOn();
}
else
{  putCD();
   moveOn();
}
```

The following main method empties the first slot of every sequence. Each execution of `sequence = new Vic()` makes `sequence` refer to the next sequence of slots. When `sequence.seesSlot()` is `false`, you have no more sequences to process:

```
public static void main (String[ ] args)
{  Vic sequence;
   sequence = new Vic();

   while (sequence.seesSlot())
   {  sequence.takeCD();
      sequence = new Vic();
   }
}  //======================
```

Programming Style It is good style to indent the subordinate part of an if-statement or while-statement by one extra tab position. That makes it clear to someone who reads the program what statements are subordinate to what other statements. This is especially useful with if-statements within if-statements or within while-statements. Note that all of the source code in this book indents after each boldfaced word.

Language elements

A Statement can be:	while (Condition) Statement
or:	while (Condition) { StatementGroup }

Exercise 3.1 Write a method `public void removeAllCDs()` for a subclass of Vic: The executor removes all the CDs from its slots and puts them on the stack. Leave the executor at the end of its sequence of slots.
Exercise 3.2 Write a method `public void toLastSlot()` for a subclass of Vic: The executor advances to the last slot in its sequence. Precondition: The executor has at least one slot somewhere.
Exercise 3.3 Write a method `public void takeOneBefore()` for a subclass of Vic: The executor backs up until it sees a slot with a CD in it, then takes it. Precondition: There will be a filled slot somewhere before the current position of the executor.
Exercise 3.4* Write an application program that fills the first slot of every sequence of slots for which the first slot is empty. Stop when the stack becomes empty.
Exercise 3.5* Write a method `public void fillOneSlot()` for a subclass of Vic: The executor advances to the next available empty slot and puts a CD in it. If this is impossible, just have the executor advance to the next available empty slot or, if all slots are filled, to the end of the sequence. "Next available" includes the current slot.
Note: All instance methods in the exercises for Chapter Three that ask you to do something for all of the executor's slots apply only to the current slot and those after it. For instance, Exercise 3.1 means the executor removes all CDs from this and later slots. You must ignore any previous slots, since you have no way to tell whether they exist.

3.2 Using The Equals Method With String Variables

The `fillSlots` method of the previous section leaves the Vic at the end of its sequence of slots. It cannot back up to the beginning, because it would not know when to stop. That makes it a not very useful Vic. This can be fixed by having the executor make a note of its current position before going through the sequence to fill the slots. Then it will be able to back up to that initial position.

The method call `sam.getPosition()` returns an object that records the current position of the Vic named `sam`. This object is a string of characters. **String** is the name of the Sun standard library class for such objects. The String class contains a method for testing whether two Strings are equal. These new methods are described in Figure 3.2.

aVic.getPosition()
produces a String object recording the current position of the object referred to by aVic.

aString.equals (someOtherString)
tests whether one String is equivalent to another. The String that getPosition() produces equals another String it has produced whenever they represent the same position in the same sequence, even if the actual String objects are different.

Figure 3.2 Two query methods for use with positions in a sequence

Now the `fillSlots` method can be revised so the executor is at the same position in its sequence at the end of execution that it was in at the beginning of execution. Put these two statements at the beginning to make a note of the current position in a variable:

```
String spot;
spot = getPosition();
```

Then put these statements at the end of the `fillSlots` method, after the while-statement that moves the executor down to the end of the sequence:

```
while ( ! spot.equals (getPosition()))
   backUp();
```

This logic checks whether `spot` (the position when `fillSlots` began execution) is the same as the current position (given by the call of `getPosition()`) and if not, backs up by one slot and then repeats the check for equality. When they are equal, the loop stops (a **loop** is the repeated execution of a group of statements; an **iteration** of a loop is one such execution of the group of statements).

The hasSomeFilledSlot method

We will define two new methods so you can move a Vic object down its sequence of slots to the last slot that contains a CD, assuming there is such a slot. You will be able to have logic in a program something like the following:

```
if (sam.hasSomeFilledSlot())
   sam.goToLastCD();
```

The `hasSomeFilledSlot` method is to send a message that asks the executor whether any slot from its current position on down has a CD in it. An initial sketch of a plan to do this is: The executor goes down its sequence until it sees a filled slot (in which case it returns `true`) or it runs out of slots to look in (in which case it returns `false`). But it returns to its original position before returning the answer to the question. You can then refine this initial sketch as shown in the accompanying design block.

DESIGN of hasSomeFilledSlot

1. Make a note of the current position.
2. Move down the sequence until you see a filled slot or you run out of slots.
3. Let `valueToReturn` record whether, at that point, there is still a slot left.
4. Back up until you get to the original position as of the start of this method.
5. Return the value in `valueToReturn` as the answer to the question, "Does the sequence have some filled slot?"

This logic works because (a) if `valueToReturn` is `true`, the first loop stopped before the end, which can only be because the executor stopped at a slot with a CD in it; and (b) if `valueToReturn` is `false`, the executor must not have seen any CD to cause it to stop early. The design is implemented in the upper part of Listing 3.2, which defines a subclass of the Vic class. Figure 3.3 shows a sample execution of this method in detail.

Listing 3.2 The VicPlus class of objects

```
public class VicPlus extends Vic
{
   /** Tell whether any slot here or later has a CD. */

   public boolean hasSomeFilledSlot()
   {  String spot;                              // design step 1
      spot = getPosition();
      while (seesSlot() && ! seesCD())          // design step 2
         moveOn();
      boolean valueToReturn;                    // design step 3
      valueToReturn = seesSlot();
      while ( ! spot.equals (getPosition()))    // design step 4
         backUp();
      return valueToReturn;                     // design step 5
   }  //=====================

   /** Move to the last CD at or after the current position. But
    *  if there is no such CD, stay at the current position. */

   public void goToLastCD()
   {  String spot;                              // design step 1
      spot = getPosition();
      while (seesSlot())                        // design step 2
      {  if (seesCD())
            spot = getPosition();
         moveOn();
      }
      while ( ! spot.equals (getPosition()))    // design step 3
         backUp();
   }  //=====================
}
```

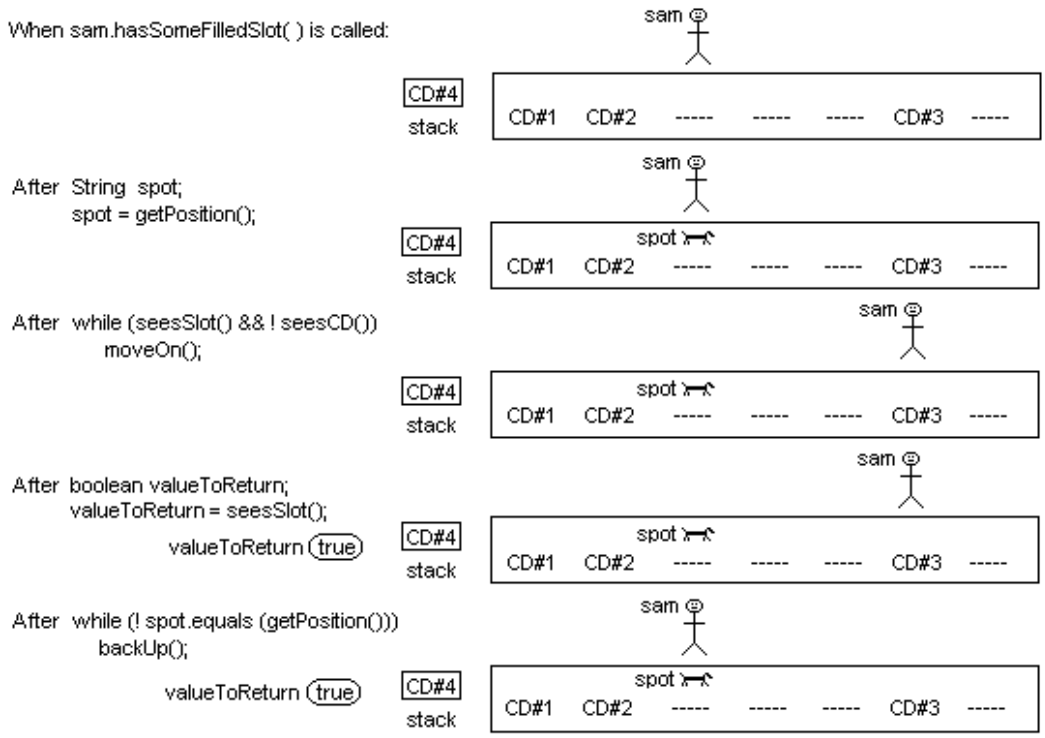


Figure 3.3 Execution of hasSomeFilledSlot

The goToLastCD method

The `goToLastCD` method advances the executor to the last slot that contains a CD. The logic to do so is a bit tricky. How do you go to the last CD? When you see a CD as you go down the sequence, you will not know whether it is the last one. A reasonable plan is to note its position and go further to see if there is another. If you do not see another, go back to the position marked. But if you do see another, forget about the earlier position and mark the later position instead. You need to see a description in ordinary English of how to do this, shown in the accompanying design block. The coding is in the lower part of Listing 3.2.

DESIGN of goToLastCD

1. Make a note of the current position in a variable; call it `spot`.
2. For each slot in the sequence from this position forward, do...
   If the slot you are at contains a CD, then...
   Change `spot` (the note) to indicate this new position instead.
3. Back up to the last position that was stored in `spot`.

Caution Always review the logic of each while-loop you write to be sure it will terminate eventually when executed. Nota Bene: Control-C in the terminal window kills the entire program. This is useful to know when you run a program with a loop that never terminates.

Language elements

A Condition can be:	MethodName (Expression)
or:	VariableName . MethodName (Expression)

Exercise 3.6 How would you revise `goToLastCD` in Listing 3.2 so the executor advances to the last empty slot in its sequence?
Exercise 3.7 How would you revise `hasSomeFilledSlot` in Listing 3.2 so the executor tells whether at least two slots at or after the current slot are filled?
Exercise 3.8 Explain why the assignment to `valueToReturn` in Listing 3.2 must not be replaced by `valueToReturn = seesCD()`.
Exercise 3.9* Write a method `public void fillFirstEmptySlot()` for a subclass of Vic: The executor puts a CD in its first empty slot. Leave the position of the executor unchanged. Precondition: It has an empty slot and a CD is on the stack.
Exercise 3.10* Rewrite the `hasSomeFilledSlot` method in Listing 3.2 to not use any boolean variable.
Exercise 3.11** Write a method `public void fillLastEmptySlot()` for a subclass of Vic: The executor puts a CD in its last empty slot, but only if it has an empty slot and has a CD on the stack. Leave the position of the executor unchanged.

3.3 More On UML Diagrams

Suppose you want to have an application program that moves all the CDs in the first sequence of slots up to the front of the sequence. A reasonable plan is shown in the accompanying design block, assuming the stack is empty (an exercise shows how to do this when the stack is not known to be empty).

DESIGN of MoveToFront
1. Create a Vic for the first sequence of slots; name it `chun`.
2. Record its current position (the first slot) in a variable named `spot`.
3. Send `chun` down the sequence of slots, placing all CDs that it sees onto the stack.
4. Move `chun` back to the beginning of the sequence, to the position stored in `spot`.
5. Send `chun` down the sequence of slots, putting a CD in each slot, until `chun` gets to the end or `chun` runs out of CDs in the stack.

Each step of the design translates quite easily into just a few Java statements. A Java implementation of this design is in Listing 3.3 (see next page). Note that calls of `Vic.say` are inserted where appropriate, even though the design does not mention them. This illustrates the fact that people sometimes add to a design during implementation.

What classes do

The Vic class illustrates three key functions of classes, which you will see time and again:

1. It defines the behaviors an individual Vic object can have (e.g., `takeCD` and `moveOn`).
2. It serves as a factory for objects, since it allows the construction of new Vic objects (by calling on `new Vic()`).
3. It defines class methods that relate to Vic objects as a group, not to one individual Vic object (e.g., `say`, `stackHasCd`, and `reset`).

Listing 3.3 An application program using a Vic object

```
public class MoveToFront
{
   /** Move all the CDs in the first sequence of slots up to the
    *  front of the sequence. Precondition: stack is empty. */

   public static void main (String[ ] args)
   {  Vic chun;                                    // design step 1
      chun = new Vic();
      String spot;                                 // design step 2
      spot = chun.getPosition();

      while (chun.seesSlot())                      // design step 3
      {  chun.takeCD();
         chun.moveOn();
      }
      Vic.say ("All CDs are now on the stack.");

      while ( ! spot.equals (chun.getPosition()))// design step 4
         chun.backUp();

      while (chun.seesSlot() && Vic.stackHasCD())// design step 5
      {  chun.putCD();
         chun.moveOn();
      }
      Vic.say ("The first few slots are now filled.");
   }  //======================
}
```

Going further with UML diagrams

Figure 3.4 is the class diagram for this MoveToFront program. It illustrates the two remaining notations for creating UML class diagrams this book uses:

1. You may give the types of parameters in parentheses after the method name if you choose, e.g., the `say` and `equals` methods in Figure 3.4.
2. You may give the type of value a method returns if you choose (standard UML notation puts it after the parentheses, in contrast to Java method headings), e.g., the `getPosition` and `seesSlot` methods in Figure 3.4.

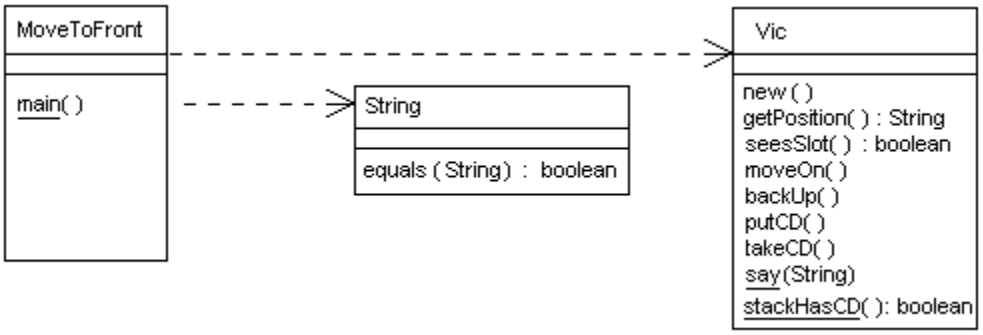


Figure 3.4 UML class diagram for MoveToFront

Exercise 3.12 The MoveToFront program has the CDs that were initially nearer the front of the sequence end up nearer the end of the sequence. Revise the program so that CDs that were initially frontwards remain so. Hint: Take CDs while backing up.

Exercise 3.13 Write a query method `public boolean lastIsFilled()` for a subclass of Vic: The executor tells whether its last slot is filled. Precondition: `seesSlot()` is true.
Exercise 3.14* Write a Finder subclass of Vic with two methods: `goToFirstEmpty` and `goToFirstFilled`, where the executor advances until it comes to the first empty slot or the first filled slot, respectively, or to where `seesSlot()` is false if necessary.
Exercise 3.15** Add a method `public void byOnes()` to the Finder class of the preceding exercise: The executor moves one CD at a time to the earliest empty slot that comes before that CD in the sequence. This gets all the CDs to the front of the sequence without having more than one extra CD on the stack at a time. This `byOnes` method should call on the other two methods in Finder as needed.

3.4 Using Private Methods And The Default Executor

The methods that go through an entire sequence, such as `fillSlots` and `toggleCDs` in Section 3.1, seem to call for a new subclass called Looper (since the methods will usually involve loops). We develop several such methods in this section and the next.

Initializing a variable when it is declared

Java allows you to combine the declaration of a variable with its initial assignment of a value. The following three statements illustrate how this language feature can be used. In each case, the one statement given replaces two statements in the specified listing:

```
boolean valueToReturn = seesSlot();   // in Listing 3.2
Vic chun = new Vic();                 // in Listing 3.3
String spot = chun.getPosition();     // in Listing 3.3
```

Caution A common error is to try to use the value of a variable before you put a value in it. For instance, you cannot say `sam.takeCD()` unless you previously assigned a value to sam, as in `sam = new Vic()`. You can avoid this error if you always assign a value to a variable in the same statement where you declare it or in the very next statement.

Listing 3.4 (see next page) illustrates the use of this new language feature in the Looper subclass of the Vic class. This class contains the `fillSlots` method discussed in Section 3.1 plus a `clearSlotsToStack` method with a very similar logic, since it does the exact opposite of `fillSlots`: It has the executor move all the CDs in its slots to the stack. Study both to make sure you thoroughly understand them.

Private methods

The third method in Listing 3.4, `backUpTo`, is for convenience; it saves writing out the loop that backs up, as seen in the two previous listings. You can call `backUpTo(x)` for any position value `x`; the value in `x` is assigned to `someSpot` inside the `backUpTo` method. So the loop in the `backUpTo` method executes until `getPosition()` returns a value that indicates the same spot `x` indicates.

The `backUpTo` method will be used only by other Looper methods (more Looper methods are in the next section). The restriction to Looper methods only is produced by having the `private` modifier in place of `public`. A **private** method cannot be mentioned outside of the class it is defined in. A method defined as `public` can be mentioned in any other class, as long as you supply the executor or other indication of the class it is in. These two modifiers indicate the "visibility" of the method.

Listing 3.4 The Looper class of objects, some methods postponed

```
   /** Process a sequence of slots down to the end, usually
    *  without changing the current position of the executor. */

public class Looper extends Vic
{
   /** Fill in the current slot and all further slots
    *  from the stack until the end is reached. */

   public void fillSlots()
   {  String spot = getPosition();
      while (seesSlot())
      {  putCD();
         moveOn();
      }
      backUpTo (spot);
   }  //=====================

   /** Move all CDs in the slots into the stack. */

   public void clearSlotsToStack()
   {  String spot = getPosition();
      while (seesSlot())
      {  takeCD();
         moveOn();
      }
      backUpTo (spot);
   }  //=====================

   /** Back up to the specified position.  Precondition:
    *  someSpot records a slot at or before the current slot. */

   private void backUpTo (String someSpot)
   {  while ( ! someSpot.equals (getPosition()))
         backUp();
   }  //=====================
}
```

You cannot call any of the methods in Listing 3.4 without an executor, i.e., an instance of the class before the dot. So all of these methods are instance methods of the Looper class.

Using the Looper methods

The new Looper methods in Listing 3.4 let you write the entire body of the earlier Listing 3.3 more simply as follows:

```
   public static void main (String[ ] args)
   {  Looper chun = new Looper();
      chun.clearSlotsToStack();
      Vic.say ("All CDs are now on the stack.");
      chun.fillSlots();
      Vic.say ("The first few slots are now filled.");
   }  //=====================
```

Garbage collection

Inside each of the two public method definitions in Listing 3.4, the String object that `getPosition()` returns is assigned to a String variable declared inside the method. Such a variable is temporary, transient. The variable is created when the method is called and it is discarded when the method is exited, by coming to the end of the commands in the method.

When the method says `spot = getPosition()`, the newly-created object has only `spot` to refer to it. When the method is exited, no variable at all refers to the String object. Whenever that happens, the runtime system automatically disposes of the object so it does not clutter up RAM. This is **garbage collection**. Programs written in languages without garbage collection can leave RAM littered with unusable space after they terminate execution; this is called memory leakage.

A metaphor may help to explain garbage collection: An object is a boat. An object variable is a metal ring on a river dock to which you can tie a boat. A boat can be tied to more than one ring at the same time. A statement such as `sam = sue` has the boat that is tied up to `sue` also tie up to `sam`. Whatever boat may have been already tied up to `sam` is cast off from `sam`, since only one boat can tie up to a ring at a time (the ring is not big enough for two ropes). If any boat becomes untied from all rings, it floats down the river, goes over a waterfall, and smashes into kindling at the bottom. The garbage is then collected by Java's automatic garbage collectors and recycled to make new boats.

The default executor: this

Within the definition of an instance method, you cannot refer to the executor by name, since it varies depending on the method call. One time you might have the statement `sam.fillSlots()`, and another time you might have `sue.fillSlots()`. So inside the definition of `fillSlots`, you cannot mention either `sam` or `sue`, because it could be either of them or some other Looper variable altogether.

Java provides a pronoun for the executor: `this` always refers to the executor when used in a statement inside an instance method. For instance, the body of the `fillSlots` method of Listing 3.4 could be rewritten as follows with the same effect:

```
public void fillSlots()  // illustrating use of this
{  String spot = this.getPosition();
   while (this.seesSlot())
   {  this.putCD();
      this.moveOn();
   }
   this.backUpTo (spot);
}  //=======================
```

If your main logic executes the statement `sam.fillSlots()`, then for that execution of the logic of `fillSlots`, `this` refers to `sam`. If your main logic later executes the statement `sue.fillSlots()`, then for that second execution of the logic of `fillSlots`, `this` refers to `sue`. The rule the compiler applies is: If you do not explicitly state the executor where an executor is required, it supplies the **default executor** `this`.

More Looper methods

Listing 3.5 contains the definition of two more Looper methods, with three methods left as exercises. The `this` pronoun appears in Listing 3.5 wherever an instance method is called, to help you remember what it means. But in the future, this book only uses the optional `this` when other objects are mentioned in the method; in such a case, `this` helps you keep straight which object you are talking about.

The `fillOddSlots` method in the upper part of Listing 3.5 fills in every other slot starting with the first one. It does not seem to need a detailed design because you can just make a small change in the logic of `fillSlots` as follows: After that logic moves on by one slot, insert an if-statement to check that there really is a slot there and, if so, move on by one extra slot. Figure 3.5 gives an example of what happens when `fillOddSlots` is called.

Listing 3.5 More methods of the Looper class

```
// public class Looper extends Vic, continued

   /** Fill in every other slot from the stack, beginning
    *  with the current slot, until the end. */

   public void fillOddSlots()
   {  String spot = this.getPosition();
      while (this.seesSlot() && stackHasCD())
      {  this.putCD();
         this.moveOn();
         if (this.seesSlot())
            this.moveOn();
      }
      this.backUpTo (spot);
   }  //=====================

   /** Tell whether every slot here and later has a CD. */

   public boolean seesAllFilled()
   {  String spot = this.getPosition();          // design step 1
      while (this.seesSlot() && this.seesCD())   // design step 2
         moveOn();
      boolean valueToReturn = ! this.seesSlot(); // design step 3
      this.backUpTo (spot);                      // design step 4
      return valueToReturn;                      // design step 5
   }  //=====================

   // the following three are left as exercises
   public void fillEvenSlots()      {  }
   public boolean seesOddsFilled()  {  }
   public boolean seesEvensFilled() {  }
```

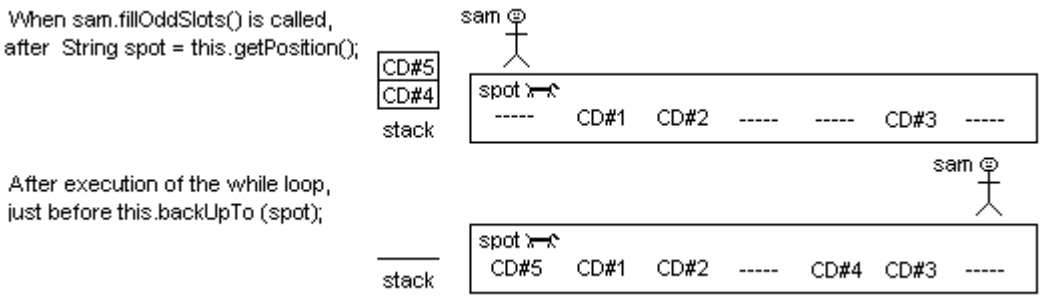


Figure 3.5 Stages of execution for fillOddSlots

Development of seesAllFilled

The `seesAllFilled` method has the executor tell whether all remaining slots, starting from the current position, contain CDs. A reasonable plan is in the accompanying design block. The coding for `seesAllFilled` is in the lower part of Listing 3.5.

DESIGN of seesAllFilled

1. Mark the current position so you can return to it when you have the answer to the question.
2. Go down the sequence until you get to the end or else you see an empty slot.
3. Determine the value to be returned, which is `false` if you are now at an empty slot and is `true` otherwise.
4. Go back to the position you had at the beginning of execution of the process.
5. Return the value found at step 3 of this logic.

If `seesAllFilled` is called when the position of the Vic is already at the end of its sequence of slots, we still say it is true that all slots are filled, in the **vacuous** sense that there is no unfilled slot as a counterexample. This is an application of the computer science meaning of an assertion of the form All-A-are-B; it may not coincide with the vernacular meaning.

Caution A common error people make in Java is to put a semicolon right after the parentheses around a condition, as in `if(whatever);` or `while(whatever);`. That semicolon marks the end of the `if` or `while` statement, so the compiler does not consider the statement on the next line to be subordinate. A bare semicolon directly after the parentheses around the `while` or `if` condition counts as a subordinate statement that does nothing. That is rarely what you want.

Language elements

A Statement can be: Type VariableName = Expression ;
You may use "private" in place of "public" in a method heading.
You may use "this" within an instance method to explicitly indicate the executor of the method.

Exercise 3.16 Explain why the following method heading causes a compilation error:
`public static void Main (string[] args)`.

Exercise 3.17 Explain why the following causes a compilation error:

```
Looper Bob = new Looper();
Vic spot = Bob.getPosition();
while ( ! spot.equals (Bob.getPosition())
   Bob.moveOn();
```

Exercise 3.18 Rewrite the `hasSomeFilledSlot` method in Listing 3.2 to use `this` wherever it is allowed.

Exercise 3.19 Write the `public void fillEvenSlots()` method described in the Looper class. The first slot filled should be the slot after the current position (if it exists). Call on `fillOddSlots` to do most of the work.

Exercise 3.20 Write the `seesOddsFilled` method described in the Looper class.

Exercise 3.21 Write the `seesEvensFilled` method described in the Looper class. Call on `seesOddsFilled` to do most of the work.

Exercise 3.22* Write an application program that tests out your solutions to the two preceding exercises by calling each one for the first sequence and printing a message saying what each returned.

Exercise 3.23* Write a Looper method `public void bringBack()`: The executor removes the CD in its current slot, if any, then brings each CD that is later in the sequence back one slot. Leave the position of the executor unchanged.

Exercise 3.24* Compare the coding of `seesAllFilled` in Listing 3.5 with the coding of `hasSomeFilledSlot` in Listing 3.2. Note that the only material difference is the presence or absence of the not-operator in two places. If both places had the not-operator, what would be a good name and comment heading for the resulting method? What would they be if neither place had the not-operator?
Exercise 3.25** Write a Looper method `public void overOrOut()`: The executor moves each CD in its sequence of slots either (a) to the following slot if there is a following slot which is empty, or (b) to the stack if not. Leave the position of the executor unchanged.

3.5 A First Look At Declaring Method Parameters

When one of the public methods in the earlier Listing 3.4 executes the statement `backUpTo(spot)`, it assigns the value in `spot` to the `someSpot` variable in the backUpTo method. That is, it executes `someSpot = spot`, so that `someSpot` now refers to the same position `spot` refers to. Then any test of `someSpot`'s object inside the method is by definition a test of the object `spot` refers to.

A value inside the parentheses of a method call is an **actual parameter** or **argument** of the call. For instance, the actual parameter of `backUpTo(spot)` is `spot`, and the actual parameter of `Vic.say("whatever")` is `"whatever"`.

A variable inside the parentheses of a method heading is a **formal parameter** of the method. For instance, `someSpot` is the formal parameter of the method called by `backUpTo(spot)`, as defined in Listing 3.4

The hasAsManySlotsAs method

The method call `sue.hasAsManySlotsAs(ruth)` is to tell whether the sequence represented by `sue` has exactly the same number of slots as the sequence represented by `ruth` has. The logic can be designed as shown in the accompanying design block.

DESIGN of hasAsManySlotsAs, with a parameter
1. Make a note of the current position of the executor; store it in `thisSpot`.
2. Repeat the following until either the executor or the parameter has no more slots...
   2a. Move both the executor and the parameter forward one slot.
3. Make a note that the value to be returned by this method is `true` only if both of the two Vics now have no more slots.
4. Back up both of them one at a time until the executor gets back to `thisSpot`.
5. Return the value noted in Step 3.

Listing 3.6 (see next page) contains an implementation of this design. Suggestion: When you need to write a method whose logic is not immediately obvious, first make a design similar to the ones you have seen so far in this chapter.

Correspondence of formal and actual parameters

Suppose the statement `sue.hasAsManySlotsAs(ruth)` is in a main method. Then it causes an execution of the boolean method that assigns the value in `ruth` to `par` and the value in `sue` to `this`. So whatever `this` does is actually being done by `sue` and whatever `par` does is actually being done by `ruth`. `par` is the formal parameter that corresponds to the actual parameter `ruth`. Note: `par` is short for parameter; this book uses this name when the context suggests no better name for a parameter.

Listing 3.6 An instance method to compare the lengths of two sequences

```
public class TwoVicUser extends Vic
{
   /** Tell whether the executor has exactly the same number of
    *  slots as the Vic parameter.  */

   public boolean hasAsManySlotsAs (Vic par)
   {  String thisSpot = this.getPosition();      // design step 1
      while (this.seesSlot() && par.seesSlot())  // design step 2
      {  this.moveOn();
         par.moveOn();
      }

      boolean valueToReturn = ! this.seesSlot()  // design step 3
                           && ! par.seesSlot();

      while ( ! thisSpot.equals (this.getPosition()))// d. step 4
      {  this.backUp();
         par.backUp();
      }
      return valueToReturn;                      // design step 5
   }  //======================
}
```

If the main method also contains `bill.hasAsManySlotsAs(ted)`, it causes an execution of the boolean method that gives `par` the value in `ted` and `this` the value in `bill`. Note: `bill` must be a TwoVicUser object, because it is the executor of a method defined in the TwoVicUser class. But `ted` can be any Vic object, such as a Vic or a TwoVicUser object. That is, you may assign to a Vic variable (such as `par`) any object of any subclass of Vic.

Review of the email metaphor

In the email metaphor of Section 1.6, a method call is an email message you send to an object. The method name is the subject line of the email. The parameter is the body text of the email. A method call with an empty pair of parentheses indicates there is no body text in the message. But when the email message is `bill.hasAsManySlotsAs(ted)`, the recipient `bill` sees from the subject line that you want to know whether it has as many slots as some other object. Then `bill` looks at the body text of the email to find out who the other object is.

The giveEverythingTo method

The action method in Listing 3.7 removes all CDs from the executor's slots and puts them into the Looper parameter's slots, along with any CDs that happen to already be on the stack. A sample call is `sam.giveEverythingTo(sue)`. The logic is quite straightforward because both the executor and the parameter are Loopers: Tell the executor to `clearSlotsToStack`, then tell the Looper parameter to `fillSlots`. Since the Giver class is a subclass of Looper which is a subclass of Vic, a Giver object inherits all the capabilities of Loopers as well as Vics. Figure 3.6 shows the **hierarchy** of object classes involving the Giver class.

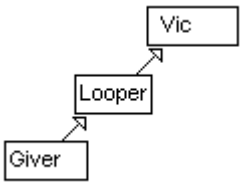


Figure 3.6 Giver's hierarchy

Listing 3.7 The Giver class of objects

```
public class Giver extends Looper
{
   /** The executor gives all of its CDs to the Looper parameter,
    *  which distributes them to its own slots to the extent
    *  possible, along with any CDs originally on the stack. */

   public void giveEverythingTo (Looper target)
   {  this.clearSlotsToStack();
      target.fillSlots();
   }  //====================
}
```

Programming Style The `giveEverythingTo` method uses the fact that the two Vics are in fact Looper objects, by calling the `clearSlotsToStack` method and the `fillSlots` method defined in the earlier Listing 3.4. The executor must of course be an instance of Giver, because the `giveEverythingTo` method is defined in the Giver class. However, the parameter only needs to be able to execute `fillSlots`, so it is enough that it be an instance of Looper. It is good style to not specify it to be an instance of Giver. That retains the greatest flexibility in the use of the `giveEverythingTo` method.

Local variables versus parameters

You cannot call the `giveEverythingTo` method unless you have a Giver object to do the giving and a Looper object to be given to. The value that is listed before the dot in that method call is assigned to `this` inside the `giveEverythingTo` method. The value that is listed inside the parentheses of that method call is assigned to `target` inside the `giveEverythingTo` method.

Say a main method declares four different Giver objects and contains the following two statements using them:

```
steve.giveEverythingTo (don);
mike.giveEverythingTo (dru);
```

The main method has `steve`, `don`, `mike`, and `dru` as its **local variables** (variables declared within the body of the method). In general, the only way you can refer to the value of a local variable of one method within another method is to have the local variable be an executor or an actual parameter of the method call. Either way, that other method cannot change which value is stored in the local variable; it can only change the state of the object to which the value refers.

On the first call of the `giveEverythingTo` method, `steve` is the executor, so `this` is an **alias** for `steve` during execution of the first call. On the second call of the `giveEverythingTo` method, `mike` is the executor, so `this` is an alias for `mike` during execution of the second call.

Likewise, `target` is the formal parameter, so `target` is an alias for the actual parameter `don` on the first call but it is an alias for the actual parameter `dru` on the second call. In effect, the first method call performs the assignment `target = don` and the second method call performs the assignment `target = dru`.

A formal parameter (variable declared inside the parentheses of a method heading) differs from a local variable (declared in the body of the method) in that (a) a formal

parameter receives its initial value at the time the method is called, but (b) a local variable has no initial value until the statements explicitly give it one. Both kinds of variables are local to the method definition in the sense that they cannot be used outside the method definition.

Multiple parameters

A method can have two or more parameters, separated by commas. For instance, the following method has two Vic parameters. The executor tells whether at least two of the three sequences (its own, `one`'s, and `two`'s) have at least one slot available:

```
public boolean atLeastTwoNotAtEnd (Vic one, Vic two)
{  if (this.seesSlot() && one.seesSlot())
      return true;
   if (this.seesSlot() && two.seesSlot())
      return true;
   return one.seesSlot() && two.seesSlot();
}  //======================
```

Two examples of how this method might be called are:

```
if (jazz.atLeastTwoNotAtEnd (pop, classical))...
boolean result = first.atLeastTwoNotAtEnd (third, second);
```

Language elements
You may put the following within the parentheses of a method heading: Type VariableName
If you have two or more such phrases within the parentheses, separate those phrases with commas.

Exercise 3.26 If you changed the first statement of `hasAsManySlotsAs` to be `String thisSpot = par.getPosition()`, what other changes would you have to make so it gives the right answer?
Exercise 3.27 How would you change the `hasAsManySlotsAs` method so the executor tells whether it has more slots than the Vic parameter?
Exercise 3.28 Write a query method `public boolean isAtOneGivenPosition (String one, String two)` for a subclass of Vic: The executor tells whether either of those two parameters is the same as its current position.
Exercise 3.29 Write a method `public void moveToCorrespondingSlot (Vic par)` for a subclass of Vic: Every CD in a slot of the parameter that corresponds to an empty slot in the executor is moved over to the executor's corresponding slot. Leave the position of the two Vics unchanged.
Exercise 3.30* Write a query method `public boolean hasMoreThanDouble (Vic par)` for Looper: The executor tells whether it has more than twice as many slots as the Vic parameter. Do not use numeric variables. Precondition: The executor is known to have an even number of slots. Extra credit: Remove the precondition.
Exercise 3.31* Draw the UML diagram for Listing 3.6.
Exercise 3.32** Write a query method `public boolean matches (Vic par)` for Looper: The executor tells whether it has a CD wherever the Vic parameter has a CD and it does not have a CD wherever the Vic parameter does not. That is, the two sequences of slots are the same in terms of the presence or absence of CDs, starting from the current slot in each.
Exercise 3.33** Write a method `public void shiftOne (Vic one, Vic two)` for a subclass of Vic: At each position where the executor has an empty slot and either of the two Vic parameters has a filled slot in the corresponding position, shift the CD to the executor's slot. When a choice is possible, take a CD from the first parameter's slot.
Exercise 3.34** Write a query method `public boolean sameNumber (Vic par)` for Looper: The executor tells whether it has the same number of CDs in its slots as the Vic parameter. Do not use numeric variables. Hint: Advance each to the next non-empty slot. Repeat this until one runs out of slots. Does the other?

3.6 Returning Object Values

You have written several methods that return boolean values. It is also legal to have a method return an object value, such as a Vic or Looper or String value. For instance, `sam.getPosition()` returns a String object.

The `lastEmptySlot` method in Listing 3.8 illustrates the return of a String object. Its purpose is to return the position of the last empty slot in a sequence of slots; but it returns the current position, empty or not, if there is no empty slot after the current position. A main method could use this `lastEmptySlot` method in a statement such as

```
String spot = sam.lastEmptySlot();
```

or in a condition such as

```
sue.lastEmptySlot().equals (sue.getPosition())
```

which tells whether `sue` is positioned at its last empty slot. This method must be in the Looper class because it calls the private method `backUpTo`, which is not even accessible from a subclass of Looper.

Listing 3.8 A Looper method returning a String

```
/** Return the position for the last empty spot in
 *  the sequence, or the current spot if no empty spot. */

public String lastEmptySlot()
{  String spot = this.getPosition();
   String lastEmpty = spot;  // in case no later slot is empty
   while (this.seesSlot())
   {  if ( ! this.seesCD())
         lastEmpty = this.getPosition();
      this.moveOn();
   }
   this.backUpTo (spot);
   return lastEmpty;
}  //=====================
```

The logic in this `lastEmptySlot` method goes through each slot in the sequence, setting `lastEmpty` to the position of each empty slot it sees. `lastEmpty` could be given several different values, but each assignment replaces whatever was already stored in the variable. So only the last value assigned is in `lastEmpty` when the loop terminates. At that time, `lastEmpty` must contain the position of the last empty slot.

No design block is given for this method because it is so similar to the `goToLastCD` method in the earlier Listing 3.2. You will find it informative to compare and contrast the two step by step.

Returning a Vic object

You can return a Vic object as well as a String object, as illustrated by the following method. It repeatedly creates new Vic objects until it finds one whose first slot contains a CD (or until it runs out of Vics).

```
public Vic firstWithCD()
{  Vic sequence = new Vic();
   while (sequence.hasSlot() && ! sequence.seesCD())
      sequence = new Vic();
   return sequence;
}  //======================
```

Caution You can avoid the most common compiler errors that beginners make if you just check two things before compiling a program: First, every left brace has a matching right brace directly below it, and vice versa. Second, no line followed by an indented line ends in a semicolon, and every line not followed by an indented line does end in a semicolon.

Reminder: Variable names should start with lowercase letters and class names should start with capitals. You may ask why it should be so. You might as well ask why you cannot say "el mano" in Spanish instead of "la mano." You would be understood alright, but it is not proper Spanish.

Exercise 3.35 What change would you have to make in the `lastEmptySlot` method of Listing 3.8 to return the position of the last non-empty slot?
Exercise 3.36 Write a Looper method `public Vic shorterOne (Vic par)`: Return the Vic with the fewer slots, either the executor or the parameter. Leave both unchanged. Precondition: They do not have an equal number of slots.
Exercise 3.37* Revise the `lastEmptySlot` method to return the position of the next-to-last empty slot. Return the initial position if the executor has less than two empty slots.
Exercise 3.38** Rewrite the `lastEmptySlot` method to have the executor go directly to the end of the sequence and then find the last empty slot as it comes back towards the starting position.

3.7 More On The Analysis And Design Paradigm

Problem Statement Write a program to work with three sequences of slots for storing CDs. The first Vic has perhaps some country music CDs in its slots, and the second Vic has perhaps some jazz CDs in its slots. You are to put all of these CDs in the third Vic's slots, alternating the two kinds of music (for variety). The third Vic may already have some CDs in its slots; you are to leave these where they are and fill in the rest of the slots, to the extent possible.

Many people, given a problem assignment such as this, are not sure where to start. It is extremely useful to break up the process into five basic stages:

Analysis (clarifying what to do) Make sure you clearly understand what the program is to accomplish. Consider exceptional cases and how you are to handle them. Write down data you will use to test the final software and figure out what will happen when that data is used. Go to the client (or your instructor if appropriate) for a decision on ambiguous points. You need a clear, complete, unambiguous specification before you can go further.

Logic Design (deciding how to do it) Make a step-by-step plan of how you will get the job done. The design method described later in this section is a reliable and efficient way to do this. You have already seen many design blocks illustrating the method.

Object Design (choosing the objects that help you do it) See what kinds of objects you have already available that can provide the services you need. Perhaps you have to add more capabilities to existing objects (e.g., additional Looper methods). Perhaps you need to invent completely new kinds of objects to do the job.

Refinement (making sure you are doing it) Go over your logic at length to make sure it satisfies the stated requirements, that it will do what it should for the given test data, and that it will behave correctly in exceptional cases.

Implementation (doing it) Translate your logic design into Java using the methods supplied by the objects you have designed.

When you implement the design, you will usually find that several steps are too complex to do easily. In that case you call a new method for which you repeat the entire process on a lower level: (a) analyze the specification for the sub-problem to make sure it is clear; (b) design a step-by-step solution of the sub-problem; (c) select or invent the object methods you need; (d) refine the plan; (e) implement the plan in Java.

To create a good plan, write out or say aloud the steps the computer will take, in ordinary English sentences. Then organize this list of steps to show which steps are done conditionally or repeatedly and to highlight the condition for doing them or repeating them. Otherwise keep it in English (or whatever natural language you speak most fluently). You can sharpen your plan by planning the data with which you will test your program when it is done and computing what the results will be for the various test runs.

Analysis for the Interleaf program

When you think further about the problem statement for the alternating CDs, you realize it is not quite clear whether the first CD moved is to come from the first sequence or the second sequence. Also, if the first or second sequence of slots has more CDs than are required to fill in the slots in the third sequence, are those extra CDs supposed to stay where they are, or should they go onto the stack?

You talk to the client to get the answers to these questions. For the rest of this discussion, assume that the client says all CDs from the first sequence are to go into the odd-numbered slots (1, 3, 5, etc.) of the third sequence, with any leftovers to be put on the stack. The client tells you there will be enough CDs in the first sequence to do this. However, if all of the odd-numbered slots of the third sequence are already filled, you are to leave the CDs in the first sequence. You are to handle the second sequence analogously, with CDs going into the even-numbered slots. The client is quite clear that the CDs put into the third sequence are to be in the same order as they were in the sequence they came from.

Logic design for the Interleaf program

A reasonable logic design of the problem is shown in the accompanying design block. This design illustrates **Structured Natural Language Design**, **SNL design** for short. Steps 3b and 4b specify that the target's slots are to be filled in reverse order from the stack, so that whatever was furthest down the source sequence, and thus ended up on top of the stack after transferring the CDs to the stack, goes furthest down the target sequence.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic

1. Create two Vic objects to serve as the source of the CDs.
2. Create a third Vic object to receive the CDs. Refer to it as `target`.
3. If `target` does not have all of its odd-numbered slots filled, then...
    3a. Transfer every CD the first Vic has in its slots to the stack.
    3b. Fill `target`'s odd-numbered slots from the stack in reverse order.
4. If `target` does not have all of its even-numbered slots filled, then...
    4a. Transfer every CD the second Vic has in its slots to the stack.
    4b. Fill `target`'s even-numbered slots from the stack in reverse order.

You saw several examples of structured design earlier in this chapter and in Chapter Two. The three ways that SNL design differs from ordinary discourse are as follows:

1. When action X is executed conditionally, express it in the form `if whatever then...X` with the action X on a separate line and indented beyond the description of the condition.
2. When action X is executed repetitively, express it in something like the form `For each value do...X` with action X on a separate line and indented beyond the description of the looping. The exact phrasing is unimportant; `Repeat until whatever...X` often makes more sense in a particular situation.
3. When you must refer to a particular value several times, give it a name (`target` in the preceding example). This is clearer than using a phrase such as "the third Vic that was created" many times.

This design is an **algorithm**, which means a step-by-step description of a process for accomplishing a task, specific enough that at each step there is no question what to do next. You do not have to put the secondary line numbers in your design if you do not want to. The crucial part is to show which actions depend on which conditions.

Everything about this design is ordinary English except for variable names and indenting to show which actions are done under which conditions. That is the "structured" part of the design. Do not write in Java until you know what you are going to say.

Do not try to break the design down into very many small steps; ten steps is usually more than enough. But include all significant steps. Your steps can specify quite complex actions, such as Steps 3a and 3b in the preceding design.

Object design for the Interleaf program

Now you decide what kinds of objects will help you get the job done quickly and easily. Checking whether odd-numbered or even-numbered slots are filled, and clearing out all the CDs from a sequence, are skills possessed by Looper objects (Listing 3.4 and Listing 3.5). So you choose them to help you, rather than the poorly-educated Vic objects.

Each step of the logic design turns out to be easy to implement in Java (using e.g. a Looper's `clearSlotsToStack` for Step 3a) except for Steps 3b and 4b. An object that can do Step 3b can do Step 4b with a small adjustment. So you really need an even smarter kind of Looper, one that can carry out the process for Step 3b. You could add a method to the Looper class for this task. But you probably will never need it again, and the Looper class is becoming rather cluttered. So you could make a subclass of Looper that has this capability, intended for use in this program only.

You need to develop an SNL design for this second-level process. It could be as shown in the accompanying design block. You then refine your overall design by studying it to make sure you understand every aspect of it and by studying the original specifications to make sure you met them all.

DESIGN of the sub-algorithm: filling odd-numbered slots in reverse order

1. Make a note of the current position in the sequence.
2. Move two slots at a time down the sequence until you reach the end.
3. If you moved an odd number of times to get to the end then...
    3a. Back up to the last slot and put a CD there.
4. Repeat the following for every other slot until you are where you started...
    4a. Back up two slots.
    4b. Put a CD in the current slot.

Implementation of the Interleaf program

The implementation stage translates each sentence of the design into a few statements of the programming language. You often make some minor additions while coding. For instance, you could start the program with Vic's `reset` command, which lets you run several test cases easily. And you could display a message when the program finishes. Listing 3.9 is a possible final solution for the coding. Figure 3.7 is the UML diagram.

Listing 3.9 Application program using the BackLooper class of objects

```
public class Interleaf
{
   /** Move the first sequence's CDs to the odd-numbered slots
    *  of the third sequence.  Move the second sequence's CDs to
    *  its even-numbered slots.  No effect if not 3 sequences. */

   public static void main (String[ ] args)
   {  Vic.reset (args);
      Looper one = new Looper();                // design step 1
      Looper two = new Looper();
      BackLooper target = new BackLooper();     // design step 2
      if ( ! target.seesOddsFilled())           // design step 3
      {  one.clearSlotsToStack();               // design step 3a
         target.fillInReverse();                // design step 3b
      }
      if ( ! target.seesEvensFilled())          // design step 4
      {  two.clearSlotsToStack();               // design step 4a
         target.moveOn();                       // design step 4b
         target.fillInReverse();
      }
      Vic.say ("All done putting CDs in #3");
   }  //====================
}
//#################################################################


public class BackLooper extends Looper
{
   /** Fill slots 0,2,4,6,... ahead of the current one, reverse
    *  order. Precondition: The executor has at least 1 slot. */

   public void fillInReverse()
   {  String spot = getPosition();           // sub-design step 1
      boolean movedInPairs = true;           // sub-design step 2
      while (seesSlot())
      {  movedInPairs = ! movedInPairs;
         moveOn();
      }
      if ( ! movedInPairs)                   // sub-design step 3
      {  backUp();
         putCD();
      }
      while ( ! spot.equals (getPosition())) // sub-design step 4
      {  backUp();
         backUp();
         putCD();
      }
   }  //====================
}
```

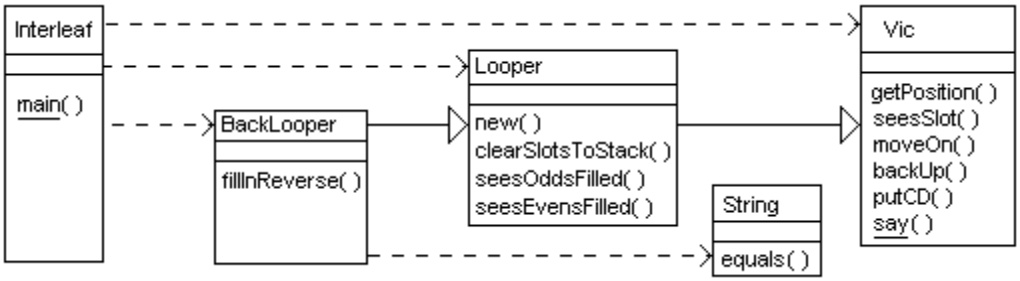


Figure 3.7 UML class diagram for the Interleaf class

The statement `movedInPairs = ! movedInPairs` illustrates a technique you have not seen before. The statement switches the value of the boolean variable between being `true` and being `false` each time through the loop. So it will be `true` at the test of `seesSlot()` if and only if the loop has executed an even number of times. If it is `false` when the loop terminates, the executor must back up one slot to be an even number of slots away from the slot where it started.

Technical Note Java will let you put the BackLooper class in the same file with the application program class if you remove the word `public` from the class heading for BackLooper. The compiler will "complain" if you try to use a non-public class in another class in some other file. But since you do not expect to use the BackLooper class for any other situation, keeping it in the same file with Interleaf is a reasonable thing to do.

Other aspects of software development

In larger projects, you should normally set an intermediate goal of developing software that does much of what the final project should do. After you test it thoroughly, you add to it to come closer to the final version. Repeat this until done. This is called **iterative development**. You will see examples of it later in this book.

Most people cannot go directly from the statement of a complex problem to the expression of the algorithm in Java with few errors. It is far easier, and takes much less time overall, to go through the intermediate stages just described.

If you recite your solution aloud in ordinary English sentences, you will more easily hear any bugs it might have. That will make less work for you in getting the final Java solution right.

Programming Style It is good style to rarely comment individual statements in your programs; commenting each method as a whole is usually enough. This book comments some individual statements in Chapters One through Three only to help you understand what newly-introduced commands and concepts mean and to see how steps of the design are implemented in the coding.

Exercise 3.39 Rewrite the first loop in the `fillInReverse` method so that it advances two slots each time through the loop, except if it can only advance one slot it sets `movedInPairs` to `false`.
Exercise 3.40* Write out a design in SNL for the program of the following exercise.
Exercise 3.41** Write an application program that moves a CD from each slot in the first sequence to the corresponding slot in the second sequence where possible, and also from each slot in the second to the corresponding slot in the first where possible.

3.8 Analysis And Design Example: Finding Adjacent Values

Suppose you need a special kind of Looper object that can answer the question, "Do you have two CDs right next to each other?" You only want it to consider CDs at or after its position at the time you ask the question. So you need a subclass of Looper with a query method that tells whether the executor contains two CDs right next to each other, looking only at slots at or after the current position. Two examples of how such a method might be used are as follows:

```
if (sue.hasTwoTogether())...
while (sam.hasTwoTogether())...
```

You could use the accompanying design block for this `hasTwoTogether` method.

DESIGN of hasTwoTogether

1. Make a note of the current position so you can return to it when you have the answer to the question.
2. Go down the sequence and find out whether you have two CDs together.
3. Go back to the position you had at the beginning of execution of the process.
4. Return the value found at step 2 of this logic.

This plan does not have enough detail. Steps 1, 3 and 4 can be implemented with just one or two Java statements, but step 2 is quite complex. You need a sub-plan to break step 2 down into enough detail that you can easily implement it in Java.

One tactic for solving this sub-problem is to make a note at each slot of whether a CD is in the slot. When you come to the next slot, you know to return true if it has a CD and your note says the previous slot has a CD. Otherwise you update the note for the current slot and go further.

This logic is hard to follow written in normal paragraph form. You need to lay it all out in a structured design so you can study it. The accompanying design block works well.

DESIGN of the sub-algorithm foundPair

1. Create a boolean variable `previousSlotIsEmpty` that can be tested at any slot to tell whether the previous slot is empty. Since you will first test it when at the second slot, initialize it to `true` if the first slot is empty, to `false` if not.
2. Move forward to the second slot.
3. For each slot in the sequence, from this second slot forward, do...
    - 3a. If you do not see a CD then...
        - Make a note that `previousSlotIsEmpty` is `true`, to be tested later.
    - 3b. But if you do see a CD and the previous slot was empty then...
        - Make a note that `previousSlotIsEmpty` is `false`, to be tested later.
    - 3c. Otherwise you see a CD and the previous slot was not empty, so...
        - Return the answer `true` without going any further in this logic.
    - 3d. Move forward to the next slot.
4. Return `false`, since you reached the end of the sequence without seeing two together.

Always review your design for logical consistency and completeness before you implement it. A review of this plan finds a defect: The program will fail if you try to move forward to the next slot (step 2 of the sub-algorithm) when you are already at the end of the sequence. So the plan should be corrected to guard against that possibility. The implementation in Java in Listing 3.10 makes this correction with a crash-guard: an extra check of `seesSlot()` avoids calling the private method when there is no slot there.

Listing 3.10 The PairFinder class of objects

```
public class PairFinder extends Vic
{
   /** Tell whether there are two CDs in a row at any point at
    *  or after this position.  Leave the executor unchanged. */

   public boolean hasTwoTogether()
   {  String spot = getPosition();              // design step 1
      boolean hasTwoTogether = seesSlot()       // design step 2
                            && foundPair();
      while ( ! spot.equals (getPosition()))    // design step 3
         backUp();
      return hasTwoTogether;                    // design step 4
   }  //====================

   private boolean foundPair()
   {  boolean previousSlotIsEmpty = ! seesCD(); // design step 1
      moveOn();                                 // design step 2
      while (seesSlot())                        // design step 3
      {  if ( ! seesCD())                       // design step 3a
            previousSlotIsEmpty = true;
         else // has one in this slot           // design step 3b
            if (previousSlotIsEmpty)
               previousSlotIsEmpty = false;
            else                                // design step 3c
               return true;
         moveOn();                              // design step 3d
      }
      return false;                             // design step 4
   }  //====================
}
```

This logic uses the boolean variable `previousSlotIsEmpty` in a way you have not seen before. The purpose of such a variable is to store information obtained during one iteration of the loop to be used during the next iteration of the loop. It is best to name such a variable to convey its meaning at the time it is tested, not at the time it is assigned a value.

Is it a bad thing that the `foundPair` method changes the object and is thus not a true query method? No, it does not count as a style violation because (a) a call of `hasTwoTogether` does not, and (b) no one outside the class can call `foundPair`, since it is a private method.

Sequential/selection/repetition

The key activity in creating software is designing and implementing methods. Specifically, you design and implement a main method which calls on other methods which, unless you have them in your library, you must also design and implement. Some of those methods in turn can call on other methods which you must then design and implement or else find in your library of existing methods, and so forth.

Whatever the objects your software uses, whether Vics or Turtles or something else, the design of a method comes down to repeatedly choosing one of three kinds of activities, as follows. The last two kinds of activities listed are usually done with an if-statement or a while-statement, respectively:

- which sequence of actions you execute, or
- which query you test to determine which of two sequences of actions you execute, or
- which query you test to determine how many times you execute one sequence of actions.

Even when you start using numbers in your programs, you will find that the calculations or the tests for inequality you perform are all done only as part of actions or queries to be used as described in the preceding list. The list of activities can be summarized as: sequential, selection, and repetition.

In short, an essential part of programming is putting together actions and queries using `if` and `while` to create a method that performs a single well-defined task. Even though most of the programming you have seen has been in the highly limited context of Vics and Turtles, the skills and concepts you have learned are highly useful in most programming situations.

Loop control values

You must check out any looping logic you write to make sure it eventually terminates. The best way to do this is to make sure it has a **loop control value**. That is a numeric expression that (a) must be positive for the loop to continue executing, but (b) decrements by at least 1 each time the loop executes.

For most of the loops you have seen, the loop control value is the number of slots left in the sequence from the current position, since (a) the continuation condition usually tests `seesSlot()` to make sure it is true, and (b) each iteration of the loop executes `moveOn()`. That is, the number of slots left must be positive and each iteration subtracts 1 from the number of slots left.

For some loops you have seen, the loop control value is the number of CDs on the stack, since (a) the continuation condition usually tests `stackHasCD()` to make sure it is true, and (b) each iteration of the loop executes `putCD()`. That is, the number of CDs on the stack must be positive and each iteration subtract 1 from the number of CDs on the stack.

For some other loops you have seen, the loop control value is the number of slots between a previous position and the current position, since (a) the continuation condition tests `! spot.equals (getPosition())`, and (b) each iteration of the loop executes `backUp()`.

Exercise 3.42 Rewrite the `foundPair` method to have only one return statement.
Exercise 3.43** Rewrite the `foundPair` method to not use a boolean variable. Instead, when the executor sees a CD, have it go forward to see if there is one after it and, if not, move back again. Then discuss whether this is a better solution than the one in Listing 3.10. Can you think of a better solution than either?

*3.9 Turing Machines (*Enrichment)*

The Vic machine described in these two chapters is a modification of a Turing Machine. A Turing Machine is an extremely simplified version of a computer, one that is highly impractical for actual use. The advantage of this is that it is far easier to develop logical proofs about what is and is not computable by a computer if the computer has maximal simplicity.

The **Church-Turing Thesis** is that, for any computational process that can be programmed on any computer, some Turing Machine program carries out exactly the same process. This thesis is generally accepted by computer scientists. So when you see a proof in a Theory of Computation course that a certain problem cannot be solved by a Turing Machine program, that is accepted as a proof that no computer program will ever exist that can solve that problem.

A Turing Machine works with a sequence of positions (like Vic slots). The sequence is called a **tape**. Each position contains a single digit or else a blank. The machine begins operation at the far left of the sequence of positions. It is not allowed to back up past the position it starts on; the tape begins at that position. However, the tape goes on as far as necessary to the right (so there is no need for anything resembling `seesSlot()`).

A **Turing Machine** can check what digit is at its current position, if any; it can write a blank or any digit at the current position; and it can go forwards and backwards on the tape. To help you understand exactly what a Turing Machine is, we describe a class of objects similar to Vic. We could call it Tum for short (from TUringMachine). A Tum object understands only four basic commands:

- `sees(0)` tells whether there is a digit 0 at the current position, and similarly for other digits 1 through 9. The `sees(-1)` message tells whether there is a blank at the current position.
- `put(0)` puts the digit 0 at the current position, and similarly for other digits. Any negative value, as in `put(-1)` or `put(-30)`, puts a blank at the current position. The new value replaces whatever value was already at that position.
- `moveOn()` goes one position further right, away from the beginning of the tape.
- `backUp()` goes one position further left, towards the beginning of the tape. It crashes the program if the current position is the one at the tape's beginning.

You also have a strong restriction on how you can put these basic commands together to create new methods for subclasses of Tum:

- Each method is to have no parameters, no local variables, and no return value. So the only way to pass information around or store it is to put it on the tape.
- Each method body for a subclass of Tum is to consist of (a) at most one while-statement, followed by (b) at most one multi-way selection statement. No two conditions are to be true for the same digit. The subordinate statements in either case are simple method calls, selected from the four basic commands and other methods in a subclass of Tum.

An example of a permissible subclass of Tum is shown in Listing 3.11. The reason for the restrictions is that whatever you write can then be easily translated to a hypothetical machine code that has only one kind of instruction, structured as follows:

> If the current method is X and the current position contains Y then...
> Put Z in that position (or leave it unchanged if you wish).
> Move 1 position forward or backward (or remain there if you wish).
> Switch to some method (or not, as you wish).

Listing 3.11 A subclass of Tum

```
public class SampleTum extends Tum
{
   public void clear()
   {  while (sees (0) || sees (1))
      {  put (-1);
         moveOn();
      }
      if (sees (-1))
         backUp();
   }  //======================

   public void switch()
   {  if (sees (0))
      {  put (1);
         backUp();
      }
      else if (sees (1))
      {  put (0);
         backUp();
         clear();
      }
   }  //======================
}
```

Each method you could be in represents a different **state** of the Turing Machine. Since a program can only have a finite number of methods, this is a **finite-state machine**.

For this implementation in Java, you cannot write on the physical tape before the machine begins its operation or read the tape after it finishes. So you need a way to initialize the tape for the Tum object and a way to display the current status of the tape. This can be done using statements such as the following:

```
Tum sam = new Tum ("104 52");
sam.carryOutSomeProcess();
sam.showStatus (4);
```

The creation of a new Tum makes the tape consist of the given String of characters followed by many blanks. And the `showStatus` command displays the tape on the screen (plus the numeric parameter, which helps you figure out which call of `showStatus` produced which output). You will learn how to fully implement the Tum class as described here by the middle of Chapter Five. It is a major programming project to do this, so the Tum class is not provided here.

For a Turing Machine that works with binary numbers, you would only allow input consisting of 1s and 0s and blanks. You could then develop, for instance, a subclass of Tum that could add two such binary numbers together and leave the result on the tape for the `showStatus` message to display.

*3.10 Javadoc Tags (*Enrichment)*

As you learned in Chapter Two, a comment that begins with `/**` and ends with `*/` and comes immediately before a public class, public method, or public variable is special (you will see public variables in Chapter Five). When you give the command

```
javadoc SomeClass.java
```

in the terminal window, the javadoc formatting tool creates a webpage named `SomeClass.html` which displays documentation for the class in a useful form. The first complete sentence in each such comment is put in a summary section, so you want to make sure it conveys the key idea of your comment. Multi-line comments can have each line after the first begin with an asterisk if you like.

The javadoc tool creates several more html files for you. One is `index.html`, which lists in one section all of the methods in your class in alphabetical order with clear descriptions. It also lists any variables you have in another section. Another is `index-all.html`, which gives an alphabetical index of all the parts of your class. Browse one of these files and click on the Tree and Help options to see other documentation.

The twelve javadoc tags

You can put `@return` in a comment to tell the reader that the phrase that follows describes the value that is returned by a method. The javadoc tool will display it in a special way, because `@return` is one of the standard javadoc **tags**. The following are three tags that can be used in javadoc comments for classes, methods or variables:
`@see` lists other classes or methods that are highly related to this one.
`@since` tells which version of the software first had this feature.
`@deprecated` means it is outdated and should not be used anymore.

A class can have the following two tags:
`@author` tells the author of the coding.
`@version` tells the current version of the software.

Methods can have the following four tags:
`@param` describes a parameter of the method.
`@return` describes the value that the method returns, if it returns one.
`@throws` names the kind of Exception thrown and under what conditions. You will learn about Exceptions shortly; they almost never arise when programming with Vics.
`@exception` is the older form of `@throws`.

The three remaining permissible tags `@serial`, `@serialField`, and `@serialData` are ones for which you will not have any use for a long time.

3.11 Review Of Chapter Three

Listing 3.4 and Listing 3.7 illustrate almost all Java language features introduced in this chapter.

About the Java language:

- A while-statement states a **continuation condition** followed by the **subordinate statements**. The continuation condition must be true in order for the subordinate statements to be executed. Those statements are placed within matching braces unless there is only one subordinate statement. An **iteration** is one execution of the subordinate statements in this **loop**.
- `someString.equals(anotherString)` is a method in the **String** class in the Sun standard library. This method tests whether the two String values have the same content, i.e., the same characters in the same order.
- A method declared as `private` can only be called from within the class where it is defined. A method declared as `public` can be called from any class.
- You can use `this` inside an instance method as a reference to the executor of the method call. If you call an instance method without an executor, the compiler uses the **default executor**, which is `this` of the method containing the method call (i.e.,

it is `this` instance of the class). This only applies when the method call is itself inside an instance method, not a class method such as `main`.

- When a method heading has a variable declaration in its parentheses, each call of the method must have a value of the same type in its parentheses. When the method executes, this **formal parameter** is initialized to the value given in the method call (the **actual parameter**, also known as the **argument**).
- You can declare additional variables within the body of a method, e.g., booleans, Vics, and Strings. These **local variables** have no connection with variables outside of the method, and they have no initial value. You can only use a local variable after the point where it is declared and inside whatever braces contain the declaration.
- See Figure 3.8 for the remaining new language features. In that grammar summary, the `Type` could be a ClassName or boolean; an `ArgumentList` is a number of expressions separated by commas; and a `ParameterList` is a number of `Type VariableName` combinations separated by commas.

`while (Condition)` `  Statement`	**statement** that repeats test-Condition-do-Statement, quitting when the Condition is false

`while (Condition)`
`{  StatementGroup`
`}`
	statement with the same effect as described above, except the entire sequence of 0 or more statements is executed between tests.

`Type VariableName = Expression;`	**statement** that combines declaring and defining a variable
`ClassName.MethodName (ArgumentList)`	**expression** that calls a no-executor-method in the class
`VariableName.MethodName(ArgumentList)` `MethodName (ArgumentList)`	**expressions** that call an instance method with parameters

`public Type MethodName(ParameterList)`
`  { StatementGroup }`
`public void MethodName(ParameterList)`
`  { StatementGroup }`
	declarations of instance methods that accept input initially assigned to the formal parameters

Figure 3.8 Declarations, expressions, and statements added in Chapter Three

Other vocabulary to remember:

- When an object your program has created has no variable that refers to it, then the object is recycled by Java's **garbage collection** mechanism.
- The **hierarchy** of some classes is the set of relationships between subclasses and superclasses of that group of classes.
- **Structured Natural Language Design** expresses the logic of an **algorithm** completely in English or some other natural language, except that (a) statements that are executed conditionally (depending on whether some condition is true) are indented relative to the condition, and (b) some variable names are used.

About Vic methods (developed for this book):

- `someVic.getPosition()` returns a String that describes the current position in the sequence represented by `someVic`. If `x` and `y` are two such Strings returned when at the same position in the same sequence, then `x` and `y` may be different String objects, but it will be true that `x.equals(y)`.
- All other Vic methods were described in Chapter Two: four action instance methods (`moveOn`, `backUp`, `putCD`, `takeCD`), two query instance methods (`seesSlot`, `seesCD`), three class methods (`reset`, `say`, `stackHasCD`), and `new Vic()`.

About UML notation (all class diagram notations used in this book):

- A **class box** is a rectangle divided into three parts. The top part has the class name and the bottom part lists any of its method calls you wish to mention.
- A **dependency** of the form `X uses Y` is indicated by an arrow with a dotted line.
- A **generalization** of the form `X is a kind of Y` is indicated by an arrow with a solid line and a big triangular head.
- Class methods and class variables are to be underlined.
- You may add the parameter types in the parentheses after a method name.
- You may add the return type after those parentheses, with a colon in between.

Answers to Selected Exercises

3.1
```
public void removeAllCDs()
{   while (seesSlot())
    {   takeCD();
        moveOn();
    }
}
```
3.2
```
public void toLastSlot()
{   while (seesSlot())
        moveOn();
    backUp();
}
```
3.3
```
public void takeOneBefore()
{   backUp();
    while ( ! seesCD())
        backUp();
    takeCD();
}
```
3.6 Put an exclamation mark in front of seesCD().

3.7 You could insert the following lines after the first while-statement:
```
if (seesSlot())
{   moveOn();
    while (seesSlot() && ! seesCD())
        moveOn();
}
```
3.8 If there is no slot there, the program fails. And if seesSlot() is true, so is seesCD().

3.12 Remove the takeCD() method call from the first while statement and replace the second while statement by the following:
```
while ( ! spot.equals (getPosition()))
{   backUp();
    takeCD();
}
```
3.13
```
public boolean lastIsFilled()
{   String spot = getPosition();
    while (seesSlot())
        moveOn();
    backUp();
    boolean valueToReturn = seesCD();
    while ( ! spot.equals (getPosition())
        backUp();
    return valueToReturn;
}
```
3.16 "string" should be capitalized, but "Main" should not be capitalized.

3.17 You cannot assign a String value to a Vic variable, so change "Vic" to "String". It is bad style to capitalize the name of a variable, but it is not a compilation error.

3.18
```
public boolean hasSomeFilledSlot()
{   String spot = this.getPosition();
    while (this.seesSlot() && ! this.seesCD())
        this.moveOn();
    boolean valueToReturn = this.seesSlot();
    while ( ! spot.equals (this.getPosition()))
        this.backUp();
    return valueToReturn;
}
```

3.19
```
public void fillEvenSlots()
{   if (seesSlot())
    {   moveOn();
        fillOddSlots();
        backUp();
    }
}
```
3.20
```
public boolean seesOddsFilled()
{   String spot = getPosition();
    while (seesSlot())
    {   if ( ! seesCD())
        {   backUpTo (spot);
            return false;
        }
        moveOn();
        if (seesSlot())
            moveOn();
    }
    backUpTo (spot);
    return true;
}
```
3.21
```
public boolean seesEvensFilled()
{   if ( ! seesSlot())
        return true;
    moveOn();
    boolean valueToReturn = seesOddsFilled();
    backUp();
    return valueToReturn;
}
```
3.26 Change the while-condition to ! thisSpot.equals (par.getPosition())

3.27 Change the middle statement to the following:
```
boolean valueToReturn = this.seesSlot();
```
3.28
```
public boolean isAtOneGivenPosition (String one, String two)
{   return one.equals (this.getPosition()) || two.equals (this.getPosition());
}
```
3.29
```
public void moveToCorrespondingSlot (Vic par)
{   String thisSpot = this.getPosition();
    while (this.seesSlot() && par.seesSlot())
    {   if (par.seesCD() && ! this.seesCD())
        {   par.takeCD();
            this.putCD();
        }
        this.moveOn();
        par.moveOn();
    }
    while ( ! thisSpot.equals (this.getPosition()))
    {   this.backUp();
        par.backUp();
    }
}
```
3.35 Remove the exclamation mark from the if-condition.

3.36 Modify the hasAsManySlotsAs method in Listing 3.5 as follows:

Replace "boolean" by "Vic" in the method heading.

Replace the statement between the two while-loops by the following:
```
Vic valueToReturn;
if (this.seesSlot())
    valueToReturn = par;
else
    valueToReturn = this;
```
3.39
```
while (seesSlot())
{   moveOn();
    if (this.seesSlot())
        this.moveOn();
    else
        movedInPairs = false;
}
```
3.42 Replace "while (seesSlot())" by the following two lines:
```
boolean found = false;
while (seesSlot() && ! found)
```
Replace "return true" within the loop by "found = true".

Replace "return false" at the end by "return found".

Interlude: Integers And For-Loops

The next two chapters, Chapters Four and Five, explain how you store data values in objects using instance variables and how you store data values in classes using class variables. You use these language features in defining object classes that depend only on the Sun standard library. For this material, you will need to know how to store and work with whole-number values. This Interlude collects this information in one place, so that the following two chapters can concentrate just on language features for defining your own classes from scratch.

You have previously seen that a variable declared as type **boolean** can store either of the values `true` and `false`. You may assign to a boolean variable the result of a call of a boolean method or the result of combining boolean values using any of the three operators `!` (meaning "not"), `&&` (meaning "and"), and `||` (meaning "or").

You may also declare a variable as type **int** (rather than e.g. String or boolean). This means that you can store a whole-number value in that variable, as long as it is in the range plus or minus 2,147,483,647. The reason for that limit is that the storage space set aside for these whole-number values is limited to 31 base-two digits along with a plus or minus sign, and 2^{31} is 2,147,483,648, just slightly over two billion.

Increment and decrement operators

The following method could be in a subclass of Vic. It tells how many slots the Vic has. First, the original position is recorded in `spot`. Then each time the first while-loop moves forward in the sequence of slots, `count` is incremented by 1. When it runs out of slots, the second while-loop backs up to the original position. Then `count` is returned as the answer to the `getNumSlots()` query. Reminder: The body of a while-statement must be enclosed in braces `{ }` unless it consists of only one statement:

```
public int getNumSlots()
{  String spot = getPosition();
   int count = 0;
   while (seesSlot())
   {  count++;
      moveOn();
   }
   while ( ! spot.equals (getPosition()))
      backUp();
   return count;
}  //======================
```

The `++` symbol is the **increment** operator; it adds 1 to the int variable to which it is appended. So the expression `count++` changes the value of count to be 1 more than its current value. Java also has a **decrement** operator, e.g., the expression `num--` would subtract 1 from the value stored in the `num` variable:

A method that returns an int value does so analogously to methods that return boolean values: You specify int in the heading as its return value and then have one or more statements consisting of `return` followed by an int value. As with any `return` statement, executing this statement immediately stops execution of the method.

You may also have int variables as parameters. The following method could be in a subclass of Turtle; a sample call of this method is `tina.makeSquare(40)`. This method would be much more useful than the separate `makeBigSquare` and `makeSmallSquare` methods of the SmartTurtle class in Listing 1.3:

```
public void makeSquare (int side)
{  paint (90, side);
   paint (90, side);
   paint (90, side);
   paint (90, side);
}  //======================
```

Arithmetic operators for integers

A **binary operator** is a symbol which combines two values to obtain a new value. Those two values are the **operands** of that operator. For instance, `&&` and `||` are binary operators. By contrast, `!` is a **unary operator**: You apply it to only one value (operand) to obtain a new value. The words "binary" and "unary" come from the Latin for "two" and "one". Java provides five binary arithmetic operators for int values:

- **`x + y`** is the integer result of adding y to x;
- **`x - y`** is the integer result of subtracting y from x;
- **`x * y`** is the integer result of multiplying y times x;
- **`x / y`** is the integer quotient after dividing y into x; and
- **`x % y`** is the integer remainder after dividing y into x.

Since `x / y` is a whole number, the fractional part of the division is discarded. So `12 / 4` and `13 / 4` and `15 / 4` are all `3`. Since `x % y` is the remainder from the division, `12 % 4` is `0` and `13 % 4` is `1` and `15 % 4` is `3`. Negative numbers can be tricky: `(-13) / 4` is `-3` and so `(-13) % 4` is `-1`.

Caution If one operand of an arithmetic operator is formed with another arithmetic operator, you should use parentheses around that operand to make your meaning clear. The compiler applies normal algebra rules for clarifying an expression such as `x + y * z`, but it is safer to parenthesize. This book does so except for repeated additions.

Comparison operators for integers

Java has six binary operators which compare two numbers to obtain a true-false value:

- **`x > y`** means x is greater than y;
- **`x < y`** means x is less than y;
- **`x <= y`** means x is less than or equal to y;
- **`x >= y`** means x is greater than or equal to y;
- **`x == y`** means x equals y (note the DOUBLE equals-sign);
- **`x != y`** means x is NOT equal to y.

You could generalize the earlier `makeSquare` method to make any triangle, square, pentagon, or other regular polygon, by supplying a second int parameter that tells how many degrees to rotate each time (`turn` must evenly divide 360 for this to work right). In this method, the statement `total = total + turn;` means that you first calculate the sum of `total` and `turn` and then store the result in `total`:

```
public void makeRegularPolygon (int turn, int side)
{  paint (turn, side);
   int total = turn;
   while (total < 360)
   {  paint (turn, side);
      total = total + turn;
   }
}  //======================
```

Strings concatenated with ints and objects

String and int are two different types of values. Java is strongly typed, which means that there are strict limits on assigning a value of one type to a variable of another type. In particular, you cannot assign an int value to a String variable or return an int value when a String value is to be returned. Nor can you assign a String value to an int variable or return it from an int-returning method. However, the **concatenation operator** `+` will combine a String value and an int value into a String value by treating the int value as if it were a numeral, i.e., the string of digits as you would normally write them. So `"a" + 572` is the string `"a572"` and `"8" + "4"` is the String value `"84"`.

The following method call prints the value of the String parameter in the terminal window (often called the DOS window). The method call usually has a concatenation of a String and a numeric value as its parameter, to help in tracing the execution of the program:

```
System.out.println ("the value of x is " + x);
```

If a method is to return a String value and you have declared `int x` in that method, the statement `return x;` would not even compile, because you cannot return an int value when a String value is required. But concatenating the empty String `""` with the int value makes a String value expression, and it is legal to return that value from the method. So you could have `return "" + x;` as a statement in the method.

General principle When you "add" a String value to anything, you always get a String value. This definition of the plus sign makes coding in Java easier. The compiler expects that you made a mistake if you assigned a non-String value to a String variable or returned a non-String value from a String method. But if you explicitly add "" to a non-String value, you are both acknowledging and correcting the incompatibility.

Java allows variables to be declared as type `double` as well as `int`, which means they can store numbers with decimal points (such as 4.7 and -0.53). Chapter Six contains detailed discussion of the use of this type of number. We do not need it until then.

Language elements

A Statement can be: VariableName ++ ;
or: VariableName -- ;
You may use int as the declared type of a variable or as the return type of a method.
An int literal is a sequence of digits, optionally preceded by a negative sign.
The operators that combine two int values to get an int value are + - * / %
The operators that combine two int values to get a boolean value are > < == != >= <=
You may use a plus sign between a numeric value and a String value. This concatenates the numeral with the string of characters.

Loop controls

The `makeBigSquare()` method of the Turtle class (Section 1.4) contained just four statements, each being the command `paint(90,40)`. And the `makeHexagon()` method contained just six statements, each being the command `paint(60,30)`. The following expresses the same logic using while-statements:

```
public void makeBigSquare()
{  int k = 0;
   while (k < 4)
   {  paint (90, 40);
      k++;
   }
}  //====================
```

```
public void makeHexagon()
{  int k = 0;
   while (k < 6)
   {  paint (60, 30);
      k++;
   }
}  //====================
```

In each case, the while-statement tests the value of `k` to see if the loop should continue (`k < someValue`). Since `k` is the only variable in the continuation condition whose value is changed by the loop, it is the **loop control variable** in each case. The while-statement is preceded by an **initializer step** `int k = 0`, to assign the starting value of the loop control variable. And the last statement in each loop is an **update step** `k++`, whose purpose is to modify the value of the loop control variable.

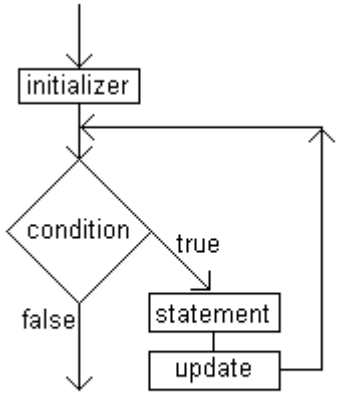
The for-statement

Java's for-statement allows us to bring these three intimately-connected parts of the logic together in one place, to make the overall structure clear. Those two earlier methods can be written as follows with exactly the same effect:

```
public void makeBigSquare()
{  for (int k = 0;  k < 4;  k++)
      paint (90, 40);
}  //======================

public void makeHexagon()
{  for (int k = 0;  k < 6;  k++)
      paint (60, 30);
}  //======================
```

In general, `for(initializer; condition; update){...}` is a loop that executes as long as the condition is true, with the initializing command executed just before the first iteration and the updating command executed at the end of each iteration. The two semicolons are required inside the parentheses. As with the other structured statements using `if` and `while`, you may omit the braces around the subordinate part of a for-statement if it is just one statement.



for (initializer; condition; update) statement

Figure 1 Flow-of-control for the for-statement

The `makeRegularPolygon` method shown earlier in this Interlude can be written using a for-statement as follows. It illustrates the fact that that the loop control variable does not always have to increment or decrement by 1 each time:

```
public void makeRegularPolygon (int turn, int side)
{  for (int total = 0;  total < 360;  total = total + turn)
      paint (turn, side);
}  //======================
```

You are not required to declare the loop control variable in the initializing step. But if you do declare a variable in the initializing step of a for-statement, the compiler does not allow you to mention it outside of the for-statement.

Progressing through a sequence of integers

The following method finds the first power of 2 that is greater than a given int value:

```
public int powerGreaterThan (int given)
{  int power;
   for (power = 1;  given >= 1;  power = 2 * power)
      given = given / 2;
   return power;
}  //======================
```

However, this is going too far. If you are undisciplined in the use of the for-statement, it loses its basic meaning of "continuation condition together with the update of the variable on which the continuation condition depends". It is not a coincidence that the preceding for-statements have an integer as the loop control variable. The common features in the earlier examples are as follows:

- The updating phrase moves the loop control variable one step further in a sequence of integers or the equivalent.
- The body of the for-statement does not change the position of the loop control variable in the sequence of integers.
- The continuation condition stops the loop when the loop control variable reaches the end of the sequence of integers, if not before.

Programming Style Some programmers use a for-statement wantonly, ignoring its basic meaning. This book restricts the use of the for-statement to situations with the three features above. It is also good style to declare the loop control variable in the for-statement heading where possible.

There are some cases in which the loop control variable is given its initial value several statements before the for-statement. You may leave out the initializer step in such a case, so that the first thing inside the parentheses of the for-statement heading is a semicolon. But the two semicolons are required in the heading of the for-statement even when the initializer step is missing.

Caution When the body of a for-statement or while-statement is a single statement rather than something in braces, that single statement cannot be a variable declaration. Similarly, when one of the two subordinate parts of an if-statement is a single statement not within braces, the statement cannot be a variable declaration. For example, the compiler does not allow the phrase `if (x == y) int max = z;`.

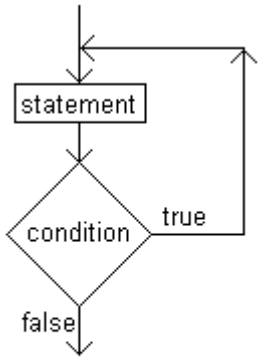
The do-while statement

You have now seen two looping statements in Java, the while-statement and the for-statement. Java has one more, the **do-while statement**. The following coding does exactly what a while-statement would do, repeatedly executing the two statements, except that no test of the condition is made until after the first time through the loop:

```
do
{  paint (turn, side);
   total = total + turn;
}while (total < 360);
```

The need for the do-while statement rarely arises. We will not use it in this book except in case studies at the ends of Chapters Four and Five.

Programming Style The do-while statement is one case where you should put something on the line with the ending brace. Do it because, if the while-part were on the next line, it would deceive people into thinking it is the beginning of a new while statement instead of the end of a do-while statement. Deceiving people is not good style. You will further avoid confusion if you use braces around the body of the do-while statement even when it has only one statement in it.



do
statement
while (condition);

Figure 2 Flow-of-control for the do-while statement

Language elements

A Statement can be: for (Initializer ; Condition ; Update) Statement
or: for (Initializer ; Condition ; Update) { StatementGroup }
or: do { StatementGroup } while (Condition) ;
An Initializer can be: int VariableName = Expression
or: VariableName ++
or: VariableName --
or: VariableName = Expression
The Update part can be an assignment or a method call.
Any one of the three parts of a for-statement can be left out, but keep the two semicolons.
If the condition is left out, the loop continues until a return statement in its body is executed.

Exercise 1 If x is 23 and y is 5, what are `x / y`, `(x+1) / y`, and `(x+2) / y`?
Exercise 2 If x is 23 and y is 5, what are `x % y`, `(x+1) % y`, and `(x+2) % y`?
Exercise 3 How would you revise the `getNumSlots` method in this section to return the number of slots that contain CDs?
Exercise 4 Write a method `public void FiveCircles()` for a subclass of Turtle: The executor draw five circles, all with the same center, but with diameters of 60, 120, 180, 240, and 300. Use a for-statement.
Exercise 5 Write an action method for a subclass of Turtle that has the executor print the word "Hi" seven times, moving ahead 20 pixels after each word. Use a for-statement. Use a parameter for 7.
Exercise 6 How many times does each of the following print "Hi"?
(a) `for (int k = -4; k <= 5; k++) System.out.println ("Hi");`
(b) `for (int k = -2; k < 8; k++) System.out.println ("Hi");`
(c) `for (int k = 3; k >= -6; k--) System.out.println ("Hi");`
Exercise 7 Rewrite the while-statement in the main method at the end of Section 3.1 as a do-while statement.

Answers to Exercises

1 23 / 5 is 4. 24 / 5 is 4. 25 / 5 is 5.
2 23 % 5 is 3. 24 % 5 is 4. 25 % 5 is 0.
3 Replace the count++ statement by: if (seesCD()) count++;
4
```
public void FiveCircles()
{   for (int radius = 30;  radius <= 150;  radius = radius + 30)
        swingAround (radius);
}
```
5
```
public void ManyHi (int numWords)
{   for (int k = 0;  k < numWords;  k++)
    {   say ("Hi");
        move (0, 20);
    }
}
```
6 (a) 10 times. (b) 10 times; (c) 10 times.
7
```
do
{   sequence.takeCD();
    sequence = new Vic();
}while (sequence.seesSlot());
```

4 Instance Variables

Overview

So far you have derived new classes that extend what objects can do. This chapter shows you how to derive new classes that extend what the objects know. These programs will be developed entirely with classes from the Sun standard library. So you will be able to run these programs without having the Vic or Turtle class or their analogs.

This chapter introduces a new context for learning about object-oriented software design. You are to develop several programs, each allowing the user to play a game against the computer. Some examples of such games are Checkers and Solitaire (although these particular games are not developed in this chapter). You only need study through Section 4.7 to understand the material in the rest of this book.

- Section 4.1 develops the overall analysis and design for game programs.
- Section 4.2-4.5 describe language features you will need for these game programs: graphical input/output, instance variables, constructors, and whole numbers.
- Section 4.6 implements fully a game of guess-my-randomly-chosen-number.
- Sections 4.7-4.8 explain overloading, overriding, polymorphism, and precedence.
- Sections 4.9-4.10 implement fully the games of Nim and Mastermind, as further examples of analysis and design.

4.1 Analysis And Design Of A Game Program

The computer games we develop will all have the same general structure. To get a fix on that structure, we begin by creating a **prototype**, a program that has the proper structure but plays a very trivial game. This gives us a solid foundation for the real games.

For this trivial game, the user tries to guess the secret word the computer has chosen. It is trivial because the secret word is not so secret -- it is "duck". The main application only needs to create a game-playing object and tell it to play. This straight-line logic is shown in Listing 4.1. For any of the more complicated games to be developed later, the main application would be the same except with "BasicGame" changed to "Mastermind" or "TicTacToe" or "Checkers" or whatever is appropriate.

Listing 4.1 A game-playing application program

```
public class GameApp
{
   /** Play the basic game repeatedly until the user tires.  */

   public static void main (String[ ] args)
   {  BasicGame game;           // declare the game variable
      game = new BasicGame();   // create an object to put in it
      game.playManyGames();     // tell the object to play
      System.exit (0);          // terminate the GUI interface
   }  //=====================
}
```

The `System.exit(0)` command simply terminates the program. You have to have it in Listing 4.1 because the game-playing object will use a graphical interface. On some computer systems, the computer will lock up if you finish running a program that has a

graphics interface but never execute the `System.exit(0)` command. You did not have to put this command in your Vic programs because it is already in the method that reacts to clicking the X-shaped window-closer icon in the top right corner of the window. System is a class in the Sun standard library, and `exit` is a class method in System.

Analysis and logic design

The way the computer plays many games is quite simple: First it plays a game. Then it asks the user if he/she wants to play another game. If the answer is no, the process stops, otherwise the computer plays another game and the cycle repeats. This general process applies to any of the several games we will develop. That is, once we have implemented it for a BasicGame, we can use that implementation for other games such as Checkers or Chess.

The progress of any one of these games is as follows: First set up the initial state of the game (in Mastermind, choose a 3-digit random number; in TicTacToe, create an empty 3x3 board; in Checkers, create an 8x8 board with 12 red and 12 black pieces placed in the appropriate squares; etc). Second, ask the user for his/her first move in the game. Then see whether that move wins the game (in almost all games, it will not) or, for that matter, loses the game.

If the user's move does not terminate the game, the computer makes its move in response or takes whatever action is appropriate, then the user takes another move or makes another choice. This cycle continues until the game is over. A reasonable plan for the logic design of the BasicGame is shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for BasicGame's playOneGame
1. Ask the user for his/her first guess.
2. Repeat the following until the current guess is completely right...
    2a. Tell the user he/she is wrong, and give a hint.
    2b. Ask the user for his/her next guess.
3. Tell the user that he/she has finally gotten it right.

This logic can be implemented with the following coding for the playOneGame method. As usual, empty parentheses indicate a method that has no parameters:

```
public void playOneGame()
{  askUsersFirstChoice();
   while (shouldContinue())
   {  showUpdatedStatus();
      askUsersNextChoice();
   }
   showFinalStatus();
}  //=====================
```

Asking the user whether another game is to be played, or asking for the next guess, requires a method for getting input from the user. And telling the user that the guess is right or wrong requires a method for displaying output to the user. The next section introduces methods from the Sun standard library that allow this.

4.2 Input And Output With JOptionPane Dialog Boxes

The various compiled classes are organized into categories called **packages**. You simply put `package X;` as the top line in a file to make the class in that file be in package X. If you do not specify a package for a class when you compile it, its package is the folder on the hard disk where its file is stored.

The JOptionPane class

The **JOptionPane** class is in the `javax.swing` package (technically, swing is the "subpackage" of the javax package). JOptionPane provides some components for a GUI (Graphical User Interface). A file containing a class that uses the JOptionPane class should have the **import directive** `import javax.swing.JOptionPane;` at the top of the file. This line directs the compiler to look in the Sun standard library package named `javax.swing` for the class. If you do not have the import directive, the compiler is not able to find JOptionPane.

The System and String classes are in the standard `java.lang` package, which the compiler automatically makes available to every class. If you want to use any class that is not in `java.lang` or in your current package, you have to give an import directive to tell the compiler where to find it. You need one import directive for each such class, except if you use several classes from the same package, e.g., `javax.swing`, you may have `import javax.swing.*;` as a shorthand for all import directives for classes from that same package.

The showMessageDialog method

The JOptionPane class has three class methods that we will use (i.e., you call them with the class name in place of the executor object). One of these is the `showMessageDialog` method that displays a small rectangle containing a message. It does not ask for user input. An example of its use is as follows:

```
JOptionPane.showMessageDialog (null, "Hi there!");
```

The first parameter of the `showMessageDialog` method call is the window to display the **dialog box** in. When this window value is `null`, the value that indicates the absence of an object, the dialog box appears in the middle of the monitor screen. This is the only way we will use the `showMessageDialog` method.

The second parameter of `showMessageDialog` is the message to be displayed. In the preceding example, it is characters in quotes, called a **String literal**. The right side of Figure 4.1 shows an example of what such a message dialog box looks like.

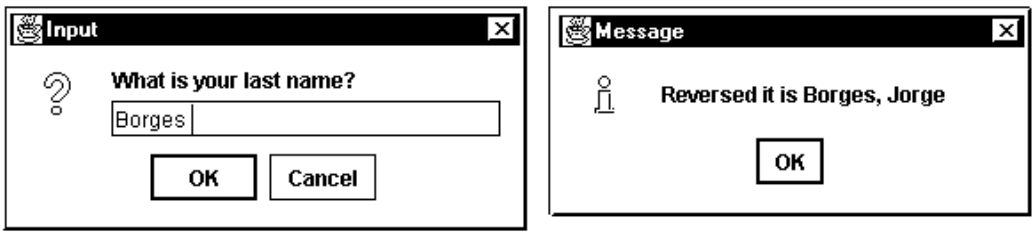


Figure 4.1 Screen shots of calls of showMessageDialog and showInputDialog

If you want to display several lines on this dialog box, you should use the `\n` symbol in the String literal. Each such use causes the start of a new line; `'\n'` is called the **newline** character. So "Hi \nthere \nCathy" displays those three words on three separate lines. The `showFinalStatus` method of the BasicGame class need only print out a message saying the user guessed the secret word right, so it could have this statement:

```
JOptionPane.showMessageDialog (null,
          "That was right.  \nCongratulations.");
```

The showInputDialog method

The `showInputDialog` method in JOptionPane displays a small rectangle containing a given string of characters (the only parameter). Then it waits until the user types something in the empty text box and presses the OK button or the ENTER key. When the user does so, the method returns the string of characters the user typed. The left side of Figure 4.1 shows what this dialog box looks like (the vertical line is the cursor). For instance, the `askUsersNextChoice` method of the BasicGame class could consist of the following statement, storing the input in a String variable named `itsUsersWord`:

```
itsUsersWord = JOptionPane.showInputDialog
                ("Guess the secret word: ");
```

Listing 4.2 is an application program that illustrates the use of these two JOptionPane methods. It asks the user for the user's first name and last name, stores each in a String variable, then prints them out in reverse order with a comma between them (e.g., "Jones, Bill"). It needs the `System.exit(0)` because it uses JOptionPane's graphic interface. Figure 4.1 shows what you might see on the screen when the program runs. Note: This is one of the few application programs in this book that does not call an instance method.

Listing 4.2 An application program illustrating the use of JOptionPane

```
import javax.swing.JOptionPane;

public class NameChange
{
   /** Ask the user for his/her first name, then for the last
    *  name, then print them out in the opposite order.  */

   public static void main (String[ ] args)
   {  JOptionPane.showMessageDialog (null,
              "Illustrate the use of JOptionPane methods");
      String firstName = JOptionPane.showInputDialog
                         ("What is your first name?");
      String lastName = JOptionPane.showInputDialog
                         ("What is your last name?");
      JOptionPane.showMessageDialog (null,
              "Reversed it is " + lastName + ", " + firstName);
      System.exit (0);   // needed when using JOptionPane
   }  //====================
}
```

The next-to-last statement of Listing 4.2 uses a plus sign between pairs of String values to indicate that the message is a single String value obtained by combining the four String values into one long String value. This is concatenation of string values.

The showConfirmDialog method

JOptionPane also has a method that asks a yes/no question (supplied as a String for the second parameter; the first parameter is again `null`). If the user clicks the YES option, the value returned is an int value named YES_OPTION, defined in the JOptionPane class. The `==` operator tests the value to see if it equals the YES_OPTION, so the following logic could be used for the `playManyGames` method of the BasicGame class:

```
playOneGame();
while (JOptionPane.showConfirmDialog (null, "again?")
                == JOptionPane.YES_OPTION)
   playOneGame();
```

As usual, we can omit the braces around the body of the while-statement when it only contains one statement. Note: Due to a glitch in this Sun standard library method, the user cannot just tab over to the NO or CANCEL option and then press the ENTER key; the user must actually click NO or CANCEL to terminate this loop.

The null value

The value **null** is a special value that can be assigned to any object variable. It indicates the absence of a reference to an actual object. For instance, if `sam` refers to some String object and you do not want it to do so anymore, execute the command `sam = null`. That erases the object reference in `sam`. `sam` then refers to no object at all, so of course you cannot execute `sam.equals(x)` or any other such method call -- it would not make sense to ask a nonexistent object to answer a question or to perform an action.

If the user clicks the OK button in response to the `showInputDialog` message without entering any value, it returns the empty String, a String with no characters, written as `""`. But if the user clicks the Cancel button or the closer icon in the upper-right corner, the method returns the `null` value. Your coding must allow for these two possibilities.

Indenting a Java program

Programming Style In Listing 4.2, four of the five statements do not fit on one line. It is good style in such cases to strongly indent the continuation of the line, so a reader does not think it is a separate statement. The line break should come after a comma or semicolon or before a left parenthesis or an operator (such as `+` and `>`).

In general, it is good programming style to indent your Java logic in the conventional way:

1. Indent one tab position for the heading of a method and two tabs for its body.
2. Indent one extra tab position for a statement subordinate to `if`, `while`, `do`, or `for` (the for-statement is described in Chapter Five).
3. Indent two extra tab positions or more for a continuation of a program line onto the next line.
4. Indent almost nowhere else.

Language elements

You may put an import directive before the first class in a file. Use one of these two formats:

```
import PackageDescription . ClassName ;
import PackageDescription . * ;
```

The value null may be assigned to any object variable or otherwise used as an object value.
The \n sequence within quotes indicates that the start of a new output line occurs at that point.

Exercise 4.1* Write an application program that asks the user for two strings of characters and then tells whether they are equal to each other.
Exercise 4.2* Draw the UML class diagram for Listing 4.2.

4.3 Declaring Instance Variables: A First Look At Encapsulation

The only way you have previously seen to store information is in variables declared within a method. Those variables "belong" to the method and only exist during a single execution of the method . But in order for a BasicGame object to do what it has to do, such as decide whether the current game should continue, it needs to know two things: What is the secret word and what is the user's most recent choice. Information is stored in variables, but a local variable is not suitable for this information.

Declaring instance variables

You need to store these two pieces of information as part of the BasicGame object. Java indicates this relation by having the variables declared outside of any method to distinguish them from local variables. You could name them `itsSecretWord` and `itsUsersWord`. They are called **instance variables**, because each instance (object) of the BasicGame class has its own separate values stored in these variables.

These two instance variables are declared as **private variables** in the BasicGame class. This means that methods outside of the BasicGame class cannot access them directly. It is legal to declare them as public instead, but we rarely do that with instance variables. Bugs in a program are usually easier to avoid if we prevent direct access by outsiders to the instance variables of objects. This is called **encapsulation**. Since this simplest of all games has the secret word "duck", and the user has made no guess when the game is first created, the BasicGame class contains the following two declarations:

```
private String itsSecretWord = "duck";
private String itsUsersWord = "none";
```

Each time a BasicGame object is created, it is given these two variables, where the initial value of the first is "duck" and the initial value of the second is "none". These declarations are exactly the same as for local variables except they have `private` at the beginning. Note: Some books define encapsulation more generally, to mean keeping data and objects together in one class so that outside classes can access variables only indirectly, through method calls that protect against inappropriate changes.

Using instance variables

Suppose `sam` refers to an instance of a class that has an instance variable `someVar`. Then `sam.someVar` is the `someVar` variable belonging to the object `sam` refers to. Within an instance method, `this.someVar` is the variable that belongs to the executor, and so is just plain `someVar` (using the default executor).

This is analogous to what you already know about instance methods: If `sam` refers to an instance of a class, then `sam.someMethod()` calls the `someMethod` belonging to the object `sam` refers to. Within an instance method, `this.someMethod()` calls the method that belongs to the executor, and so does just plain `someMethod()`.

Listing 4.3 (see next page) shows the complete BasicGame class. Some people prefer to put the instance variables first in the class, as shown, and others prefer to put them last in the class. The order of the declarations of instance variables and instance methods in a class generally has no effect on whether the class compiles or how it executes.

The logic of the methods for the basic game in Listing 4.3 was discussed earlier:

- `askUsersFirstChoice` stores the input in the game's own variable named `itsUsersWord`.

- `shouldContinue` is true whenever the two words are not the same word. The `equals` method for Strings tests whether two objects have the same contents. If the user clicked Cancel, the input is `null`, and the `equals` method returns `false` when the parameter is `null`.
- `askUsersNextChoice` does the same as `askUsersFirstChoice` for this game.
- The two methods that show the status simply print an appropriate message.

Listing 4.3 The BasicGame class of objects

```
import javax.swing.JOptionPane;

public class BasicGame extends Object
{
   private String itsSecretWord = "duck";
   private String itsUsersWord = "none";


   public void playManyGames()
   {  playOneGame();
      while (JOptionPane.showConfirmDialog (null, "again?")
                     == JOptionPane.YES_OPTION)
         playOneGame();
   }  //======================

   public void playOneGame()
   {  askUsersFirstChoice();
      while (shouldContinue())
      {  showUpdatedStatus();
         askUsersNextChoice();
      }
      showFinalStatus();
   }  //======================

   public void askUsersFirstChoice()
   {  itsUsersWord = JOptionPane.showInputDialog
                     ("Guess the secret word:");
   }  //======================

   public boolean shouldContinue()
   {  return ! itsSecretWord.equals (itsUsersWord);
   }  //======================

   public void showUpdatedStatus()
   {  JOptionPane.showMessageDialog (null,
                     "That was wrong.  Hint:  It quacks.");
   }  //======================

   public void askUsersNextChoice()
   {  askUsersFirstChoice(); // no need to write the coding again
   }  //======================

   public void showFinalStatus()
   {  JOptionPane.showMessageDialog (null,
                     "That was right.  \nCongratulations.");
   }  //======================
}
```

The **state** of an object is determined by its instance variables. They give the object its "personality". A method that modifies the executor's state and does nothing else is a **mutator** method (a special kind of action method, e.g., `askUsersNextChoice`). A method that merely accesses the executor's state is an **accessor** method (a special kind of query method, e.g., `shouldContinue`).

The Object class

The superclass for the BasicGame class is the Object class. **Object** is a class in the Sun standard library. It is in the `java.lang` package, which means you do not need an import directive to help the compiler find it. The Object class is the ultimate superclass of every class in Java; if you define any class without an `extends` phrase, the compiler supplies the **default extension** `extends Object`. So we could have left that phrase out of Listing 4.3 with no difference in effect. This book leaves it out for classes that do not have any instance methods or instance variables.

The Object class contains a method named toString. It returns a String equivalent of the object. A plus sign between a String value and an object reference `x` causes the object reference to be interpreted as `x.toString()`, using that object's `toString` method. So if `now` is an object variable for which `now.toString()` has the value `"0730"`, then the phrase `"now = " + now` has the value `"now = 0730"`.

Exercise 4.3 Revise the `shouldContinue` method to change the secret word to "goose" for the second and later guesses if the first guess is not "duck".
Exercise 4.4 Revise the BasicGame logic so that the user gets it right after at most five guesses, as follows: Add an instance variable initialized to be the empty String at the start of each game. Concatenate "x" to it every time the user makes a guess. Then count whatever answer the user gives as right when that instance variable has the value "xxxxx".
Exercise 4.5* Revise the BasicGame logic so that the user who takes more than three guesses to get it right will never succeed in guessing the right answer. Hint: See the preceding exercise.
Exercise 4.6* Explain the difference between an instance variable and a local variable.

4.4 Defining Constructors; Inheritance

When you create a new object, as in `sam = new SmartTurtle()` or `sue = new Vic()`, you do it by calling on a **constructor** method in that class. If you (or whoever developed the class) did not provide one or more constructor methods, the compiler provides one for you by default. It is as follows for the SmartTurtle class:

```
public SmartTurtle()  // default constructor
{  super();
}  //=======================
```

This **default constructor** (supplied by the compiler) is the method that `sam = new SmartTurtle()` calls, since the SmartTurtle class (Listing 1.3) does not explicitly define a constructor. The default constructor says you construct a new SmartTurtle object by doing nothing but calling the constructor method of the superclass of the SmartTurtle class. That is, of course, the Turtle class. The Turtle class provides an explicit constructor which the SmartTurtle constructor calls with the command `super()`.

The default constructor for the BasicGame class or any other class is the same as the above except of course the method heading is different, e.g.:

```
public BasicGame()  // default constructor
{  super();
}  //======================
```

`super()` calls the no-parameter constructor of the superclass, which for BasicGame is the Object class. The Object class has a number of instance variables and methods that all objects need. One instance variable tells the class of the object, so the runtime system can inspect it to see what methods it is allowed to call. Another instance variable keeps track of information that the automatic garbage collection process needs. These Object instance variables are private, so you cannot access them in a program.

Explicitly defined constructors

When you define a class for which you want something more than what the default constructor gives you, you have to define the constructor(s) explicitly in the class. The first statement should always call `super()` (or its equivalent, discussed later). `super()` is only allowed for the first statement of a constructor.

Listing 4.3 specifies initial values for the two instance variables of a BasicGame. An alternative is to not have an assignment in the declaration, but to assign the value in the constructor itself. That is, the two lines declaring the instance variables in Listing 4.3 would be replaced by the following variable declarations and constructor:

```
private String itsSecretWord;
private String itsUsersWord;

public BasicGame()            // constructor
{  super();
   itsSecretWord = "duck";
   itsUsersWord = "none";
}  //======================
```

There is no reason to do so in this case, but usually it is necessary. For example, suppose you want a Vic object to be able to answer the question, "Was your first slot empty when you were created?" The answer is `true` if the Vic saw a slot but did not see a CD in that slot. You could test `someVic.firstWasEmpty()` at any time if you have the following instance variable and two methods in a subclass of Vic:

```
public class GoodVic extends Vic
{
   private boolean itsFirstWasEmpty;

   public GoodVic()           // constructor
   {  super();
      itsFirstWasEmpty = seesSlot() && ! seesCD();
   }  //======================

   public boolean firstWasEmpty()
   {  return itsFirstWasEmpty;
   }  //======================
}
```

Inheritance

An inconvenience with Vics is that you cannot return to the front of the sequence because you do not know how far back to go. That can easily be fixed if you define the following class of objects that know their starting position:

```
public class BasicVic extends Vic
{
   private String itsInitialPosition;

   public BasicVic()          // constructor
   {  super();
      itsInitialPosition = getPosition();
   }  //======================

   public void returnToStart()
   {  while ( ! itsInitialPosition.equals (getPosition()))
         backUp();
   }  //======================
}
```

Now if you change the heading on any subclass of Vic to say `extends BasicVic` instead of `extends Vic`, it inherits the `returnToStart` method. Then you can execute `sam.returnToStart()` for any object `sam` of that subclass, to have `sam` repositioned at the front of its sequence. Reminder: A subclass **inherits** all public methods and variables of the superclass it extends, i.e., it can refer to them as if they were defined in the subclass.

Parameters of constructors

A constructor may have one or more parameters. For instance, you might want to define a class of objects that remember their first names and their last names and can provide them on request. The Person class in the upper part of Listing 4.4 (see next page) is such a class. You then must supply both names when you construct a new Person object, as in the following statement:

```
al = new Person ("Jorge", "Borges");
String fullName = al.getFirstName() + " " + al.getLastName();
```

The constructor in the Person class illustrates the most commonly used format for constructors, namely, the parameters supply the initial values of instance variables. You could call it the **natural constructor**, since the primary purpose of a constructor is to initialize all instance variables.

To develop software for a hospital, you might define Patient as a subclass of Person. Then the first statement in the Patient constructor is the `super` call of the constructor from its superclass, which is Person. Constructing a Person object requires supplying the first and last name. So Patient's `super` call must include those two parameters to initialize the private instance variables in the Person superclass. You cannot use plain `super()`, since the Person class does not have a constructor with no parameters. Therefore, part of the Patient class could be as shown in the lower part of Listing 4.4.

It is legal to omit the call of `super`, though this book does not do so. If you do so, the compiler inserts `super()` (i.e., no parameters) in your constructor by default. So the `super` statement in the Person constructor is optional, but the explicit call of `super` in the Patient constructor is required because the insertion of `super()` would call a non-existent constructor with no parameters, which the compiler will not allow.

Listing 4.4 The Person class of objects

```
public class Person extends Object
{
   private String itsFirstName;
   private String itsLastName;

   public Person (String first, String last)    // constructor
   {  super();
      itsFirstName = first;
      itsLastName = last;
   }  //======================

   public String getFirstName()
   {  return itsFirstName;
   }  //======================

   public String getLastName()
   {  return itsLastName;
   }  //======================
}
//############################################################


public class Patient extends Person
{
   private String itsDoctor;

   public Patient (String first, String last, String doc)
   {  super (first, last);
      itsDoctor = doc;
   }  //======================
}
```

Name shadowing

Caution A method is allowed to have a local variable or parameter with the same name as an instance variable. But it usually causes trouble. Within that method, the instance variable is **shadowed**: Use of the variable name is a reference to the locally declared variable, not to the instance variable. A common logic error is to "redeclare" an instance variable; e.g., to write the last statement of the Patient constructor as `String itsDoctor = doc`. That extra word String means you are declaring a new variable named `itsDoctor` local to the method. The compiler will not point out that it is surely not what you meant to do.

A related error occurs in the Patient constructor if you name the instance variable `doc`, the same name as the parameter, and so the last statement is `doc = doc`. This merely assigns the value of the parameter to the parameter, not to the instance variable. This form of assignment is never necessary and usually indicates a logic error in the program.

This book puts a prefix of "its" on almost all instance variables and never anywhere else. This hallmark makes both of the above-mentioned bugs less likely. Some people prefer the prefix "my", as in `myFirstName`, `myLastName`, `myDoctor`. If you do not do this sort of thing in your own definitions, at least obey the following safety principle: Never name a parameter or local variable the same as an instance variable.

Default variable values

If you do not assign a value to an instance variable in its declaration, the compiler assigns one for you: The **default initial value** is `null` for any object variable (such as String), `false` for any boolean variable, and zero for any other variable. You may re-initialize this value during execution of the constructor.

Programming Style All instance variables in this book are explicitly initialized to a value even if it is what the default would give. This makes programs clearer. The initialization is normally made in the declaration rather than the constructor when possible, to make it clear to the reader that the initial value does not depend on the constructor's parameters.

The use of this for instance variables

If the body of instance method X contains a method call without saying which object is its executor, then its executor is by default the executor of X. The same principle applies to instance variables: If the body of instance method X mentions an instance variable without saying which object it belongs to, it is by default a use of the executor's instance variable. Since the keyword `this` can be used in an instance method to refer to the executor of the method, the statement in the Person's `getFirstName` method of Listing 4.4 could be written as `return this.itsFirstName;` to do the same thing.

A constructor method is technically neither a class method nor an instance method; it is a method of constructing objects of the class. Within a constructor definition, the use of an instance variable without saying which object it belongs to is by default a use of the instance variable of the object being constructed (the same is true of instance methods). The pronoun `this` refers to the object being constructed. So the last two statements of the Person constructor in Listing 4.4 could be written as follows without changing the constructor's effect:

```
this.itsFirstName = first;
this.itsLastName = last;
```

Caution Do not assign a value to `this`, as in `this = whatever;` it is never necessary, and it is almost always an indication of an error in logic.

Language elements

Instance variables are declared outside of any method. You may use one of these two formats:

```
private Type VariableName = Expression;
private Type VariableName ;
```

A Declaration may be: public ClassName (Parameters) { StatementGroup }

A Statement may be: super (ExpressionsSeparatedByCommas) ;

Such a statement is only allowed as the first statement in a constructor.

Exercise 4.7 Write out the default constructor for the SmartTurtle class.

Exercise 4.8 Write a RedTurtle constructor for a subclass of Turtle, in which the turtle object always starts with the drawing color red and a due west heading.

Exercise 4.9 Define a NamedTurtle class: Objects are given a name when constructed (via a parameter) and they tell you what the name is when you use the `getName` method in an expression such as `sam.say ("I am " + sam.getName())`.

Exercise 4.10* Define a Terrapin subclass of the NamedTurtle class in the preceding exercise. A Terrapin is given its name and its best friend (another Turtle object) when constructed, and can tell you its name and who its best friend is when asked.

4.5 Integer Instance Variables

The next game to be developed uses numbers. You can store numeric information in an instance variable of an object. For example, if you want each Person object to remember its birth year, you could add an instance variable to the Person class of Listing 4.4 as follows:

```
int itsBirthYear;  // instance variable for Persons
```

Then the Person constructor could have a third parameter so that each construction of a Person object supplies the birth year, as in `sam = new Person ("Jorge", "Borges", 1899)`. A representation of this Person object is in Figure 4.2.

```
public Person (String first, String last, int year)
{  super();
   itsFirstName = first;
   itsLastName = last;
   itsBirthYear = year;
}  //=======================
```

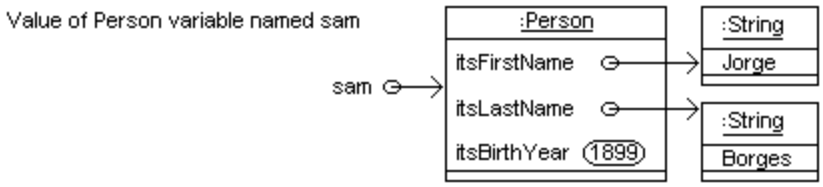


Figure 4.2 UML object diagram for a Person object

You would also want to add to the Person class a method to allow people to ask a Person object for its birth year with a method call such as `sam.getBirthYear()`:

```
public int getBirthYear()
{  return itsBirthYear;
}  //=======================
```

A Turtle object needs to keep track of its current position (conventionally measured from the top left corner of the frame), and its current heading (conventionally measured in degrees counterclockwise of a due east heading). So the instance variables might be declared and initialized as follows:

```
private int itsHeading = 0;  // due east
private int itsX = 380;  // in the center of 760 pixels wide
private int itsY = 300;  // in the center of 600 pixels tall
```

The Time class of objects

In some programming situations you work with clock times and you need to compute how many hours and minutes there are between two clock times. For instance, you may need to compute that 10:15 in the morning is 2 hours and 45 minutes after 7:30 in the morning. You may also need to solve problems such as finding out what clock time is 2 hours and 45 minutes after 7:30 in the morning. When you add in the complexities of accounting for afternoon versus morning, you can see that encapsulating these calculations in a class of Time objects will simplify things greatly.

Listing 4.5 defines a class of Time objects that can track the time of day, measured in hours and minutes since midnight. You can then have statements such as the following:

```
now = new Time (7, 30);  // 7:30 in the morning
wait = new Time (2, 45);  // 2 hours 45 minutes
later = now.add (wait);   // produces 10:15 in the morning
```

This class makes Time an **abstraction**, so that the development of logic that works with clock times is easier.

Listing 4.5 The Time class of objects and its driver (in two different files)

```
public class Time extends Object
{
   private int itsHour;
   private int itsMin;

   /** Create an object for the given hour and minute.  If min
    *  is negative, adjust the values to make 0 <= min < 60. */

   public Time (int hour, int min)            // constructor
   {  super();
      itsHour = hour;
      for (itsMin = min;  itsMin < 0;  itsMin = itsMin + 60)
         itsHour--;
   }  //======================

   /** Return the time expressed in military time. */

   public String toString()
   {  if (itsHour < 10)
         return ("0" + itsHour) + itsMin;
      else
         return ("" + itsHour) + itsMin;
   }  //======================

   public Time add (Time that)
   {}  // left as an exercise
}
//#############################################################



import javax.swing.JOptionPane;

public class TimeTester
{
   public static void main (String[] args)
   {  Time t1 = new Time (13, 25);
      Time t2 = new Time (8, -150);
      JOptionPane.showMessageDialog (null, "1 " + t1.toString());
      JOptionPane.showMessageDialog (null, "2 " + t2.toString());
      Time t3 = t1.add (t2);
      JOptionPane.showMessageDialog (null, "3 " + t3.toString());
      t1 = t2.add (t3);
      JOptionPane.showMessageDialog (null, "1 " + t1.toString());
      System.exit (0);
   }  //======================
}
```

The Time class has two int instance variables for the hours and the minutes, and the constructor is the natural one that initializes the two instance variables with the values of the two parameters. It makes a reasonable adjustment for a negative number of minutes. The `toString` method returns the time expressed as a String value indicating the usual military time. For instance, 14 hours and 15 minutes is expressed as "1415" but 7 hours and 30 minutes is expressed as "0730".

Time's `toString` method concatenates the "0" with the digits of `itsHour` to get a String value, then concatenates the result with the digits of `itsMin`. The order of the operations is important; `"0" + (itsHour + itsMin)` would have a quite different result, e.g., "042" if `itsHour` were 12 and `itsMin` were 30.

Driver programs

Before you use a class of objects in a program having several classes, you should test out the methods in that class separately. A **driver program** is an application program whose only purpose is to test the methods of a class thoroughly. The TimeTester class in the lower part of Listing 4.5 does this fairly well, by creating three Time objects and using both the `toString` method and the `add` method several times.

This driver program would be an even better test if it could accept input from the user that gives the hour and minute values for one or both of `t1` and `t2`. However, you do not as yet have any way to get a number from the user; `showInputDialog` only gets a string of characters, which is not at all the same thing. This situation will be rectified in the next section.

Programming Style Always put a space on each side of an operator (including + = <) and also directly after a comma or semicolon; it makes your program much easier to read. The space between a method name and its parentheses is optional -- some people prefer it and some do not.

Exercise 4.11 If x is 23 and y is 5, what are `"x" + (x + y)` and `("x" + x) + y`?
Exercise 4.12 If a Time object `t4` has 13 for `itsHour` and 5 for `itsMin`, what is `t4.toString()`? What should it be in military time?
Exercise 4.13 How would you revise Time's `toString` method to avoid the problem indicated by the preceding exercise?
Exercise 4.14 Write a Person method `public int getAge (int currentYear)`: The executor tells the current age of the Person, given the current year. But it returns zero if the current year is before the Person's birth year.
Exercise 4.15 Write the Time method `public Time add (Time that)`: The executor returns a new Time object that is the sum of the two, e.g., 0740 add 1430 is 2210. If the sum is more than 2359, drop the extra 24 hours, e.g., 1300 add 1400 is 300.
Exercise 4.16* Write a Time method `public int timeInMinutes()`: The executor tells the total number of minutes that have passed since midnight.
Exercise 4.17* Write a Time method `public Time subtract (Time that)`: The executor returns a new Time object that is itself minus the parameter, e.g., 0720 subtract 1430 is 1650. If the difference is negative, add an extra 24 hours.
Exercise 4.18* If you wanted Time objects to have a third attribute, the name of the day of the week, then (a) What instance variable declaration would you add? (b) What change would you make in the constructor? (c) What instance method would you have that tells the caller the object's day of the week? Write and compile the revised class.
Exercise 4.19* Revise the Time constructor to properly adjust for `itsMin` larger than 59, adding to `itsHour` as needed. Then repeatedly add or subtract 24 from `itsHour` until the value is in the range 0 to 23 (discard the excess; we do not care what day it is).
Exercise 4.20** Revise the Time constructor as stated in the preceding exercise, but do not have any looping statements anywhere in the constructor. Hint: Use the % operator.

4.6 Making Random Choices

We next develop a more interesting game than a BasicGame. The game object starts by picking a secret number at random in the range from 1 to 100, inclusive. It then asks the user to guess the number. If the user gets it right, of course the game is over. Otherwise the game object tells the user whether the guess was higher than the secret number or lower, and the game continues with more guessing.

Random integers

One thing this game needs is the ability to choose an integer value at random from a certain range of values. Fortunately, the Sun standard library has a **Random** class in the `java.util` package that provides objects that can do this. The phrase `new Random()` creates a random number generator that you may assign to a variable of Random type, called perhaps `randy`.

A Random object has an instance method named `nextInt` with a positive int value as the parameter. It returns an int value chosen at random in the range from 0 up to but not including that parameter value:

`Random randy = new Random()` creates a random-number generator.
`randy.nextInt(6)` returns one of 0, 1, 2, 3, 4, or 5, with equal likelihood.
`randy.nextInt(2)` returns either 0 or 1, with equal likelihood.
`randy.nextInt(100)` returns one of 0, 1, 2, ..., 99, with equal likelihood.

In general, `nextInt(n)` returns one of the first `n` non-negative int values, chosen with equal likelihood. These numbers are not truly random, since computers hardly ever do anything at random. But the sequence of numbers you get by repeated calls of the method is close enough for most purposes.

Since the number-guessing game wants one of the 100 int values in the range 1 to 100 inclusive, you need to call `randy.nextInt(100)` to get one of 100 different int values, then add 1 to the result to get one of the 100 different int values beginning with 1. If instead you needed a random int value in the range 20 to 30 inclusive, a total of 11 possibilities, you would call `randy.nextInt(11)` and then add 20 to the result.

The general principle is that the expression `min + randy.nextInt(num)` gives one of `num` consecutive int values with the smallest being `min`. If, however, you want a multiple of ten chosen at random from 30, 40, 50, ..., 150, you could use the expression `30 + 10 * randy.nextInt(13)`, which clearly gives one of the 13 possible multiples of 10. And if you want to get the result of rolling two dice, you could use the following:

```
int die1 = 1 + randy.nextInt (6);
int die2 = 1 + randy.nextInt (6);
JOptionPane.showMessageDialog (null,
      ("The 2 dice show " + die1) + die2);
```

The GuessNumber game

It will be quite convenient to have the GuessNumber class be a subclass of BasicGame. That means we can use whatever methods of the BasicGame class we want. For instance, the `playManyGames` and `playOneGame` methods can be used unchanged for a GuessNumber game. That is, we have the advantage of **reusable software** by making GuessNumber a subclass of BasicGame.

A GuessNumber object should apparently have two instance variables, perhaps named `itsSecretNumber` and `itsUsersNumber` (parallel to the instance variables in Listing 4.3). The `askUsersFirstChoice` method should begin by having a random number generator assign to `itsSecretNumber` a value from 1 to 100 inclusive. That means you should have the random number generator as an instance variable as well. This initialization of the secret number cannot be done in the constructor, since each time the game is played, the secret number must be re-initialized.

This logic is shown in the top part of Listing 4.6 (see next page). It has the import directive for JOptionPane but not for Random. You do not need to have an import directive if you explicitly name the package that Random is in every time you mention the word Random. This is the "fully qualified name" of the Random class. The alternative to what is shown is to have `import java.util.Random;` as a second line before the class heading and then simply use an unadorned Random in two places.

The `askUsersFirstChoice` method does the same as for any later choice, except that it first initializes `itsSecretNumber`. So it can call the `askUsersNextChoice` method to get the input. There is no reason to write the same coding twice.

The `shouldContinue` method returns `true` whenever `itsSecretNumber` is not equal to `itsUsersNumber`. The test of equality of numbers is always made with the operator `==` or `!=`. The `equals` method can only be used with objects, and int values are not objects.

The rest of Listing 4.6 should be clear except for `askUsersNextChoice`, which is discussed next. Figure 4.3 gives the UML class diagram.

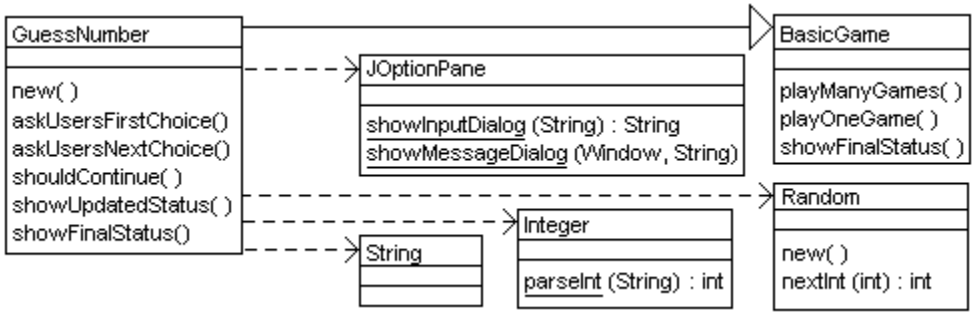


Figure 4.3 UML class diagram for GuessNumber

Caution A test for equality of Strings, as in the `shouldContinue` method of the earlier Listing 4.3, is always made with the `equals` method. Do not use `==` or `!=` to compare strings except in a comparison with `null`.

The use of the parseInt method for askUsersNextChoice

The `showInputDialog` method returns a String value (which can be `null` if the user clicked the Cancel button). A String value is not an int value, so you cannot assign it to the int variable `itsUsersNumber`. You have to convert the string of characters, which is a numeral, to a number. For instance, the number 88 can be added to or subtracted from other numbers; the numeral "88" is just keystrokes or marks on the screen. Some people like to remember the difference this way: Half of the number 88 is 44, but half of the numeral "88" is "8" (or even "oo" if you cut it in half horizontally rather than vertically). Or perhaps this way: Pro football players have big numerals on their jerseys and big numbers in their bank accounts.

Listing 4.6 The GuessNumber class

```
import javax.swing.JOptionPane;

public class GuessNumber extends BasicGame
{
   private java.util.Random randy;
   private int itsSecretNumber;
   private int itsUsersNumber;


   public GuessNumber()
   {  super();
      randy = new java.util.Random();
   }  //======================

   public void askUsersFirstChoice()
   {  itsSecretNumber = 1 + randy.nextInt (100);
      askUsersNextChoice();
   }  //======================

   public void askUsersNextChoice()
   {  String s = JOptionPane.showInputDialog
                    ("Guess my number from 1 to 100:");
      if (s != null && ! s.equals (""))
         itsUsersNumber = Integer.parseInt (s);
      else
         itsUsersNumber = -1;  // just to have a value there
   }  //======================

   public boolean shouldContinue()
   {  return itsUsersNumber != itsSecretNumber;
   }  //======================

   public void showUpdatedStatus()
   {  if (itsUsersNumber > itsSecretNumber)
         JOptionPane.showMessageDialog (null, "Too high");
      else
         JOptionPane.showMessageDialog (null, "Too low");
   }  //======================

   // inherited from BasicGame:
   //    playManyGames
   //    playOneGame
   //    showFinalStatus
}
```

Fortunately, the Sun standard library has a class named Integer that can help convert a numeral to a number. **Integer** (in the `java.lang` package) contains a class method:

```
Integer.parseInt (someString)
```

This method returns the int value you get when you interpret the keystrokes as digits of a number. If the user enters letters instead of digits or otherwise provides a badly-formed numeral, the program can crash. This is a violation of the robustness principle: All application programs should be written so that they cannot crash. But you may ignore this possibility until you see the advanced techniques in Chapter Six to avoid such crashes by checking that the String is a well-formed numeral. Until then, all user input that is supposed to be numeric will be assumed to be so, unless it is `null` or `""`.

Parse means to analyze a string of characters into its component parts. For instance, when you make sense of a Roman numeral, you are not "translating" it to an ordinary number, you are "parsing" it.

So the `askUsersNextChoice` method in Listing 4.6 gets the String value input using JOptionPane's class method and then has it parsed to an int value using Integer's class method. However, a `null` value or empty String returned by `showInputDialog` is taken as -1. Note that the order of the operands of `&&` is crucial; you cannot ask the `null` value if it equals the empty String. With the order given, if `s` is `null`, short-circuit evaluation will avoid testing the call of `equals`.

You have now seen all three varieties of object design:

1. Use existing objects and their methods (the Vic class in Listing 3.3).
2. Use existing objects but add new methods (the GuessNumber class in Listing 4.6).
3. Invent new objects to do the job (the Time class in Listing 4.5).

Language elements
You may use == or != between two object values (but rarely do so unless one is null).

Exercise 4.21 Write an expression that gives a random int value from -5 to 5, inclusive. Also write an expression that gives a random even number from 30 to 50, inclusive.
Exercise 4.22 Revise the GuessNumber game to have the secret number be in the range from 200 to 300, inclusive.
Exercise 4.23 Revise the GuessNumber game to tell the user "close enough; you win" when the guess is no more than 2 away from the right answer but not exactly right.
Exercise 4.24 What is the most number of guesses it would take for a really smart person to win this GuessNumber game? Explain your answer.
Exercise 4.25 Write an application program that asks the user for a sequence of int values and, when `showInputDialog` returns `null`, announces the smallest of them.
Exercise 4.26* Write an application program that asks the user for two int values and then announces whether one evenly divides the other (using %).
Exercise 4.27* Revise the GuessNumber game to say "you're hot" if within 2 of the right answer, "you're warm" if within 6 of the right answer, and "too high" or "too low" otherwise.
Exercise 4.28* Revise the GuessNumber game to have the program lie one-third of the time that the guess is too high; it says it is too low. But it never lies about the guess being too low, nor does it lie twice in a row.
Exercise 4.29* Find the best strategy for getting the right answer in the fewest guesses for the revision of GuessNumber described in the preceding exercise.

4.7 Overloading, Overriding, And Polymorphism

The **signature** of a method is its name followed by the types of its parameters in parentheses (leave out the return type and the names of its parameters). For instance, the signatures of the three JOptionPane methods described in Section 4.2 are as follows (a Component is a graphical object, and a Window is a Component):

```
showMessageDialog (Component, String)
showInputDialog (String)
showConfirmDialog (Component, String)
```

and the signatures of the methods of the Time class in Listing 4.5 are as follows:

```
Time (int, int)
toString()
Time add (Time)
```

Overloading of method names

Java lets you have several methods in the same class with the same name if they have different signatures. This is **overloading** of method names. This is quite common for constructors; you may have as many as you want as long as the compiler can distinguish them based on the parameter structure. For instance, you might want a second Time constructor that has a single int value as input, being the total number of minutes to be converted to hours plus minutes. So you could add to Listing 4.5 a constructor with the following heading:

```
public Time (int totalMinutes)
```

You might also want the Time class to have a second `toString` method, one with a String parameter telling which time zone to use, so it could have this heading:

```
public String toString (String timeZone)
```

In the email metaphor of Section 1.6, a Time object that gets the message with subject line `toString` first checks the body of the message. If it contains a String, that is an indication of the time zone. If the body of the message is blank, it returns military time.

Overriding of method definitions

A class may have the same method heading as a method in its superclass. For instance, both the BasicGame class and the GuessNumber class have a method with the heading `public boolean shouldContinue()`. The Java language rule in such a case is, if `x` is declared as a GuessNumber object somewhere, then `x.shouldContinue()` calls the method in GuessNumber, not the one inherited from BasicGame. The new definition of `shouldContinue()` **overrides** the original definition of `shouldContinue()` for GuessNumber objects, since it has the same signature (name and parameter pattern).

To illustrate the overriding of method definitions, we have just one more listing involving a subclass of Vic (the last in this book, because you are probably getting tired of Vics by now). This subclass is named BigVic. Each BigVic object keeps its own record of the number of filled slots and the number of empty slots it has. These are two int variables named `itsNumFilled` and `itsNumEmpty`. They are instance variables, because each instance (object) of the BigVic class has its own.

If a program says `sue = new BigVic()`, then `sue's` object has the two instance variables `itsNumFilled` and `itsNumEmpty` attached to it, in addition to whatever all Vic objects have. A call of `sue.getNumFilled()` has `sue` look up the value of `sue.itsNumFilled` and report it back. If the program also says `sam = new BigVic()`, then `sam`'s object has its own two variables which are completely distinct from `sue`'s.

When `sam` is a BigVic, a main method that calls `sam.putCD()` cannot simply execute the original `putCD()` method in the Vic class, because that would make `sam`'s counts wrong. So the BigVic class must have its own `putCD()` method that makes the two counts right. The method call `sam.putCD()` will execute the `putCD()` in the BigVic class, not the original one inherited from the Vic class. In other words, the new definition of `putCD()` overrides the original definition of `putCD()` for BigVic objects, since it has the same signature.

Caution You cannot have two methods defined in the same class with the same signature. If they have the same name, they must have a different parameter structure. It is not enough to be different in the part of the method heading that comes before the method name.

Calling a method that has been overridden

Part of the job of the new `putCD` method in BigVic is to execute the original `putCD` method in the Vic class. But if you write `putCD()` as one of the statements in the definition of the new `putCD` method, the executor would execute the new BigVic method, not the old Vic method, which would cause no end of difficulty.

Listing 4.7 (see next page) contains the BigVic class other than the constructor (left as an exercise). The method call `super.putCD()` tells the runtime system to execute the `putCD` method from the superclass, i.e., from the class BigVic extends. That will be the `putCD` method defined in the Vic class (since the intermediate Looper superclass does not override that definition of `putCD()`). Similarly, `super.takeCD()` inside BigVic's `takeCD` method will execute the `takeCD` method from the original Vic class. In either case, the BigVic object also corrects the counts for the number of filled slots and empty slots.

As you know, a method call inside a BigVic instance method has `this` as its executor if none is specified. So `super.takeCD()` actually means `this.super.takeCD()`. Similarly, if some other class declares `sam` as a BigVic object, then `sam.takeCD()` calls the BigVic method and `sam.super.takeCD()` calls the original Vic method.

Polymorphism

If you execute `sue = new BigVic()` and then execute `sue.fillSlots()`, that contains a call of `putCD()` (`fillSlots` is defined for Loopers in the earlier Listing 3.4, and every BigVic object is a Looper object as well). That call changes two counters, `sue.itsNumFilled` and `sue.itsNumEmpty`, in the BigVic object. But calls of `fillSlots()` with an ordinary Looper object do not change any counters. The command `putCD()` within the definition of `fillSlots` has different effects depending on the class of the object that is executing the command. So we say that the `putCD()` statement in Listing 3.4 is **polymorphic** (meaning it has more than one form).

Similarly, when a method executes `game = new GuessNumber()` and then executes `game.playOneGame()`, that causes a call of `shouldContinue()`. This is because `playOneGame` is defined for BasicGames in the earlier Listing 4.3, and every GuessNumber object is a BasicGame as well. That call tests `itsUsersNumber`. But calls of `shouldContinue()` with an ordinary BasicGame object test `itsUsersWord` instead. The command `shouldContinue()` within the definition of `playOneGame` has different effects depending on the class of the object that is executing the command. So we say that the `shouldContinue()` condition in Listing 4.3 is polymorphic. **Polymorphism** is the execution of polymorphic method calls.

Stickies

Think of polymorphism this way: Each object has a sticky-note on its side saying what class it belongs to. When the runtime system executes `this.shouldContinue()` for a GuessNumber object, it checks out the sticky-note on the side, sees that `this` is a GuessNumber object, and so executes the `shouldContinue` method defined in the GuessNumber class. If it found that the sticky-note said it was a BasicGame object, it would execute the `shouldContinue` method defined in the BasicGame class.

How does the sticky get there, you ask? The `super` call in the constructor puts the sticky on the object. The `super` call has to be the first statement in the constructor so that the rest of its statements can use `this` (implicitly or explicitly). (Note: In case you had not guessed by now, this is a metaphor. There really is no sticky in RAM, just an extra instance variable defined in the Object class and initialized by the `super` call.)

Listing 4.7 Four methods in the BigVic class

```
public class BigVic extends Looper
{
   private int itsNumFilled;     // count this's filled slots
   private int itsNumEmpty;  // count this's non-filled slots


   public BigVic()                        // constructor
   {  // left as an exercise
   }  //=====================

   /** Tell how many of the executor's slots are filled. */

   public int getNumFilled()
   {  return itsNumFilled;
   }  //=====================

   /** Tell how many of the executor's slots are empty. */

   public int getNumEmpty()
   {  return itsNumEmpty;
   }  //=====================

   /** Do the original putCD(), but also update the counters. */

   public void putCD()
   {  if ( ! seesCD() && stackHasCD())
      {  itsNumFilled++;
         itsNumEmpty--;
         super.putCD();
      }
   }  //=====================

   /** Do the original takeCD(), but also update the counters. */

   public void takeCD()
   {  if (seesCD())
      {  itsNumFilled--;
         itsNumEmpty++;
         super.takeCD();
      }
   }  //=====================
}
```

The equals method for Strings

The Object class defines a method named `equals`: `x.equals(y)` returns an answer of `true` or `false`, depending on whether `x` and `y` refer to exactly the same object. The String class overrides that definition with its own `equals` method: `x.equals(y)` for two String references returns `true` if the String objects contain exactly the same characters in the same order, even if they are two different String objects. In effect, it tests whether two boxes have the same contents, even if they are different boxes.

Caution It is almost always bad, and often uncompilable, to override a method in a superclass that does not have exactly the same method heading in all respects. For instance, Object's method has the heading `public boolean equals (Object o)`, so any `equals` method you define should also begin with `public boolean`.

Language elements
Two methods may have the same name if they are in different classes or have different signatures.
The following kinds of method call will call MethodName in the superclass of someObject:

```
someObject . super . MethodName ( )
someObject . super . MethodName ( ExpressionsSeparatedByCommas )
```

Exercise 4.30 Create a Liar subclass of the Person class in Listing 4.4: When you ask a Liar for its first name, half the time it says it is Darryl even if that is a lie.
Exercise 4.31 What changes would you make in Listing 4.7 to use an instance variable named `itsNumSlots` (the total number of slots) but not the instance variable `itsNumEmpty`, and still do the same things with the same method calls?
Exercise 4.32* Write out the signature of each method called in Listings 1.10 and 1.11.
Exercise 4.33* Write out the full Time constructor with the heading `public Time (int totalMinutes)`. Be sure to allow for negative inputs.
Exercise 4.34** Write out the BigVic constructor that Listing 4.7 needs.

4.8 The Rules Of Precedence For Operators

Sometimes you have to put parentheses around parts of an expression to have it mean what you want it to mean. For instance, as you learned in algebra class, `x + y * z` needs parentheses if you mean `(x + y) * z` but not if you mean `x + (y * z)`. This is because multiplication takes precedence over addition.

This section gives all the rules of precedence you need, if you choose to minimize the number of parentheses you use. This thorough discussion requires talking about a few things you will not see until the next two chapters. So just accept for now that it can be useful to put `(int)` or `(Time)` in front of an expression. You can use the name of any type of value in those parentheses, so the generic form is `(someType)`.

Accept also that you can put not only parentheses `( )` but also brackets `[ ]` around an expression to get something useful. Both of these are "grouping symbols." For this section only, we call a matched pair of grouping symbols plus their contents "groupers".

The **algebra rules of precedence** in Java are as follows:

1. For arithmetic expressions, multiply and divide operations are done first, then add and subtract. The % symbol counts as a divide operation. So `3+5 * 8` has the value 43, not 64.

2. The compiler works left-to-right in a sequence of multiply and divide operations. So `10 / 2*5` is 25, not 1.

3. The compiler also works left-to-right in a sequence of add and subtract operations. So `10 - 2+5` is 13, not 3. When one of the two added values is a String, the result is a string of characters, not a numeric value. So when you take into account the left-to-right evaluation, you see that the value of `3 + 4 + "x" + 3 + 4` is `"7x34"`.

4. You need to put parentheses around an expression formed with the addition, subtraction, multiplication, and division signs + - * / % if you negate it (as in `-(x + y)`) or you apply `(someType)` to it or you want to override the usual algebra rules of precedence.

The **general rules of precedence** in Java are as follows:

1. You never need to put parentheses around an expression that only involves words connected by dots and/or followed by groupers, `++` or `--`.

2. You never need to put parentheses around `! Whatever` or `- Whatever` if the latter is the negation operator (as in `y > -x`; we are not talking about subtraction here). You need to put parentheses just around the Whatever part if that part consists of an operator applied to two or more operands and the operator is not a dot or grouper.

3. You only need to put parentheses around `(someType) Whatever` when that expression is directly followed by a dot or grouper. You need to put parentheses just around the Whatever part if that part consists of an operator applied to two or more operands and the operator is not a dot or grouper.

4. You need to put parentheses around an expression formed with `||`, such as `Stuff || MoreStuff`, if you use `&&` to combine that or-expression with another value.

5. You should always put parentheses around an expression formed with `?:` except when it is to be assigned or returned (you will see `?:` expressions in Chapter Six).

6. You should also use parentheses liberally in some quite rare situations which are not used in this book: if you use shift or logical or bitwise operators, if you test for equality between two boolean expressions, or if you use the value of an assignment operation in a statement.

Caution You would not put a space in the middle of the word String or boolean and expect it to compile, would you? In Java, `++` and `<=` and `==` are all words, and the compiler will not like it if you put a space in the middle of one of those words.

1	2	3	4	5	6	7	8	9	10	11
( ) [ ] .	++ --	! (type) - +	* / %	+ -	< <= > >= instanceof	== !=	&&	||	?:	= += -= *= /= %=

Figure 4.4 Precedence of operators: The highest precedence is towards the left. The + and - at level 3 are not add and subtract; they apply to one operand.

Exercise 4.35 Neither of the following expressions will compile correctly. Add correcting parentheses to fix each one: (a) `! y + 2 < 3 && z.isTall()` (b) `(Boss) person.getJob()`.

Exercise 4.36 There are eight possible assignments of `true` and `false` to the boolean variables `b`, `c`, and `d`. Which assignments give `(b || c) && d` a different value from `b || (c && d)`?

Exercise 4.37* Same question as the preceding exercise, but for the two expressions `!b && (c || !d)` and `!(b || !c && d)`.

4.9 Analysis And Design Example: The Game Of Nim

Nim is a somewhat challenging game. The game starts with a pile of stones or other markers, perhaps 20 to 40 in all. The two players take turns removing 1 to 5 stones from the pile (although the upper limit might be set at some other small number, such as 3 or 4). The player who takes the last stone wins the game.

For instance, if the pile starts with 23 stones, the first player might take 4, leaving 19, and then the second player could take 3, leaving 16. If the first player then takes 2, leaving 14, the second player could take 4, leaving 10. Then the first player might take 1, leaving 9, and the second player could take 3, leaving 6. Then whatever the first player takes, the second player will be able to take the rest and thereby win the game. The accompanying design block is a reasonable plan for this program.

STRUCTURED NATURAL LANGUAGE DESIGN for Nim
1. Choose at random an initial number of stones for the pile, 20 to 40.
2. Choose at random the maximum number to take each turn, 3 to 5.
3. Ask the user how many he/she wants to take for the first turn.
4. Repeat the following until the pile has no more than the maximum one may take...
    4a. Choose the number the computer takes and tell the user.
    4b. Ask the user how many he/she wants to take for the next turn.
5. Tell the user who won and why.

Object design for Nim

The BasicGame logic applies here, so Nim should be a subclass of BasicGame. That is, BasicGame objects can be "retrained" to perform acceptably the tasks that this situation requires. The `playManyGames` and `playOneGame` logic can stand without change, so we have five methods to develop.

A single game seems to have just two attributes, the number of stones currently in the pile and the maximum number of stones one may take on each turn. Store this information in two instance variables named `itsNumLeft` and `itsMaxToTake`. The `askUsersFirstChoice` method should initialize these two values, the former as a random number in the range from 20 to 40 and the latter as a random number in the range from 3 to 5. This implies that a Nim object also needs a random number generator. The Nim object should keep the generator around to use in deciding its next move, so the generator should be an instance variable rather than a local variable of one method. Once the `askUsersFirstChoice` method has chosen the initial values, it asks for the user's next choice. These declarations are in the top part of Listing 4.8.

The askUsersFirstChoice method

Implementing `askUsersFirstChoice` is a bit more complicated than the other methods. The general idea should be to ask for input, repeating as needed until the user supplies a permissible choice, then subtract that number from `itsNumLeft`. When you rework this basic logic in detail, you could come up with the logic in the accompanying design block, for which the coding is in the middle part of Listing 4.8.

STRUCTURED NATURAL LANGUAGE DESIGN for askUsersNextChoice
1. Do the following until the `choice` is in the range from 1 to `maxToTake`...
    1a. Get input from the user using showInputDialog (so it is a String or else null).
    1b. If the input contains one or more characters then...
        1ba. Convert the string of characters received into an integer `choice`.
2. Subtract `choice` from `itsNumLeft` to get the new value of `itsNumLeft`.

Listing 4.8 The Nim class of objects

```
import javax.swing.JOptionPane;

public class Nim extends BasicGame
{
   private java.util.Random randy = new java.util.Random();
   private int itsNumLeft;
   private int itsMaxToTake;


   public void askUsersFirstChoice()
   {  itsNumLeft = 20 + randy.nextInt (21);
      itsMaxToTake = 3 + randy.nextInt (3);
      askUsersNextChoice();
   }  //======================

   public void askUsersNextChoice()
   {  int choice = 0;
      do
      {  String s = JOptionPane.showInputDialog
               (itsNumLeft + " left. Take 1 to " + itsMaxToTake);
         if (s != null && ! s.equals (""))
            choice = Integer.parseInt (s);
      }while (choice < 1 || choice > itsMaxToTake);
      itsNumLeft = itsNumLeft - choice;
   }  //======================

   public boolean shouldContinue()
   {  return itsNumLeft > itsMaxToTake;
   }  //======================

   public void showFinalStatus()
   {  if (itsNumLeft == 0)
         super.showFinalStatus();
      else
         JOptionPane.showMessageDialog (null, "I take "
                + itsNumLeft + " and so I win.");
   }  //======================

   public void showUpdatedStatus()
   {  int move = itsNumLeft % (itsMaxToTake + 1);
      if (move == 0)
         move = 1 + randy.nextInt (itsMaxToTake);
      itsNumLeft = itsNumLeft - move;
      JOptionPane.showMessageDialog (null, "I take " + move
                + ", leaving " + itsNumLeft);
   }  //======================
}
```

The shouldContinue and showFinalStatus methods

The `shouldContinue` method is the easiest to develop next: If there are no stones left, the game is over and the human player wins. Otherwise, if the computer opponent can take all remaining stones and win, it should do so, which also makes the game over. Otherwise the game should continue. You may conclude, therefore, that the game should continue if and only if `itsNumLeft` is greater than `itsMaxToTake`.

The logic for the `showFinalStatus` method is implicit in the preceding analysis. It is basically an if-statement: If there are no stones left, the program should announce that the user wins. This can be done by using `super` to call on the `showFinalStatus` method in the BasicGame superclass. But if there are a few stones left, it will not be more than `itsMaxToTake`, so the computer opponent announces that it takes the remaining stones and wins. See the middle part of Listing 4.8 for the coding of these two methods.

The showUpdatedStatus method

The hard part of the `showUpdatedStatus` method is to decide what move the computer player is to make. This is a problem of strategy rather than a programming problem. Extensive thought leads to the conclusion that the computer opponent can always win if it leaves a multiple of six stones each time (assuming the maximum allowed is five; the general formula is `itsMaxToTake+1`). So the computer's move will always be to calculate the remainder left after `itsNumLeft` is divided by `itsMaxToTake+1` and take that many stones.

However, the human player might have left a multiple of six stones (or whatever `itsMaxToTake+1` is). In that case, the computer opponent must choose some number of stones and hope that, on the next move, the human player does not leave a multiple of six (or whatever). A random choice from 1 to `itsMaxToTake` is appropriate here. The program then subtracts that many stones from `itsNumLeft` and announces the result.

Note how easily the preceding two paragraphs lead to the correct coding for `showUpdatedStatus`, as shown in the lower part of Listing 4.8. The most important principle for developing the correct logic for a method is to write it out in English first (or whatever natural language you are most comfortable with), normally with a structured organization. Read it over, make sure there are no errors in it, then write it down in Java.

Debugging The primary information you need for debugging a program is the intermediate values of key variables. Each variable in the Nim class is printed right after it is assigned a value with one exception. You can get that information by temporarily inserting the following as the last statement of `askUsersNextChoice`:

```
System.out.println (itsNumLeft + "=num; choice=" + choice);
```

Note on terminology If X extends Y, we often say that X "is derived" from Y, that X is a "child" of Y, and that Y is a "parent" of X. Since BasicGame is the parent class of Nim and GuessNumber, that makes those two "sibling" classes.

Exercise 4.38 Revise Listing 4.8 so that an illegal choice by the user is changed to be a guess of 1. Be sure to inform the user of this adjustment each time it happens.
Exercise 4.39 Revise Listing 4.8 so that the user gets to decide who goes first, once informed of `itsNumLeft` and `itsMaxToTake`.
Exercise 4.40* What happens if `itsMaxToTake` is 5, the human player leaves 6, the computer takes 4, and then the human player tries to take 3? Fix the inconsistency.
Exercise 4.41* Revise Listing 4.8 so that the rule is that a player can take up to half of the remaining stones, but always at least 1, and the size of the initial pile of stones is from 50 to 100. Have the computer take about a third of the remaining stones each time.
Exercise 4.42** For the game rules of the preceding exercise, give the computer a strategy that guarantees it wins if the human player does not make the one right choice every time.
Exercise 4.43** Revise Listing 4.8 so that the `showFinalStatus` method announces how many games so far the human player has won and how many the program has won.

4.10 Analysis And Design Example: The Game Of Mastermind

Suppose you decide to write a program to play the game of Mastermind. The program is to choose a three-digit whole number (000 to 999) and the human player is to guess the number. Each time the user announces a guess, the program is to tell whether all three digits are right. If the user's choice is not completely right, the program is to tell how many digits of the guess were in the right position. It is also to tell how many other digits of the guess were in the wrong position in the number. Then it has the user guess again.

Analysis and logic design

To make sure you have the concept right, you try it out with some sample data, e.g., if the program's chosen number is 123 and you guess 325, the program is to say you have one digit in the right place (the 2) and one digit in the wrong place (the 3). You realize that you are not sure how to count the number wrong when duplicate digits are involved, e.g., how many digits are wrong in the guess 225 if the program's chosen number is 123? So you check with your client and find out that the first 2 is considered to be in the wrong position. That is, the game should announce 1 in the right position and 1 in the wrong position. After several more trials, you are ready to work out the logic design. A reasonable plan is shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for Mastermind

1. Choose a secret three-digit whole number.
2. Ask the user for his/her first guess.
3. Repeat the following until the current guess is completely right...
    3a. Tell the user how many digits are right and how many are wrong.
    3b. Ask the user for his/her next guess.
4. Tell the user he/she has finally gotten it right.

Object design and initialization

This logic has the same structure as the BasicGame logic, so you can make Mastermind a subclass of BasicGame. The inherited `playManyGames`, `playOneGame`, and `showFinalStatus` methods can be used unchanged.

The construction `game = new Mastermind()` should create the game-playing object and `askUsersFirstChoice` should choose a secret three-digit number at random. Since you will have to compare individual digits of the guess, it is probably more convenient to pick three separate one-digit numbers at random. So you could have the following initializations of instance variables (the digits are in the range 0 to 9 inclusive):

```
randy = new Random();
itsFirst = randy.nextInt (10);
itsSec   = randy.nextInt (10);
itsThird = randy.nextInt (10);
```

The user's input will be a three-digit integer. It will be easiest to store the three digits in three separate variables, so you can compare them with the three digits of the secret number. Since one instance method gets these three values and another instance method looks at them to see if they match, you must store the three digits of the user's guess in instance variables. These three variables could be named `usersFirst`, `usersSec`, and `usersThird`. The upper part of Listing 4.9 declares the instance variables, the constructor, and the rather obvious `askUsersFirstChoice` method (which has most of its work done by the `askUsersNextChoice` method).

Listing 4.9 The Mastermind class of objects, one method postponed

```
import javax.swing.JOptionPane;
import java.util.Random;

public class Mastermind extends BasicGame
{
   private Random randy;
   private int itsFirst;    // 100's digit of secret number
   private int itsSec;      //  10's digit of secret number
   private int itsThird;    //   1's digit of secret number
   private int usersFirst;  // 100's digit of user's guess
   private int usersSec;    //  10's digit of user's guess
   private int usersThird;  //   1's digit of user's guess


   public Mastermind()
   {  super();
      randy = new Random();
   }  //======================

   public void askUsersFirstChoice()
   {  itsFirst = randy.nextInt (10);
      itsSec   = randy.nextInt (10);
      itsThird = randy.nextInt (10);
      askUsersNextChoice();
   }  //======================

   public void askUsersNextChoice()
   {  String s;
      do
      {  s = JOptionPane.showInputDialog
                     ("Enter a three-digit integer:");
      }while (s == null || s.equals (""));
      int guess = Integer.parseInt (s);
      usersThird = guess % 10;
      usersFirst = guess / 100;
      usersSec = (guess / 10) % 10;
   }  //======================

   public boolean shouldContinue()
   {  return usersFirst != itsFirst || usersSec != itsSec
                                    || usersThird != itsThird;
   }  //======================
}
```

The askUsersNextChoice method

First you have to get a string of characters as input and convert it to an integer value stored in `guess` (except if the input string has no characters, you get another input). Now you need to separate out the digits. Taking the remainder from dividing `guess` by 10 clearly gives the last digit of `guess`. Taking the quotient from dividing `guess` by 100 clearly gives the first digit of guess (assuming the user has not made a mistake and entered more than 999). But how do you get the middle digit?

If you take the quotient from dividing `guess` by 10, you suppress the last digit, so the last digit of that quotient must be the next-to-last digit of the original `guess`. In other words, the computation `(guess / 10) % 10` gives the middle digit of `guess`. The coding for this `askUsersNextChoice` method is in the middle part of Listing 4.9.

The shouldContinue method

If each one of the user's digits matches the corresponding secret digit, the game is over. So if any one of the three user's digits is not the same as the corresponding secret digit, the game should continue. That gives the obvious coding in the bottom of Listing 4.9.

The showUpdatedStatus method

The `showUpdatedStatus` method has to calculate the number of user's digits in the right position and the number of user's digits that are in the wrong position. You can simply go through the user's digits one at a time and for each one, count it as right if it matches the corresponding secret digit, otherwise see if it matches either of the other two secret digits (in which case it counts as being in the wrong position).

The easiest way is to initialize two counters `numRight` and `numWrong` to 0 and increment the appropriate one when you see that a particular user's digit matches some secret digit. Then print a message telling the user the result. The coding is in Listing 4.10.

Listing 4.10 The Mastermind class of objects, part 2

```
   public void showUpdatedStatus()     // in Mastermind
   {  numRight = 0;
      numWrong = 0;

      if (usersFirst == itsFirst)
         numRight++;
      else if (usersFirst == itsSec || usersFirst == itsThird)
         numWrong++;

      if (usersSec == itsSec)
         numRight++;
      else if (usersSec == itsFirst || usersSec == itsThird)
         numWrong++;

      if (usersThird == itsThird)
         numRight++;
      else if (usersThird == itsSec || usersThird == itsFirst)
         numWrong++;

      JOptionPane.showMessageDialog (null, " You have "
            + numRight + " in the right place and "
            + numWrong + " in the wrong place.");
   }  //=====================
```

Exercise 4.44 Under what circumstances can the user get one or more digits right if the user chooses a negative number for the guess? Which digits are they?
Exercise 4.45 Explain what happens when the user chooses a positive four-digit or longer number for the guess.
Exercise 4.46** Revise the Mastermind program so that, after each game played, it tells the user the number of games played to date and the average number of guesses required to get the right answer.
Exercise 4.47** Rewrite the Mastermind program so that `askUsersNextChoice` calculates `numRight` and `numWrong`. Have `shouldContinue` and `showUpdatedStatus` use those calculated values.

4.11 Using BlueJ With Its Debugger

You can download for free the BlueJ IDE (this acronym stands for "Integrated Development Environment"). BlueJ is an environment in which you can compile, execute, and analyze Java classes. It uses true Java, since you have to first install a recent official version of Java (from e.g. java.sun.com/products) in order to use BlueJ. The primary purposes of the BlueJ environment are to help you debug your programs and to let you manipulate objects one step at a time (execute a single method, see the result, execute another method, etc.).

Go to the URL www.bluej.org and click on the icon for downloading the latest version. Store it in a folder named bluej. You will then have a file with a name something like bluej-115.jar. Open the bluej folder and click on that jar file. The installer will ask you for the name of the folder where your current version of Java JDK is stored, which you should supply (e.g., jdk1.3.1_01). You can then enter the BlueJ environment by clicking on the icon specified in the installation instructions (probably bluej or bluej.bat).

Checking out an example

Click on the Project option, then click Open Project. Double-click on the examples sub-folder in the bluej folder. Choose one of the examples listed there to try out. We illustrate the process with the Hello example:

- Click on the Hello sub-folder of examples and then click `open`. This loads the Hello project into BlueJ. You will see a class box named Hello in a central **workarea**. It is cross-hatched to indicate that it is not yet compiled.
- Click on Compile. This compiles all the classes showing in the workarea. The cross-hatching disappears from the Hello class box to indicate this.
- Right-click the Hello class box. You will see a list of the methods in the Hello class that can be called without using an instance of the Hello class. There are but two: a constructor `new Hello()` and `public static void main(String[] args)`.
- Click on the main method. It allows you to enter the `args` parameter value, which you may ignore (we rarely use command-line arguments). Then click OK to execute the main method. A terminal window pops up with the output "Hello world", which is all this main method produces.
- Read the source code for the Hello class, which you can view by clicking on the "Open Editor" option within the Hello class box. The source code is shown below.
- Close the editor and click on the constructor `new Hello()`. You are asked for the name of the Hello object you are constructing; enter `sam` and click OK. You will see an object box (rounded corners, "sam:" at the top) to represent this Hello object.
- Right-click on the object box to see a list of the instance methods you can call for this Hello object. Click on the only one shown, namely, `public void go()`. That will execute `sam.go()`, which causes a second "Hello, world" to appear in the terminal window.

```
class Hello
{
    public void go()
    {   System.out.println("Hello, world");
    }

    public static void main(String[] args)
    {   Hello hi = new Hello();
        hi.go();
    }
}
```

In general, you can execute any class method or constructor by right-clicking on a class box and then clicking the method name. You can execute any instance method by right-clicking on an object box (which represents an instance of the class) and then clicking the method name. You can also inspect the current values of all class variables (described in Chapter Five) and instance variables by clicking on the Inspect option for an object box. Figure 4.5 shows what the BlueJ display looks like at this point.

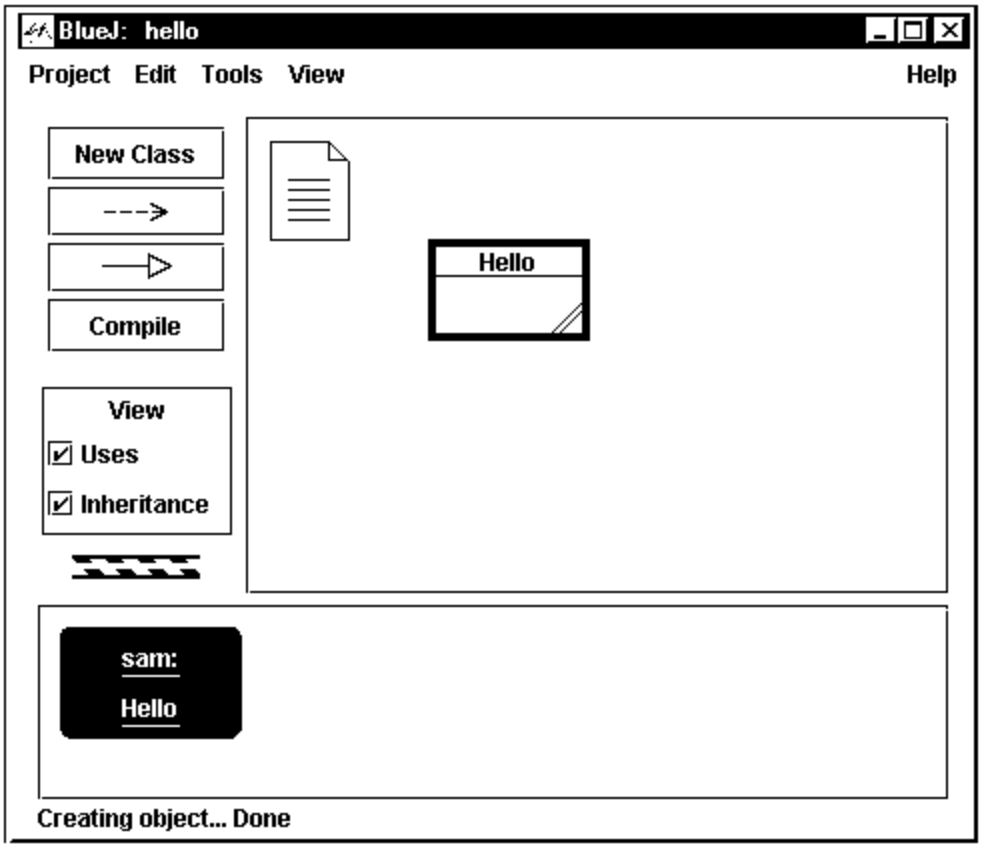


Figure 4.5 The BlueJ display for the Hello class with an instance of Hello

Adding a new class to the project

Click New Class and give it the name Adder. A class box with that name will appear in the workarea. Right-click it and choose Open Editor to create code for it. You will see a minimal coding for the class, with an instance variable `x` and a method named `sampleMethod`. Replace the body of that method to have the following coding:

```
public int sampleMethod (int y)
{   x = x + y;
    return x;
}
```

Close the editor window, compile the Adder class (click Compile), then construct an Adder object named `adder_1` (right-click the Adder class box and click on `new Adder()`). Right-click on `adder_1` and then click Inspect to see that the current value of the instance variable `x` is 0. Figure 4.6 shows what the display looks like now.

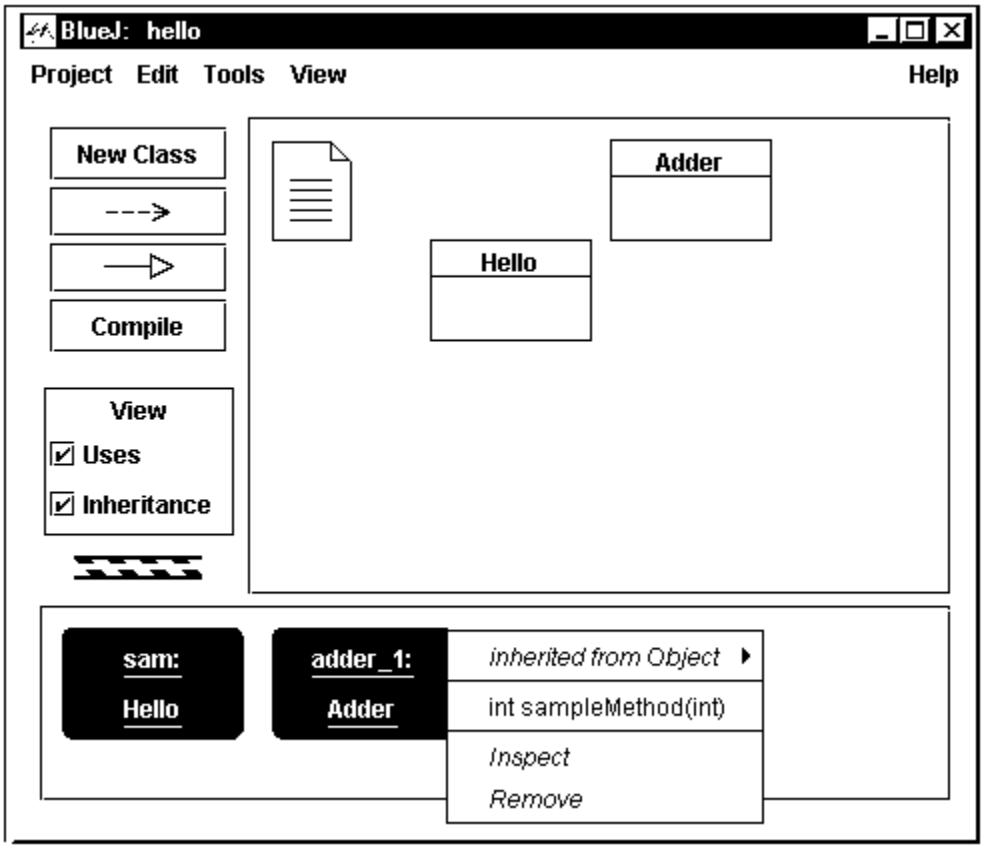


Figure 4.6 The BlueJ display with the Adder class and an Adder instance

Right-click on `adder_1` and then click `int sampleMethod(int y)` to execute that method. You have to enter the value of the parameter; type 5 and then click OK. BlueJ will then tell you that the result of the method call `sampleMethod(5)` is 5.

Repeat with parameter value 3; BlueJ will tell that the result of the method call `sampleMethod(3)` is 8. Inspect `adder_1` to see that the instance variable `x` now has the value 8. Make sure you understand why these things happen before you go on. Play around with it all a bit more.

Using BlueJ with existing classes

Put the following classes from this chapter in a folder named `games`: GameApp, BasicGame, and GuessNumber. Click on the Project menu choice and select Open Non-BlueJ Project. Go to the folder that contains the games folder, click on `games`, then click on Open in BlueJ. All three classes will appear in the workarea. Click on Compile to compile them all.

You have created a BlueJ project unit to manage these three classes, so you can now create instances of the class and try them out. For instance, once you create an instance of GuessNumber, you can right-click on it, choose `void playOneGame()` from the list of methods inherited from BasicGame, and thereby play a game. During the game, right-click the GuessNumber object from time to time and click Inspect so that you can see the current values of the instance variables.

Debugging

The basic debugging process is to set a **breakpoint** in your coding where you want to look closely at what is happening. You do this by choosing Open Editor for some class and then clicking in the far left column, next to some statement in the coding. A stop sign icon should appear there. That indicates the breakpoint you have set.

Now if you execute the coding, execution pauses when it comes to that stop sign (just before executing that marked statement) and a debugger window appears. This window shows the current values of all variables local to the method. You may also inspect the current state of all the objects involved in the execution.

You may click the Step option to advance the execution by one statement at a time. Inspect the values of variables at each point of interest. When you have seen all you want to see, click on the stop sign to remove it, then click Continue in the debugger window. Execution will then continue normally.

Figure 4.7 shows how the debugger window looks. You may also bring up the debugger window on a program while it is running by clicking on the turning barber-pole icon. The Call Sequence window shows what method calls are currently active. In this example, you are currently executing the `askUsersNextChoice` method, which was called from the `askUsersFirstChoice` method, which was called from the `playOneGame` method, which was called by clicking on the method call in the GuessNumber object box.

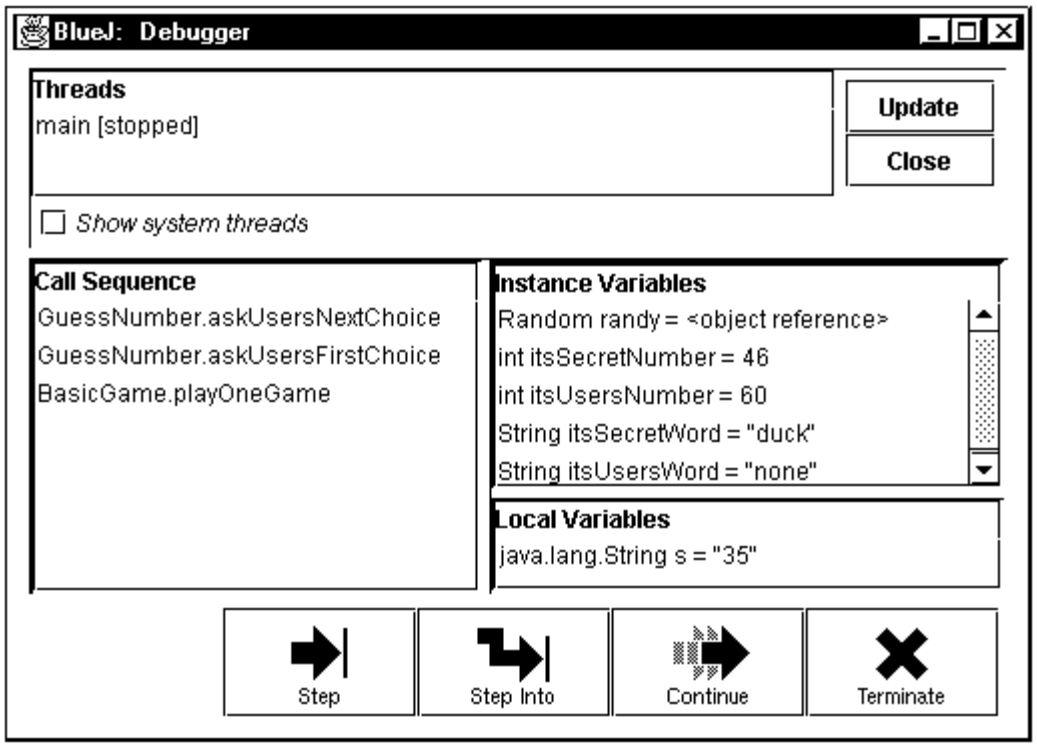


Figure 4.7 A debugger window for BlueJ

This has been a very short introduction to how to use BlueJ. You will understand it much better if you read the complete 30-page tutorial available at www.bluej.org (click on Documentation) and try it out with the example programs and with your own programs.

4.12 Review Of Chapter Four

Listing 4.3, Listing 4.5, and Listing 4.7 illustrate almost all Java language features introduced in this chapter. This review includes the preceding Interlude.

About the Java language:

- `package X;` as the top line of a file makes the class in that file in **package** X.
- The **import directive** `import javax.swing.JOptionPane` goes in a compilable file before any class definitions if you use JOptionPane in some class in the file. Similarly, use `import java.util.Random` if you use Random. However, you may instead give the full name (`java.util.Random` or `javax.swing.JOptionPane`) at each mention of the class.
- If you want to import several classes from the same package, e.g., from `java.util`, use the import directive `import java.util.*;`. Classes from the `java.lang` package (e.g. System, String, Object, Integer) do not need a directive.
- You may assign the value `null` to any object variable. It signals the absence of any actual object reference stored in the variable.
- The result of evaluating a plus sign between two strings of characters or between a string and a number is the first string of characters followed directly by the second. If one is a number, it is converted to a decimal numeral (a string of characters) before the **concatenation**. The **empty String** `""` is the String value whose length is zero.
- If the **newline** character `\n` is within the quotes of a **String literal**, it causes the start of a new line.
- You may declare a variable in a class outside of any method. Then each object of the class has its own value for the variable if it is an **instance variable** (which, as you will see in the next chapter, requires that it not be declared using `static`).
- If one class extends another, then each public method and public variable of the superclass is indirectly in the subclass. If a class does not explicitly extend any class, then it makes the **default extension** of the **Object** class.
- Within the body of an instance method or constructor, the use of an instance variable without saying which object it belongs to signals that it belongs to the executor of the instance method or to the object being constructed, respectively. The use of `this` inside a constructor is a reference to the object that is being constructed.
- A class must have one or more **constructors**, to say what happens when an instance of the class is created. If you do not explicitly define a constructor, one will be provided for you by the compiler. This **default constructor** has no parameters and no commands other than `super()`.
- If the declaration of an instance variable assigns it a value, that takes effect when the constructor is called. The runtime system supplies a **default initial value** if you do not: zero or `null` or `false`, as appropriate.
- For any **int** variable `x`, `x++` **increments** `x` and `x--` **decrements** `x`.
- The six comparison operators are `>` `<` `>=` `<=` `==` `!=` .
- The five operators for int values are +, -, *, /, and % (for remainder).
- The **signature** of a method is its name followed by parentheses containing the types of its parameters (not their names). Different methods can have the same name if they have a different parameter pattern (signature) or are in different classes. This is **overloading of a method name** if they are in the same class.
- If a subclass defines an instance method with the same signature as an instance method in its superclass, that is **overriding of a method definition**. Suppose the signature is `doStuff(int)` and `sam` refers to an object of the subclass. Then `sam.doStuff(3)` calls the subclass method and `sam.`**`super`**`.doStuff(3)` calls the superclass method.
- One of the two **rules of precedence** people get wrong most is that one should put parentheses around the whole `(someType) Whatever` expression when followed

by a bracket [or dot. The other one is that ! takes precedence over && which in turn takes precedence over ||. See Section 4.8.

- Figure 4.8 describes the remaining new language features. The Initializer part of a for-statement can assign a value to the **loop control variable** (the only variable mentioned in the Condition of the for-statement whose value changes during execution of the loop). The Initializer part may be the declaration and assignment of the loop control variable if that variable is not used outside of the for-statement.

`public ClassName (`*`ParameterList`*`)`
`{ StatementGroup`
`}`
	declaration of constructor method. The parameters are optional. The first statement should be super(...)

`super (`*`ArgumentList`*`);`	**statement** calling the superclass's constructor. Only allowed as the first statement in the constructor

`for (`*`Initializer`*` ; Condition ;`
`Update)`
`Statement`
	statement that executes the Initializer part first, then repeats test-Condition-do-Statement-do-Update, quitting when the Condition is false

`do`
`{ Statement`
`}while (Condition);`
	statement that repeats do-Statement-test-Condition, quitting when the Condition is false

`private Type VariableName` `= Expression;`	**declaration** of instance variable outside any method. Initializing is optional
`super.MethodName (`*`ArgumentList`*`);`	**statement** that calls the method of that name in the superclass

Figure 4.8 Declarations and statements added in Chapter Four

About some key Sun standard library methods:

- `System.exit(0)` terminates an execution of a program immediately. A program that has a graphical user interface should explicitly execute this statement when it is done, because otherwise some computer systems lock up. The 0 in the parentheses indicates a normal termination.
- `System.out.println(someString)` displays the value in the parentheses in the terminal window. The argument may also be a numeric value.
- `someObject.equals(otherObject)` tests whether the two objects are identical.
- `JOptionPane.showMessageDialog(null, messageString)` displays the `messageString` on the screen until the user clicks OK or closes the window.
- `JOptionPane.showInputDialog(promptString)` displays the `promptString` and waits for input. It returns the String input, except it returns `null` if the user clicks the Cancel button.
- `JOptionPane.showConfirmDialog(null, messageString)` displays the `messageString` on the screen with three buttons labeled Yes, No, and Cancel, until the user clicks one of the buttons or closes the window. It returns the int value `JOptionPane.YES_OPTION` if the user clicked the Yes button.
- `new Random()` constructs a Random object.
- `someRandom.nextInt(limitInt)` returns an int value chosen randomly with equal likelihood from zero up to but not including `limitInt`.
- `Integer.parseInt(someString)` returns the int value that the String value represents, assuming that it is not badly-formed with letters or other defects.
- General note: In these review sections, we normally specify Sun standard library methods as **generic method calls**, so you see how they are called: The executor of an instance method is named "`some`" followed by the executor's class (e.g., `someRandom`), and the parameters are specified the same way (e.g., `someString`) except that parameters may be numeric or boolean or the like (e.g. `someInt`) and "`some`" may be replaced by a more descriptive word (e.g., `limitInt`).

Other vocabulary to remember:

- A **natural constructor** is a constructor that assigns the values of its parameters to all or almost all of the instance variables of the class.
- **Encapsulation** means primarily preventing outside classes from changing instance variables directly (i.e., without calling a method in the class). It includes declaring instance variables as `private`. The purpose is **abstraction**: sending a message to an object to have a complex task done rather than doing the task directly.
- A local variable or parameter **shadows** an instance variable when they have the same name. The name refers to the local variable instead of to the instance variable.
- A **binary operator** is a symbol that combines two values to obtain a new value. `&&` and `+` are binary operators. But `!` is a **unary operator**: You apply it to only one value to obtain a new value.
- Software is **reusable** if it is designed to be used in several different programming situations. A **driver program** is an application program whose only purpose is to test the methods of a class thoroughly.
- A method call is **polymorphic** if it calls an instance method and the executor could be from any of two or more classes that have differing definitions of the method being called. **Polymorphism** is the execution of polymorphic method calls.

Answers to Selected Exercises

4.3
```
public boolean shouldContinue()
{   if (itsSecretWord.equals ("goose") || itsUsersWord.equals ("duck"))
        return itsSecretWord.equals (itsUsersWord);
    itsSecretWord = "goose";
    return false;
}
```
4.4 Add this declaration outside of any method: `String itsMark = "";`
Add this statement early in askUsersFirstChoice: `itsMark = "x";`
Add this statement to askUsersNextChoice: `itsMark = itsMark + "x";`
Add this statement to showFinalStatus: `itsMark = "";`
Replace the statement in the body of shouldContinue by this statement:
`return ! (itsSecretWord.equals (itsUsersWord) || counter.equals ("xxxxx"));`

4.7
```
public SmartTurtle()
{   super();
}
```
4.8
```
public RedTurtle()
{   super();
    switchTo (RED);
    move (-180, 0);
}
```
4.9
```
public class NamedTurtle extends Turtle
{   private String itsName;
    public NamedTurtle (String name)
    {   super();
        itsName = name;
    }
    public String getName()
    {   return itsName;
    }
}
```
4.11 "x" + (x + y) is "x28" and ("x" + x) + y is "x235".

4.12 t4.toString() is "135", but it should be "1305".

4.13 Insert the following at the beginning of the toString() method body:
```
if (itsMin < 10) {  if (itsHour < 10) return ("0" + itsHour) + "0" + itsMin;
                    else return ("" + itsHour) + "0" + itsMin;  }
else
```
4.14
```
public int getAge (int currentYear)
{   if (currentYear < itsBirthYear)
        return 0;
    else
        return currentYear - itsBirthYear;
}
```
4.15
```
public Time add (Time that)
{   Time valueToReturn = new Time (this.itsHour + that.itsHour, this.itsMin + that.itsMin);
    if (valueToReturn.itsMin >= 60)
    {   valueToReturn.itsMin = valueToReturn.itsMin - 60;
```

```
            valueToReturn.itsHour++;
        }
        valueToReturn.itsHour = valueToReturn.itsHour % 24;
        return valueToReturn;
    }
```

4.21 -5 + randy.nextInt (11) and 2 * (15 + randy.nextInt (11)), assuming Random randy.

4.22 Replace the last statement of the constructor by:

```
itsSecretNumber = 200 + randy.nextInt (101);
```

Replace the string literal in askUsersChoice by: "Guess my number from 200 to 300:".

4.23 Replace the statement in the body of shouldContinue by:

```
return itsUsersNumber < itsSecretNumber - 2  ||  itsUsersNumber > itsSecretNumber + 2;
```

Add a showFinalStatus method to GuessNumber with this statement in its body:

```
if (itsUsersNumber != itsSecretNumber)
    JOptionPane.showMessageDialog (null, "close enough; you win");
else
    JOptionPane.showMessageDialog (null, "That was right.  \nCongratulations.");
```

4.24 Guess 50. If you are wrong, you know there are at most 50 possible right answers.
Guess 25 or 75, depending on whether the answer was "too high" or "too low".
If you are wrong, you know there are at most 25 possible right answers. Continue guessing in the middle of the range of possible right answers, reducing the number of possible right answers to 12, 6, 3, and 1 in that order. The seventh guess will then get it right.

4.25

```
public class FindSmallest
{   public static void main (String [ ] args)
    {   JOptionPane.showMessageDialog (null, "Finding the smallest of a group of integers");
        String prompt = "Enter an integer.  Click Cancel when done.";
        String s = JOptionPane.showInputDialog (prompt);
        if (s == null)
            System.exit (0);
        int smallest = Integer.parseInt (s);
        s = JOptionPane.showInputDialog (prompt);
        while (s != null)
        {   int input = Integer.parseInt (s);
            if (smallest > input)
                smallest = input;
            s = JOptionPane.showInputDialog (prompt);
        }
        JOptionPane.showMessageDialog (null, "The smallest was " + smallest);
        System.exit (0);
    }
}
```

4.30

```
public class Liar extends Person
{   public Liar (String first, String last)
    {   super (first, last);
    }
    public String getFirstName()
    {   if (new java.util.Random().nextInt (2) == 1)
            return "Darryl";
        else
            return super.getFirstName();  // you cannot mention itsFirstName here
    }
}
```

4.31 Replace itsNumEmpty by itsNumSlots in the declaration of the instance variables.
The statement in the body of getNumEmpty should be: return itsNumSlots - itsNumFilled.
Omit the statements that mention itsNumEmpty in putCD and takeCD.

4.35

```
! (y + 2 < 3) && z.isTall()  OR  ! (y + 2 < 3 && z.isTall())
((Boss) person).getJob()
```

4.36 In the two cases when b is true and d is false and c is either true or false,
(b || c) && d is false but b || (c && d) is true. In the other six cases they match.

4.38 Delete "do {" and the line beginning "}while". Put the following just before the last statement:

```
if (choice < 1 || choice > itsMaxToTake)
{   choice = 1;
    JOptionPane.showMessageDialog ("Illegal; changed to 1.");
}
```

4.39 Replace the last statement of askUsersFirstChoice by the following:

```
if (JOptionPane.showConfirmDialog (null, "Taking up to " + itsMaxToTake + " and starting with"
            + itsNumLeft + ".  Will you go first?") == JOptionPane.YES_OPTION)
    askUsersNextChoice();
```

4.44 A negative guess will still get a digit right if the guess's digit is 0 and the secret digit is also 0.
This can happen at any or all of the three digits.

4.45 usersFirst will be wrong, since it will be 10 or more. The other two digits will be right or wrong according to whether they would have been without the extra digits.

5 Class Methods and Class Variables

Overview

This chapter shows you how to define and use methods and variables that do not need executors, such as the `Vic.say` method introduced in Chapter Two:

- Sections 5.1-5.3 describe and illustrate class methods, class variables, and final variables.
- Sections 5.4 and 5.5 develop a full implementation of a Vic simulator, which completes the top-down development of Vic software (first see how to use the methods, then learn how to define them). It uses two new String methods, `substring` and `length`. If you want to move on quickly to the material in later chapters, you may postpone or skip everything in Chapter Five after Section 5.4.
- Sections 5.6-5.8 describe software for working with Networks. The Network software illustrates the use of for-statements, class methods, class variables, and final variables. It does not involve any new language features.
- Section 5.9 explains and illustrates the use of recursion. The first half of the section is independent of Networks.

5.1 Defining Class Methods

A method to find an integer average of several integer values would be quite useful. If the `sum` of a group of 5 numbers is 49, the average is best approximated as 10, but if their `sum` is 46, the average is best approximated as 9. You cannot calculate this average as `sum / count`, because you would get 9 in either case; division of int values drops the fractional part. You could simply add half of the count before you divide and then have the method return that result:

```
return (sum + count / 2) / count;
```

This works for positive numbers. But if you try this when the `sum` is negative, it gives the wrong answer: `-49 + 5 / 2` is -47 and `-47 / 5` is -9; `-46 + 5 / 2` is -44 and `-44 / 5` is -8. Some more thought leads you to the correct answer, as follows:

```
public int average (int sum, int count)
{  if (sum >= 0)
      return (sum + count / 2) / count;
   else
      return (sum - count / 2) / count;
}  //======================
```

The `average` method now computes the correct value: `average(-49, 5)` returns the number -10 and `average(-46,5)` returns the number -9. However, something feels wrong here. This is an instance method, but there is no instance to act as the executor. The method deals with numbers only, no objects at all. You would have to create an object of the class to which the method belongs before you could use the `average` method, and then the object would be irrelevant to the calculation.

Class methods

Java provides a mechanism to handle such situations: The word `static` in the method heading means that it is a class method, i.e., you may call it with the name of the class in

place of the executor. In such a method you cannot use `this`, either explicitly or by default, since there is of course no executor that `this` refers to. So the above method should have its heading begin `public static int average`.

Several different calculations come up from time to time involving numbers alone, so it is useful to have an entire class containing such utility methods. We will call it MathOp. So `x = MathOp.average(49,5)` stores 10 in `x`. Listing 5.1 describes the MathOp class, with the `average` method and another useful method for raising a number to a power of 2: `MathOp.powerOf2(5)` returns the number 32 and `MathOp.powerOf2(10)` returns the number 1024. Note that `powerOf2` illustrates a for-statement that does not have any initializer part (because `expo` is already initialized). It multiplies `power` by 2 once for each time through the loop.

Listing 5.1 The MathOp utilities class

```
/** Class methods that do useful things with numbers.  */

public class MathOp
{
   /** Return the value of 2 to the power expo. */

   public static int powerOf2 (int expo)
   {  int power = 1;
      for (;  expo > 0;  expo--)
         power = power * 2;
      return power;
   }  //=====================

   /** Return sum divided by count rounded to the nearest int. */

   public static int average (int sum, int count)
   {  if (sum >= 0)
         return (sum + count / 2) / count;
      else
         return (sum - count / 2) / count;
   }  //=====================
}
```

Both of these MathOp methods could cause the program to fail in some situations. These defects are corrected in the exercises. Can you see what those situations are without peeking ahead to the exercises?

Utilities classes

We call a class with no instance methods or instance variables or main method a **utilities class**. MathOp is an example, and you will see more later. Since all of its methods are class methods, there is no point in creating objects of MathOp type. Some people call such a class a non-instantiable class. You could think of "MathOp" as being an "operative", a person who carries out certain math-related tasks for you. It does not have to be constructed because it has no state (i.e., instance variables that store information).

It is of course possible for outside classes to create MathOp objects (though pointless). Since MathOp does not define a constructor explicitly, outside classes can use `new MathOp()`, the default constructor supplied by the compiler. You can prevent this by adding a MathOp constructor and declaring it `private`, as in the following:

```
private MathOp()
{  super();
}  //======================
```

Now the default constructor does not exist, since the compiler supplies it only when no other constructor is defined. And the actual constructor is not visible to outside classes, since it is private. Some people feel it is a good idea to do this.

Four categories of methods

The phrase that precedes the return type (or void) in a method's heading tells which of several categories it is in (X denotes the class in which the method is defined):

- `public`: Callable from any class with an executor of class X.
- `private`: Callable only within class X with an executor of class X.
- `public static`: Callable from any class with X in place of the executor.
- `private static`: Callable only within class X with X in place of the executor.

If you call the method from within class X, you may omit the executor or class name before the dot -- it defaults to the executor of the method it is in or to the class it is in, respectively. A class method can be called with an executor if you wish, a variable that can refer to an object of the class, but there is no advantage in doing this.

Independent class methods

The `average` and `powerOf2` methods in Listing 5.1 could be put in any classes at all and they would work the same. They are **independent class methods**. Independent class methods could also be called utility methods. The `Vic.say` and `Vic.reset` introduced in Chapter Two are also class methods. They are defined in the Vic class. A key difference between them and the `MathOp.average` and `MathOp.powerOf2` methods is that the Vic methods can only be in the Vic class. This is because the Vic class methods access private parts of the Vic class that you cannot otherwise get to. So those two Vic class methods are not independent class methods.

Language elements
A declaration of a method can have the word static before its return type.
Such a method can be called using the class name in place of an executor.

Exercise 5.1 The `average` method causes the program to fail if `count` is zero, since division by zero does not make sense. Modify the method to return zero in such cases.
Exercise 5.2 The `powerOf2` method returns 1 whenever the `expo` is negative, which is okay. But it produces the wrong answer if `expo` is more than 30, because the largest possible int value is $2^{31} - 1$. Modify the method to return 2^{30} in such cases.
Exercise 5.3 Write a method `public static int power (int base, int expo)` for MathOp: It returns `base` to the `expo` power. Return -1 in all cases in which the power cannot be computed using int values. But return zero if either parameter is negative. Hint: The largest int value is 2,147,483,647.
Exercise 5.4* Write a method `public static int gcd (int one, int two)` for MathOp: It returns the greatest common positive divisor of its two int parameters. Hint: If either is negative, multiply it by -1; then repeatedly replace the larger by the remainder from dividing the larger by the smaller until one goes evenly into the other, in which case that one will be the greatest common divisor. What do you do if either is zero?
Exercise 5.5* Under what circumstances can a call of a class method be polymorphic?
Exercise 5.6** Write a MathOp class method that finds the factorial of a given int value (e.g., `6!` is `6 * 5 * 4 * 3 * 2 * 1`). Watch out for negatives.

5.2 Declaring Class Variables; Encapsulation

You could put the following variable declaration in the Person class of Listing 4.4, outside of every method definition. This declaration means that the variable named `theNumPersons` is initially zero (i.e., when the program starts), though it is expected to change as the program executes:

```
public static int theNumPersons = 0;
```

A local variable is declared inside a method definition. You can only use it inside that one method definition. Since `theNumPersons` is not a local variable, you can use it everywhere. The word `static` means that, if you mention it outside of the Person class, you may refer to it as `Person.theNumPersons`, i.e., with the name of the class. So it is called a **class variable**, analogous to a class method. That way, (a) the compiler knows where to look for its declaration, and (b) you can declare a different variable named `theNumPersons` in any other class if you want. Note that this is the same access rule Java has for class methods.

A **field variable** is any variable declared outside any method definition. If it is declared with the word `static`, it is a class variable (e.g., `theNumPersons` just described). The class itself contains the only copy of class variables such as `theNumPersons`.

If a field variable is not declared using the `static` keyword, it is an instance variable (e.g., `itsFirstName` for the Person class). Each instance (object) of a class contains its own separate copy of the instance variables. For example, if `sam` and `sue` are Person variables referring to Person objects, then `sam.itsFirstName` is a completely different variable from `sue.itsFirstname` but `sam.theNumPersons` and `sue.theNumPersons` are the same variable. In general, use an instance variable to store information about an individual object, but use a class variable to store information about the class as a whole (information that is not specific to an individual instance).

You could add the statement `theNumPersons++;` to the constructor `public Person()`. Then `theNumPersons` keeps track of the number of completely new Person objects that have been created so far. Any outside class can find out from the Person class how many it has made by looking at the value of `Person.theNumPersons`.

Unfortunately, no outside class can trust that `Person.theNumPersons` truly does tell the number of Persons that have been made. After all, any other outside class could underhandedly change the value of `Person.theNumPersons`, perhaps doubling it to fool the other classes.

Problem: This way of making information available makes the information worthless.
Solution: Encapsulation, also known as information-hiding.

Encapsulation

Encapsulation basically requires that a class not let outside classes change its variables. So field variables should generally be declared as private. That way nothing outside of the class that owns the variable can sneak in and change the state of the variable without the owner class knowing about it. We will add to the Person class the following declaration of `theNumPersons` instead of the one given earlier:

```
private static int theNumPersons = 0;
```

You can still make available to outside classes the value of the variable `theNumPersons` by having the following class method in the Person class:

```
public static int getNumPersons()
{  return theNumPersons;
}  //======================
```

Any outside class can then use `Person.getNumPersons()` to get the value of `theNumPersons`. But no outside class can change its value. In general, any method or field variable that does not need an executor should be declared as a class method.

Failure to encapsulate was the most pernicious cause of bugs in programs in the early decades of programming. With encapsulation, any outside classes that modify these field variables must go through the class's methods to do so, if then. This makes bugs less likely. Listing 5.2 (see next page) contains the complete Person class as revised from Listing 4.4.

In the Nim class of Listing 4.8, each game object should have its own individual value of `itsNumLeft` and `itsMaxToTake`, but they can all share the same random number generator. So it could be declared outside of every method as follows:

```
private static java.util.Random randy
                        = new java.util.Random();
```

Initial values of class variables

A class variable exists independently of any instance of the class. You should almost always give it an initial value where it is declared. If the initial value is not given there, the compiler gives it a **default initial value**: zero for a numeric variable, `null` for an object variable, and `false` for a boolean variable.

Initializations of class variables at runtime are done when the class is first loaded, before any instances of the class are created, and in the order they are listed in the class definition. So the initialization of one class variable should not refer to the value of a class variable declared later in the class definition.

Programming Style It is good style to explicitly state in your classes the initial value of a class variable when your logic requires it to have one, rather than relying on the default value. Note that Listing 5.2 does this. That way a reader of the class does not have to stop and think what the default value is.

Typical structure of an information facility

Listing 5.2 illustrates a very common way of equipping a class to provide certain information about itself. In this case, the information to be provided is the number of Person objects created so far in the program. Listing 5.2 has three relevant parts:

(a) A class query method so that others can access the information, e.g., the method call `Person.getNumPersons()`. This kind of method usually just returns the value of a class variable (`theNumPersons` in this case).
(b) The declaration of the class variable with its initial value specified.
(c) Statements to update the value of the class variable in each method that modifies the information to be provided (only the constructor in Listing 5.2).

You have seen a similar structure for equipping an object to provide certain information about itself. An example is the last name of a Person. Listing 5.2 has three relevant parts:

(a) An instance query method so that others can access the information, e.g., the method call `sue.getLastName()`. This kind of method usually just returns the value of an instance variable (`itsLastName` in this case).
(b) The declaration of the instance variable, sometimes with its initial value specified. However, the initial value may be specified in the constructors instead (as is done for `itsLastName` in Listing 5.2).
(c) Statements to update the value of the instance variable in each instance method that modifies the information to be provided, e.g., `sue.setLastName("Jones")`.

Listing 5.2 The Person class of objects

```
public class Person extends Object
{
   private static int theNumPersons = 0;  // initialize num
   ////////////////////////////////
   private String itsFirstName;
   private String itsLastName;
   private int itsBirthYear;

   /** Construct a new Person with given names and birth year. */

   public Person (String first, String last, int year)
   {  super();
      theNumPersons++;                    // update num
      itsFirstName = first;
      itsLastName = last;                 // initialize last name
      itsBirthYear = year;
   }  //======================

   /** Tell how many different Persons exist. */

   public static int getNumPersons()      // access num
   {  return theNumPersons;
   }  //======================

   /** Return the birth year. */

   public int getBirthYear()
   {  return itsBirthYear;
   }  //======================

   /** Return the first name. */

   public String getFirstName()
   {  return itsFirstName;
   }  //======================

   /** Return the last name. */

   public String getLastName()            // access last name
   {  return itsLastName;
   }  //======================

   /** Replace the last name by the specified value. */

   public void setLastName (String name)  // update last name
   {  itsLastName = name;
   }  //======================
}
```

Scope and positioning of a field variable declaration

The declaration of an instance variable or class variable can be put anywhere in the class. Its position in the listing does not affect where it can be used within methods. Some people like to put all instance variables at the end of the class, and some like to put them at the beginning. The **scope of a variable** is where it can be used without being directly preceded by a dot and an object or class reference:

- The scope of a class variable is its entire class.
- The scope of an instance variable is all instance methods and constructors in its class.
- The scope of a formal parameter is its method's body.
- The scope of a variable declared in the initializer part of a for-statement is the entire for-statement.
- The scope of any other local variable (declared in a method) is from the point where it is declared to the end of the innermost pair of braces within which it is declared.

Similarly, the placement of methods within the class definition does not affect how they are called within other methods. But if one method calls another in the class, most people find it easier to understand if the method doing the calling comes before the method being called.

Technical note If you declare a variable in e.g. the third statement of a method, you are not allowed to refer to that variable in the first two statements of that method. It is as if the creation of the variable does not occur until that third statement. However, the bytecode that the compiler produces creates all of the local variables of a method when the method begins execution, regardless of where they are declared. The point of declaration does not determine when the variable is created at runtime, it only determines where the variable can be used.

Language elements
A variable declaration that is outside of any method can have the word static before its type. Such a variable can be used with the class name in place of its executor.

Exercise 5.7 Change the Person class of Listing 5.2 so that any outside class can find out the first name of the Person who was most recently created. Use "none so far" for the answer if no Persons have yet been created.
Exercise 5.8 Write a method `public static int range (int one, int two)` for the MathOp class in Listing 5.1: The method returns an integer chosen at random within the range of the two parameters (i.e., between or equal to the two parameters). Use a Random class variable.
Exercise 5.9* Change the Person class of Listing 5.2 so that any outside class can find out the smallest birth year of all the Persons who have been created so far. Use -1 for the answer if no Persons have yet been created.
Exercise 5.10* Change the Person class of Listing 5.2 so that any outside class can find out the average birth year of all the Persons who have been created so far. Use 0 for the answer if no Persons have yet been created.

5.3 Final Local, Instance, And Class Variables

If a variable declaration has the word `final` immediately before the name of the type, the compiler will not let any statement change the value any time after you give it its initial value. Such variables are called constants in most programming languages, but Java programmers tend to call them **final variables**.

Final class variables

The Time class constructor in Listing 4.5 has the phrase `itsMin = itsMin + 60` in one method. Other methods would also mention 60, since that is the number of minutes in an hour. The logic of the Time class would be clearer if you declare a final class variable with the value 60 and use it instead. By convention, we write the variable name all in capital letters if it is a final class variable. So the declaration in the Time class, and the corresponding change in the Time constructor, could be as follows:

```
public static final int MIN_PER_HOUR = 60;  // class variable
itsMin = itsMin + MIN_PER_HOUR;
```

The BasicGame class in Listing 4.3 has an instance variable `itsSecretWord`. Every BasicGame object has exactly the same secret word, though they may have different user's words (depending on what the user chose for that game). It would make more sense to have just one copy of this value for the whole class, rather than one for each BasicGame object. Since the value of this variable never changes, you could write the class variable declaration in the BasicGame class, and the revised statement in the `shouldContinue` method, as follows:

```
public static final int SECRET = "duck";  // class variable
return ! SECRET.equals (itsUsersWord);
```

Programming Style Any constant value that is used in two or more methods in a class should be declared as a final class variable and that variable used in place of the constant value. The name of the variable should be all in capital letters. It may or may not be public. Some exceptions: 2, 1, 0, and "".

This book lists field variables in the order `public static final`, then `private static`, then a divider //// as shown in Listing 5.2, and then the instance variables. This book also puts a prefix of "the" on almost all names of non-final class variables, and never anywhere else. This hallmark, along with the prefix "its" on almost all names of instance variables, makes bugs less likely. If you do not do this in your own definitions, at least obey the following safety principle: Never name a parameter or local variable the same as a class variable or instance variable.

Final instance variables

In Listing 5.2, no provision is made for changing two of the values of the Person instance variables once they have been assigned. This should be made clear in the declarations:

```
private final String itsFirstName;
private final int itsBirthYear;
```

They could be made publicly visible if there is a good reason to do so, because that would not violate the encapsulation principle: No outside class could change the value of any instance variable of any Person object. But as it is, we have the getXXX methods to retrieve any one of the values, so we need not make them public. In general, it is preferable to access even final instance variables through getXXX methods rather than directly -- it makes it easier to upgrade the software in the future.

The value of a final instance variable must be assigned in the declaration or else in every constructor of the class. You can see that the Person class does the latter, so the addition of the word `final` is the only change needed. When all of an object's instance variables are final, the object is said to be **immutable** (because you cannot change its attributes once it has been created). String values, for instance, are immutable.

Final local variables

You may declare a variable that is local to a method as `final`, in which case you should immediately assign its final value at the point where you declare it. This should normally be done if you use a constant value in two or more places within the method. Some people like to make the name all in capital letters; others reserve that for class variables.

Some people go so far as to say that every constant value used in a method should be a named final variable, other than perhaps 0, 1, and 2 and the empty String. This book does not go that far. That principle would require that every string literal be named.

Caution Do not use the words private or public inside a method definition. If you make this mistake, the message that the compiler gives you can be baffling. Local variables are by nature private, since you can only mention them inside their methods. But only class variables or instance variables are explicitly designated as public or private.

Language elements
A variable declaration can have the word final before its type.
The value of such a variable cannot be changed once it has been assigned.

Exercise 5.11 Rewrite Listing 4.5 to store 10 in a named local final variable, then use it wherever it is appropriate.
Exercise 5.12* Rewrite Listing 4.6 to store both 1 and 100 in named public final variables standing for the lower and upper limits, then use them wherever appropriate.
Exercise 5.13* Rewrite the Nim constructor in Listing 4.8 to use local final variables for the numbers 21 and 3, then use them wherever it is appropriate.

5.4 Two New String Methods

You now have enough background to understand and profit from a complete simulation of the Vic's Programmable CD Organizer. Of course, no one can write the actual program in Java; since it moves armatures and gears and springs, the Vic engineers have to write it in **native** code. If you were to look at their implementation, you would see almost all the methods with the notation `native` and no method bodies.

The Vic simulation we develop in these two sections will not involve graphics; it is too early for that. But the simulation will produce **tracing output** to the terminal window such that, each time one of the four action instance methods is executed, you will see a full description of that sequence of slots. The simulation makes heavy use of class variables and final variables. And it introduces two new String methods from the standard library.

The data structure

Each sequence of slots is represented by a string of non-blank characters, where 0 signals an empty slot and any other character signals a slot that contains a CD whose name is that character. The instance variables are `itsSequence` (the String), `itsPos` (a position in the sequence) and `itsID` (a positive int). So `getPosition` only needs one statement to report a string of characters containing the current position and ID:

```
return itsPos + "," + itsID;
```

The value of `itsID` is also used in the tracing output, to tell you which Vic object is being described. The `trace` method each action instance method calls is the following. The method call `System.out.println` has a single String parameter that it prints to the terminal window:

```
private void trace (String action)
{  System.out.println ("Vic# " + itsID + ": " + action
         + itsPos + "; sequence= " + itsSequence);
}  //=====================
```

The `Vic.say` method can use this output statement as well, having just one statement:

```
System.out.println ("SAYS: " + message);
```

The `backUp` method first checks that `itsPos` is greater than 1, since otherwise the program is to terminate immediately. Then it decrements `itsPos` and calls the trace method in an obvious way. The logic for `backUp` and the three one-liner Vic methods discussed so far is in the upper part of Listing 5.3 (see next page). `System.exit(0)` is not a graceful way of terminating; adding an explanatory message is an exercise.

String methods

Before you go further, you need to know about the two methods in the String class that this software uses. One is the `length` method: `s.length()` returns the number of characters in the String `s`. The other is the `substring` method, which returns a new String value that is a portion of the executor. For instance, `s.substring(2,4)` returns the String value consisting of the characters numbered starting from 2 and going up to but not including 4. That is, you get the characters numbered 2 and 3, in that order.

You also need to know that Java numbering of the positions in a String is **zero-based**: The first character is numbered 0, the second is numbered 1, the third is numbered 2, etc. Therefore, `"abcdef".substring(2,5)` is the string `"cde"`. This all means that the `seesSlot` method can be coded as just one statement, as follows.

```
return itsPos  < itsSequence.length();
```

For instance, if `itsSequence` has length 6, and thus numbers the characters 0 through 5, `itsPos` is beyond the end of the sequence of slots if `itsPos` is 6. For the `seesCD` method, you must look at the substring consisting of one character starting at position `itsPos`, which is expressed in Java as follows:

```
itsSequence.substring (itsPos, itsPos + 1);
```

You now have all the information you need to understand the part of the Vic simulation that does not involve the stack or initializing Vic objects. Study Listing 5.3 for a while (a long while) to be sure it makes sense to you before you go on. A named constant String value NONE represents the absence of a CD, thus has the value `"0"`. The coding for `moveOn` parallels `backUp`. Figure 5.1 describes the two new String methods.

s.length()
is the number of characters in the String object referred to by s.

s.substring(start, end)
returns a new String object consisting of the characters of s in positions start through end-1. The program can crash unless 0 <= start <= end <= s.length().

Figure 5.1 Two String methods

Listing 5.3 The Vic class of objects, part 1 of a String-based simulation

```
public class Vic extends Object
{
   private static final String NONE = "0";
   ////////////////////////////////
   private String itsSequence = "";
   private int itsPos = 1;
   private final int itsID;  // so tracing output can identify it


   public String getPosition()
   {  return itsPos + "," + itsID;
   }  //=====================

   public static void say (String message)
   {  System.out.println ("SAYS: " + message);
   }  //=====================

   private void trace (String action)
   {  System.out.println ("Vic# " + itsID + ": " + action
            + itsPos + "; sequence= " + itsSequence);
   }  //=====================

   public void backUp()
   {  if (itsPos == 1)
         System.exit (0);
      itsPos--;
      trace ("backUp to slot ");
   }  //=====================

   public void moveOn()
   {  if ( ! seesSlot())
         System.exit (0);
      itsPos++;
      trace ("moveOn to slot ");
   }  //=====================

   public boolean seesSlot()
   {  return itsPos  < itsSequence.length();
   }  //=====================

   public boolean seesCD()
   {  if ( ! seesSlot())
         System.exit (0);
      String s = itsSequence.substring (itsPos, itsPos + 1);
      return ! s.equals (NONE);
   }  //=====================
}
```

Chaining

If a method call returns an object value, you may use the method call for the executor of another method call. This is called **chaining** of method calls, or sometimes cascading. For instance, the following statements are legal:

```
// replace spot.equals (getPosition()) by:
getPosition().equals (spot);
// replace the last two statements of seesCD by:
return itsSequence.substring (itsPos, itsPos+1).equals (NONE);
// replace the first three statements in Listing 4.1 by:
new BasicGame().playManyGames();
```

Exercise 5.14 Write a private class method in the Vic class that has a String parameter and an int parameter and returns the one-character substring at the given position. Then replace more complex coding by a call of that method in Listings 5.3, 5.4, and 5.5.
Exercise 5.15* An attempt to `moveOn` or `backUp` or to evaluate `seesCD` when it is illegal causes an abrupt `System.exit(0)` without explanation. The user would appreciate a tracing output in such cases. Revise these three methods to call a private method that explains the problem (with `showMessageDialog`) and then terminates.

5.5 String Implementation Of A Vic Simulator

The stack operations

Since Vics all share the same stack, we begin by declaring a class variable `theStack`, initially an empty string of characters. The `stackHasCD` class method then tells whether the string is not empty, i.e., it tells whether `theStack.length()` is positive.

The process for `takeCD` first checks that `seesSlot()` is `true`; if not, the program terminates. If `seesCD()` is `false`, nothing happens, otherwise `theStack` appends the substring `itsSequence.substring(itsPos, itsPos+1)`. Also, `itsSequence` puts NONE in place of that substring. The way that `itsSequence` puts NONE at position `itsPos` is to make the new value of `itsSequence` be (a) the substring from position 0 up to `itsPos`, followed by (b) NONE, followed by (c) the substring running from position `itsPos + 1` up to the end. The last thing that `takeCD` does is call the `trace` method.

The upper part of Listing 5.4 (see next page) contains the Java coding for the `stackHasCD` method and the `takeCD` method just described. The logic for the `putCD` method is rather more complex; a reasonable plan is in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for putCD
1. Exit the program if there is no slot at the current position.
2. If the current slot does not have a CD and the stack does, then...
    2a. Change `itsSequence` to be a new string consisting of three parts:
        (a) all its characters up to but not including the current position;
        (b) the top value on the stack;
        (c) all its characters after the current position.
    2b. Remove the top value from the stack.
3. Print out a tracing message.

The implementation of `putCD` in the lower part of Listing 5.4 differs somewhat from the design. The first thing the coding does is test `! seesCD()`, which will exit the program immediately if there is no slot at the current location. So Step 1 (testing `seesSlot()`) does not have to be coded explicitly.

Listing 5.4 The Vic class of objects, parts involving theStack

```
// public class Vic continued: using the stack

   private static String theStack = "";   // initially empty

   public static boolean stackHasCD()
   {  return theStack.length() > 0;
   }  //======================

   public void takeCD()
   {  if (seesCD())
      {  theStack = theStack
                  + itsSequence.substring (itsPos, itsPos + 1);
         itsSequence = itsSequence.substring (0, itsPos) + NONE
                  + itsSequence.substring (itsPos + 1,
                                    itsSequence.length());
      }
      trace ("takeCD at slot ");
   }  //======================

   public void putCD()
   {  if ( ! seesCD() && stackHasCD())
      {  int atEnd = theStack.length() - 1;
         itsSequence = itsSequence.substring (0, itsPos)
                     + theStack.substring (atEnd, atEnd + 1)
                     + itsSequence.substring (itsPos + 1,
                                    itsSequence.length());
         theStack = theStack.substring (0, atEnd);
      }
      trace ("putCD  at slot ");
   }  //======================
```

The reset method

The value passed in to the `reset` method has the type description `String[]`, which has not yet been discussed in this book. A full explanation of these array variables has to wait until Chapter Seven, but the foretaste you get here should be manageable.

The Vic class has a class variable named `theTableau` where it keeps what are initially the three empty String values it allows. The type description of this variable is `String[]`, which means it can hold several String values. The class variable `theTableau` can be initialized to be three empty Strings with this coding:

```
   private static String[] theTableau = { "", "", "" };
```

The `reset` method simply assigns its `args` parameter to this `theTableau` variable, replacing the current value of three empty Strings by however many non-empty Strings the user gives as input in `args`. However, in accordance with the specifications for `reset` in Chapter Two, it does not make the assignment if `args` has no strings at all or if some Vic object has already been constructed.

The class variable `theNumVics` keeps track of the number of Vic objects created so far. This is done just as you saw for `theNumPersons` in the earlier Listing 5.2: Initialize it to zero in the declaration, then increment it in the constructor. This variable is used to determine the `itsID` value for each Vic, and it is also used by the `reset` method: If `theNumVics` is not zero, the `reset` method has no effect.

Each object whose type is `String[]` has a public final instance variable named `length` that you can test to find out how many elements are in the array. So the `reset` method makes sure that `args.length` is positive before it replaces `theTableau` by `args`. The full coding for the `reset` method is therefore as follows:

```
public static void reset (String[ ] args)
{  if (theNumVics == 0 && args.length > 0)
      theTableau = args;
}  //=====================
```

The Vic constructor

We saved the hardest method for last. To help the constructor do its job, we have two class variables: One is a random number generator and the other is a named constant that contains the first 12 letters of the alphabet. These letters are at positions 1 through 12 of the LETTERS string, with the character at position 0 left blank. The value of `itsPos` will always be at least 1 for any Vic object. In effect, we switch to a one-based numbering of characters in a String in spite of Java's preference for zero-based. You should compare the following development of the constructor with its coding in Listing 5.5 (see next page), where each line of the method is numbered.

The first thing the Vic constructor does is create the object (line 1) and then check that `theNumVics` is less than the number of strings in `theTableau` (which will be three unless they were replaced by the `reset` method, in which case it could be dozens, depending on the user's choice; see line 2). If `theNumVics` is too large, we already have all the objects we are allowed, so we do nothing but increment `theNumVics` (line 13) set `itsID` to that value (line 14), and print a tracing message (line 15). These three actions are what we do at the end of the construction process for regular Vics too. So the question is, what do we do if `theNumVics` is not too large?

First we need to make `itsSequence` equal to the corresponding array element (line 3). That is done with the following statement (fully explained in Chapter Seven; we do not say anything more about arrays here):

```
itsSequence = theTableau[theNumVics];
```

If this is not the empty String, it must be the one that came from `reset`, so we are done with the construction process (except putting a blank on the front and doing the three actions mentioned earlier). But if it is the empty String (line 4), we have a loop that executes from 3 to 8 times (randomly chosen in line 5), each time adding a 1-character string to the front of `itsSequence`. The string it adds is NONE half the time (again chosen at random; lines 6-7) and is otherwise the corresponding letter from the LETTERS value (lines 8-10). For instance, an object with six slots and only the second and fourth slots having no CD has `"a0c0ef"` for `itsSequence`. Then a blank is put on front (to make it one-based; line 11) and the three actions mentioned earlier are done.

Caution Any time you work out a complex logic such as this, you need to go over it very carefully to make sure that there are no errors. This Vic object class crashed on its fourth test run because the constructor does not always assign a value to `itsSequence`. After this bug was found, the declaration of `itsSequence` was changed to initialize it to the empty String. The bug was caused by a failure to obey a simple principle: Initialize every instance variable in its declaration unless you are absolutely sure it is initialized in every constructor.

Listing 5.5 The Vic class of objects, completed

```
// public class Vic completed: constructor and reset

   private static final java.util.Random randy
                     = new java.util.Random();
   private static final String LETTERS = " abcdefghijkl";
   private static int theNumVics = 0;
   private static String[ ] theTableau = {"", "", ""};


   public static void reset (String[ ] args)
   {  if (theNumVics == 0 && args.length > 0)
         theTableau = args;
   }  //=====================

   public Vic()
   {  super();                                              // 1
      if (theNumVics < theTableau.length)                   // 2
      {  itsSequence = theTableau[theNumVics];              // 3
         if (itsSequence.length() == 0)                     // 4
            for (int k = 3 + randy.nextInt (6); k >= 1; k--) // 5
               if (randy.nextInt (2) == 0)                  // 6
                  itsSequence = NONE + itsSequence;         // 7
               else                                         // 8
                  itsSequence = LETTERS.substring (k, k + 1) // 9
                              + itsSequence;                // 10
         itsSequence = " " + itsSequence;                   // 11
      }                                                     // 12
      theNumVics++;                                         // 13
      itsID = theNumVics;                                   // 14
      trace ("constructed ");                               // 15
   }  //=====================
```

Exercise 5.16 If the user calls the `reset` method twice before any Vic is constructed, what happens the second time?
Exercise 5.17 Modify the simulation by having each tracing statement begin with a list of the elements in `theStack`.
Exercise 5.18* Describe the consequence of reversing the order of indexing in the constructor to have `for (int k = 1; k <= 3 + theRandy.nextInt(6); k++)`.
Exercise 5.19* Describe the consequence of forgetting the phrase `! seesCD() &&` in the coding of `putCD`.
Exercise 5.20* Describe the consequence of forgetting the phrase `&& stackHasCD()` in the coding of `putCD`.

5.6 Case Study: Introduction To Networks

This Network material presents a completely different software situation from the Vic software, so you can see more examples of how to use the language elements you have learned. No new language features are introduced in these three sections.

Some situations where networks arise

One relevant situation is the network of nodes on the World Wide Web. Each node can send messages directly to several other nodes. A message can be routed from one node to any other node by having it pass through several direct node-to-node connections along the way.

Another relevant situation is a group of students who are registered for a group of courses. Each student has registered for several courses and each course has several students registered for it.

A third relevant situation is the set of airports served by a particular airline. Each airport provides direct flights to only a few other airports. But one can get to any other airport (hopefully) by taking a series of direct flights.

Networks and nodes

These situations have common elements you can model in software. A Network object corresponds to the WWW or the college or the airline. Each Network object has a (usually long) list of all the Node objects in the network. Nodes correspond to internet nodes or students or courses or airports. Each Node object has a (relatively short) list of other Node objects that it connects to. Figure 5.2 shows an example of a network and gives the node-list for the entire network and for each node in it.

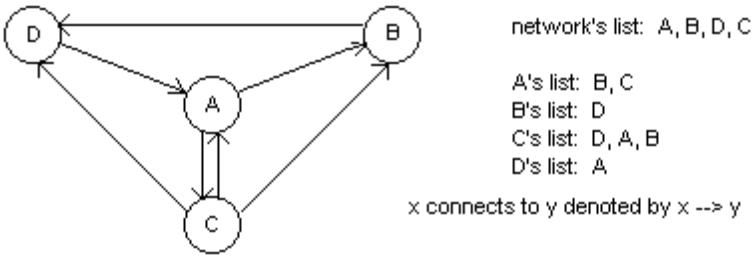


Figure 5.2 Example of a network with four nodes

"Node x connects to Node y" models any of these relations:

- Internet Node x can send a message directly to Node y.
- College Student x is registered for Course y.
- College Course x has registered in it Student y.
- The Airline has a direct flight from Airport x to Airport y.

The Network and Position classes

The only two methods for the Network class are as follows:

- `net = new Network()` constructs a Network object that represents an actual network. In other words, it creates a virtual network. Each use of this constructor normally gives a different network.
- `net.nodes()` produces the list of Nodes in the Network object `net`, starting with the first node.

The `nodes` method is the key method. It produces a new Position object. That Position object iterates through the list of all the Nodes in the Network one at a time. If you get a Position object from a Network `net` using e.g. `Position pos = net.nodes()`, you can use the following three instance methods:

- `pos.moveOn()` changes the position of `pos` to the next Node on the list of nodes (like Vic's `moveOn`).
- `pos.hasNext()` tells whether there is a Node at the current position of `pos` in its list (like Vic's `seesSlot`).
- `pos.getNext()` returns the Node object at the current position of `pos` in its list (not like Vic's `takeCD`, since `getNext` does not remove or change the Node at the position, it only lets you look at it).

The application program in Listing 5.6 illustrates all of these commands, as well as one for Node objects: `current.getName()` returns the name of the node referred to by `current`. This coding also illustrates the basic counting logic: If you initialize a variable to zero and increment it once each time through a loop, then when you exit the loop that variable will contain the number of iterations of the loop.

Listing 5.6 A program using a Network object

```
import javax.swing.JOptionPane;

public class NetApp
{
   /** List all nodes and tell how many there are. */

   public static void main (String[ ] args)
   {  Network school = new Network();
      int count = 0;
      for (Position pos = school.nodes();  pos.hasNext();
                   pos.moveOn())
      {  Node current = pos.getNext();
         JOptionPane.showMessageDialog (null,
                   current.getName() + " is one of the nodes.");
         count++;
      }
      JOptionPane.showMessageDialog (null,
                   "The total number of nodes is " + count);
      System.exit (0);
   }  //=====================
}
```

When a program executes, it almost always processes input and produces output. Typically, the output is the answer to a problem the program solves. For a Vic object, the input is the initial state of the mechanical components and the output is the final state of the mechanical components. For a Network object, the input is the initial state of the list of Nodes and their connections; the output is whatever you display for the user to see.

The Node class

The `pos.getNext()` method call produces a Node object, one of the Nodes in the Network. You can do several things with Nodes. For instance, some network situations involve two distinct groups of Nodes, as with colleges and students. Or they might represent people, some of whom are male and some female. To have a general model for such cases, we say some nodes are blue and some not.

For the college situation, blue nodes might represent students and non-blue nodes courses. Or for people, blue nodes might represent males and non-blue nodes females. For airports, we indicate that color does not matter by having all nodes blue. The method call `sam.isBlue()` tests the Node object that `sam` refers to to see if it is blue. The logic in Listing 5.6 could be modified to count the blue nodes and print the name of each by replacing the body of the for-statement by the following:

```
   Node current = pos.getNext();
   if (current.isBlue())
   {  JOptionPane.showMessageDialog (null,
                current.getName() + " is a blue node.");
      count++;
   }
```

You cannot change the state of a Network. But you can go down the list of nodes a given node connects to to find out things about those connections, as you will see next. The methods for Networks, Nodes, and Positions provide some very useful services for answering questions about a network.

The four Node methods

You have already seen two instance methods for Nodes. We introduce here two new ones, so you now have four altogether:

- `aNode.getName()` returns a String representation of the Node.
- `aNode.isBlue()` tells whether `aNode` belongs to one of two categories of Nodes.
- `aNode.equals(anotherNode)` tells whether two (possibly different) Node objects represent the same Node in the network, analogous to String `equals`.
- `aNode.nodes()` returns a Position object that iterates through the list of all Nodes that `aNode` connects to in the network.

A software suite to track water flow through the pipes of a municipal system would involve network operations. Figure 5.3 shows a network of water pipe connections. The arrows indicate the direction of flow through the pipes. If `pipes` refers to the Network here, the list that `pipes.nodes()` produces has all eight nodes of the network on it. However, the list that `x.nodes()` produces for various node values `x` has at most two nodes on it. For instance, as the figure implies, `E.nodes()` produces a Position object `pos` for which `pos.getNext()` is H, and `pos.moveOn(); pos.getNext()` is F.

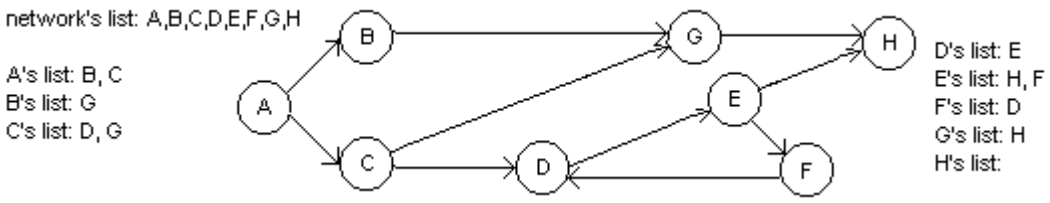


Figure 5.3 A network with 8 nodes

You might get two different descriptions of the same node when you use the `getNext` method. For instance, if `bee` and `cee` are Node variables that refer to nodes B and C in Figure 5.3, then `Node x = bee.nodes().getNext()` and `Node y = cee.nodes().getNext()` may both give you an object that refers to node G, but they may be different objects (i.e., stored in different places in RAM, so `x == y` is `false`). However, `x.equals(y)` will be true since they both represent node G.

A utilities class for Nodes

No constructor has been specified for Positions or Nodes, so you cannot make a subclass that gives additional abilities to Positions or Nodes. But you can create a NodeOp class to hold various class methods that deal with Nodes. These methods are declared using the word `static`, meaning you call them with the class name in place of the executor. Listing 5.7 (see next page) is a start on this utilities class.

The seesOnlyBlue method

An example of the use of a NodeOp method is the following statement:

```
if (NodeOp.seesOnlyBlue (sam))
   JOptionPane.showMessageDialog (null, "only blue nodes");
```

Listing 5.7 The NodeOp class

```
/** Answer queries about one or two given nodes. */

public class NodeOp
{
   /** Tell whether par connects only to blue nodes. */

   public static boolean seesOnlyBlue (Node par)
   {  for (Position p = par.nodes();  p.hasNext();  p.moveOn())
         if ( ! p.getNext().isBlue())
            return false;
      return true;
   }  //======================

   /** Tell whether from connects to target.  */

   public static boolean connected (Node from, Node target)
   {  for (Position p = from.nodes();  p.hasNext();  p.moveOn())
         if (p.getNext().equals (target))
            return true;
      return false;
   }  //======================

   /** Tell whether par connects to any node of the same color.*/

   public static boolean seesSameColor (Node par)
   {  return (par.isBlue() && ! seesOnlyNonBlue (par))
             || (! par.isBlue() && ! seesOnlyBlue (par));
   }  //======================

   /** Tell whether par connects to no blue node. */

   public static boolean seesOnlyNonBlue (Node par)
   {} // left as exercise
}
```

The purpose of the `seesOnlyBlue(Node)` class method in Listing 5.7 is to tell whether the parameter `par` connects to nothing but blue nodes. A good design is: You run down the list of nodes `par` connects to. If you see a node that is not blue, the answer to the question, "Does `par` connect only to blue nodes?" is `false`. If you get to the end of the list without seeing a non-blue node, the answer to the question is `true`.

The method call `p.getNext()` returns a Node value, and the executor of the `isBlue` method must be a Node value. Therefore, `p.getNext().isBlue()` is a legal chain of method calls that asks whether the Node returned by `p.getNext()` is blue.

The connected method

The purpose of the `connected(Node, Node)` class method is to tell whether the first parameter `from` connects to the second parameter `target`. A good design is: You run down the list of nodes `from` connects to. If you see the `target` node, the answer to the question "Does `from` connect to `target`?" is `true`. If you get to the end of the list without seeing the `target` node, the answer to the question is `false`.

Note that these two methods have almost exactly the same structure, except the `true` and `false` values are switched. You will see these two looping patterns very frequently. The first is typical of All-A-are-B conditions (specifically, "All nodes connected to `par` are blue"), so we call it the **All-A-are-B looping action**. The second is typical of Some-A-are-B conditions (specifically, "Some node connected to `from` equals `target`"), so we call it the **Some-A-are-B looping action**.

The seesSameColor method

The purpose of the `seesSameColor(Node)` method is to tell whether some node the parameter connects to is the same color as the parameter. It is logical that a node is connected to another node of the same color if and only if it is blue but not connected only to non-blues, or it is non-blue but not connected only to blues.

All three of the methods in Listing 5.7 are query methods because, after the executor returns `true` or `false` from the method call, every object is in the same state it had when you made the call. You may protest that the object obtained by the statement `p = par.nodes()` has changed, and you would be right. But that object is irrelevant, since:

1. That object did not exist when you called the method,
2. That object cannot be used after you return from the method, and
3. The fact that it was created and modified has no effect on anything after you return from the method. This is because each future call of `nodes()` get a totally new Position object.

Exercise 5.21 Write a main method that only prints out the name of the last blue node, but prints "no blues" if there are none.
Exercise 5.22 Write the method `seesOnlyNonBlue` described in Listing 5.7.
Exercise 5.23 Write a method `public static int getNumNodes (Node par)` for NodeOp: It tells how many nodes its Node parameter connects to.
Exercise 5.24 Write a method `public static int bidirectional (Node par)` for NodeOp: It tells whether every Node `par` connects to, connects to `par`.
Exercise 5.25* Revise Listing 5.6 to print the percentage of nodes that are blue.
Exercise 5.26* Draw the UML class diagram for Listing 5.6.
Exercise 5.27* Write a method `public static int hasA (Node par)` for NodeOp: It tells whether the name of any Node that `par` connects to begins with "A".

5.7 Extending The Network Class

The SmartNet class in Listing 5.8 (see next page) augments the Network class by adding three useful methods. To use this class, you need only start your program with a command such as `SmartNet net = new SmartNet()`. Figure 5.4 shows the UML class diagram for the SmartNet class.

The purpose of the `connectsToAll(Node)` method is to tell whether the Node parameter connects to every other node. The executor looks through the list of all nodes in the network until it sees one the given node does not connect to, then returns `false`. It returns `true` only when the given node connects to every other node. So this is another All-A-are-B looping action.

The purpose of the `getNumNodes()` method is to tell how many nodes are in the whole network. The executor initializes a counter to zero. Then it goes through the list of all nodes in the network and adds 1 to the counter each time it sees a node. So the final answer must be the number of nodes in the whole network.

Listing 5.8 The SmartNet class

```
import javax.swing.JOptionPane;

public class SmartNet extends Network
{
   /** Tell whether par connects to all other nodes in this
    *  network. */

   public boolean connectsToAll (Node par)
   {  for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
      {  Node current = pos.getNext();
         if ( ! (current.equals (par)
                     || NodeOp.connected (current, par)))
            return false;
      }
      return true;
   }  //=====================

   /** Return the total number of nodes in this network. */

   public int getNumNodes()
   {  int count = 0;
      for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
         count++;
      return count;
   }  //=====================

   /** List all nodes in this network that connect to
    *  some node of the same color. */

   public void printSameColorConnections()
   {  for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
      {  Node current = pos.getNext();
         if (NodeOp.seesSameColor (current))
            JOptionPane.showMessageDialog (null,
                                           current.getName());
      }
   }  //=====================
}
```

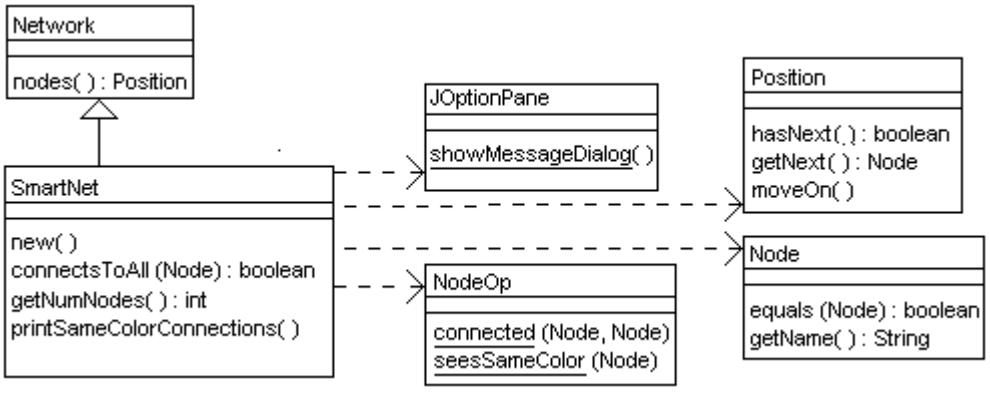


Figure 5.4 UML class diagram for the SmartNet class

The `printSameColorConnections()` method is used in a situation where you expect every node to connect to a node of a different color, such as students registered for courses. But you want to make sure this is true. The executor looks through the list of all nodes in the network to find those that are connected to a node of the same color (i.e., a blue node connected to a blue node or a non-blue node connected to a non-blue node). It prints all such nodes it sees.

You may wonder why this subclass of Network is called SmartNet. If you have a friend to whom you teach Spanish, is your friend still the same person? Answer: Yes, but a smarter person, since now your friend can speak Spanish. Similarly, making `net = new SmartNet()` instead of `net = new Network()` would produce the same network of nodes, but that network would be able to answer the question, "Does this particular Node connect to all other Nodes?", which a plain Network cannot answer. That is, SmartNet objects are smarter than plain Network objects.

Exercise 5.28 Rewrite the condition in SmartNet's `connectsToAll` method to use `&&` rather than `||`.
Exercise 5.29 Write a SmartNet method `public boolean noLoners()`: The executor tells whether each of its nodes is connected to at least one node.
Exercise 5.30 Write a SmartNet method `public boolean atLeastOneNonBlue()`: The executor tells whether at least one of its nodes is not blue.
Exercise 5.31* Write a SmartNet method `public boolean isBipartite()`: The executor tells whether every node connects only to nodes of the opposite color. Call on an existing NodeOp method to do most of the work.
Exercise 5.32* Write a SmartNet method `public boolean numBlues()`: The executor tells how many blue nodes it has.
Exercise 5.33* Write a SmartNet method `public boolean hasUniversalNode()`: Tell whether any node is connected to every node except possibly itself.
Exercise 5.34* Rewrite the SmartNet class so that `itsNumNodes` is an instance variable and `getNumNodes` returns its value rather than re-calculating it each time it is called.
Exercise 5.35* Draw the UML diagram for the SmartNet class.
Exercise 5.36** Write a SmartNet method `public boolean tellFirst (Node one, Node two)`: Tell which of the two given nodes comes first on the list of all network nodes. Return `null` if neither is on the list.

5.8 Analysis And Design Example: The Reachability Problem

Marking nodes with numbers

Each Node has a color (blue or not) and a name; you cannot change them. But each Node also has an integer value you can change. This value is used to mark Nodes you have processed during execution of an algorithm, so you do not process them again.

- `x.setMark(5)` sets Node `x`'s marker number to 5 (you can use any int value here). The marker number is initially zero for all Nodes when the Network is created.
- `x.getMark()` returns the current value of `x`'s marker number.

The Reachability Problem

An important problem in the study of networks is to find out whether it is possible to send a message from a given starting point to every other node. The message can be routed through as many other nodes as is necessary, as long as it gets through eventually. For the Airline situation, the problem amounts to finding out whether the airline can get you to any airport from a given airport starting point, clearly a desirable quality in an airline.

This is the **Reachability Problem**. The `setMark` and `getMark` methods are intended to help solve this problem and many others. They are used to solve the Traveling Salesman Problem in the optional Section 5.9 on recursion.

Solution to the Reachability Problem

Solving the problem of whether all nodes are reachable from a given node is rather complex, so you need a plan. A first approach is as follows: Mark 1 on every node you can reach from the starting point. Then mark 1 on every node you can reach from one of those you marked. Then mark 1 on every node you can reach from one of those, etc. When you cannot mark any more, see if every node in the network has been marked.

How do you keep track of each node you have checked out (that is, you have marked the nodes you can reach from it)? You can use a different mark, say 2 instead of 1. So 1 means it is reachable but you have not yet checked out which nodes you can reach from it, and 2 means it is both reachable and checked out. This design can be refined as shown in the accompanying design block. Figure 5.5 traces the first few steps of this algorithm.

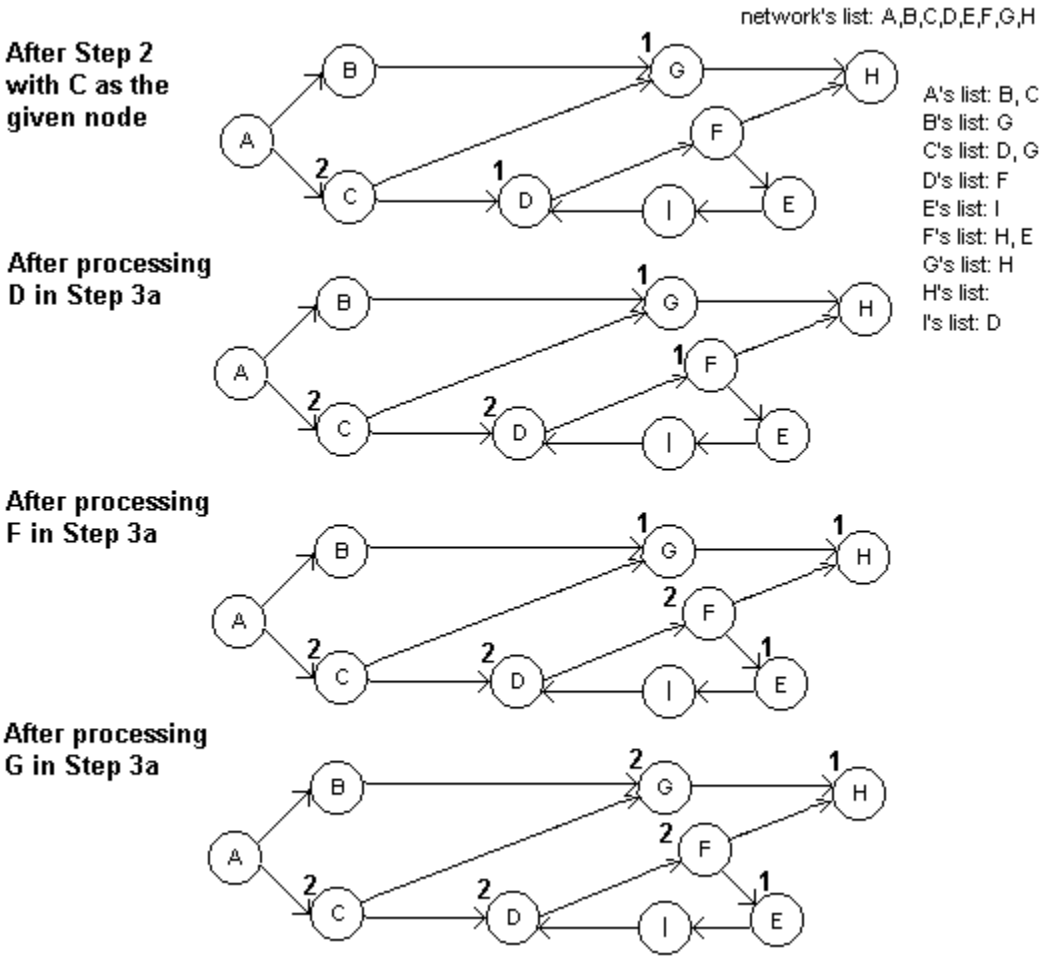


Figure 5.5 Steps in the Reachability algorithm

How do you know when to repeat Step 3? Since this is a yes-no question, you could use a boolean variable. Set it `false` at the beginning of Step 3, then set it `true` if you find a node marked 1 at Step 3a. Repeat Step 3 if you find the boolean variable has turned `true` after going through the list of nodes. Listing 5.9 contains the complete logic.

STRUCTURED NATURAL LANGUAGE DESIGN for the searching method

1. Mark 2 on the given node (since you are about to check it out).
2. Mark 1 on every node you can reach from the given node.
3. For each node `current` in the list of nodes of the network, do...
   - 3a. If `current` is marked 1 (reachable but not yet checked out), then...
     - 3aa. Mark 2 on `current` (since you are about to check `current` out).
     - 3ab. Mark 1 on every node you can reach from `current` unless it already has a mark of 1 or 2.
4. Repeat Step 3 until you find no more nodes marked 1.
5. Return `true` if no nodes are marked 0; return `false` otherwise (since a 0 means it cannot be reached from the original node).

Listing 5.9 A SmartNet method for the Reachability Problem

```
/** Tell whether every node is reachable from the given node.
 *  Precondition:  source is not null.  */

public boolean allReachableFrom (Node source)
{  checkOut (source);  // marks 2 on source, 1 on some others
   boolean foundNodeToCheckOut;
   do
   {  foundNodeToCheckOut = false;
      for (Position pos= nodes(); pos.hasNext(); pos.moveOn())
      {  Node current = pos.getNext();
         if (current.getMark() == 1)
         {  foundNodeToCheckOut = true;
            checkOut (current);
         }
      }
   }while (foundNodeToCheckOut);
   return allNodesAreMarked();
}  //=====================

/** Mark 2 on par; mark 1 on all nodes reachable from par
 *  except for those already marked 1 or 2. */

private static void checkOut (Node par)
{  par.setMark (2);
   for (Position p = par.nodes();  p.hasNext();  p.moveOn())
   {  Node current = p.getNext();
      if (current.getMark() == 0)
         current.setMark (1);
   }
}  //=====================

private boolean allNodesAreMarked()
{  boolean valueToReturn = true;
   for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
      if (pos.getNext().getMark() == 0)
         valueToReturn = false;
      else
         pos.getNext().setMark (0);
   return valueToReturn;
}  //=====================
```

Step 2 requires more than one or two statements to implement, so it is done as a call to a separate private helper method named `checkOut` which includes Step 1. Step 3ab uses the same method. Step 5 also requires more than one or two statements, so it has a separate method too, named `allNodesAreMarked`. Note: Listing 5.9 is rewritten in a much more efficient way in Section 5.9.

Programming Style You may well ask why the `checkOut` method is a class method instead of an instance method. The reason is, the Network object itself is not used at all (no explicit or implicit `this`). It would be deceptive to make it an instance method, and deception is not good style.

Implementing the Network class with a prototype

You are probably wondering why you do not get any Network software with which to test your programs, analogous to the Vic software. The reason is simple: This is a major programming project at the end of Chapter Seven (when you have learned about arrays).

For now, you can use the three classes in this section. They have overly simple logic, they are not adequate for realistic use of networks, and parts of the Node class are left as exercises. However, they are sufficient to allow you to test the methods you write. Study them carefully to reinforce your understanding of instance variables and integers.

A standard technique in software development is to develop a **prototype** of a system that sort of fakes the functionality of the real thing, for purposes of seeing how it looks and feels. Then you toss it when you write the real thing. These three classes are examples of such prototypes.

The Position methods

In this prototype implementation, a Position object keeps track of the id number of the Node at its current position in its list. When you call `getNext`, it returns a Node object with that id. The Node returned will be equal to any other Node object with the same id, because the `equals` method returns `true` if and only if the executor Node has the same id number as the parameter Node. Nodes are number from 0 to 99 inclusive, and `itsCurrent` may be greater than 99, so calculating the current node's number may require subtracting 100.

A Position object also keeps track of the id number of the last node in its list of nodes, so it knows when it has gone to far. So `getNext` returns `null` when its current node id is beyond its last node id, and `hasNext` simply verifies that its current node is not beyond its last node. The `moveOn` method adds 1 to the id number for its current node. These methods are coded in Listing 5.10. Listing 5.11 provides the corresponding definition of the other two classes.

Exercise 5.37 Rewrite the `allReachableFrom` method to have just one statement subordinate to the for-loop, an if-statement, and thus not have braces around it.
Exercise 5.38 Continue the trace of the algorithm in Figure 5.5 for two more executions of Step 3a.
Exercise 5.39 Write the `isBlue` Node method (an odd `itsID` means it is blue) and the `getName` Node method (every Node is named "Darryl") for Listing 5.11.
Exercise 5.40 Write the `setMark` and `getMark` Node methods for Listing 5.11. Add an extra instance variable named `itsMark` to do this.
Exercise 5.41 In the Node class of Listing 5.11, which Nodes does Node #6 connect to? Node #42? Node #97?
Exercise 5.42* Revise Listings 5.10 and 5.11 so each Node connects to five other Nodes of the opposite color (an odd `itsID` means it is blue). Hint: Have #18 connect to #19, #21, #23, #25, #27.

Listing 5.10 Prototype Position class of objects

```
public class Position extends Object
{
   private int itsCurrent; // Node at current position on list
   private int itsLast;    // Node at last position on list

   public Position (int first, int last)
   {  super();
      itsCurrent = first;
      itsLast = last;
   }  //====================

   public Node getNext()
   {  if (itsCurrent > itsLast)
         return null;
      else
         return new Node (itsCurrent % Network.NUM_NODES);
   }  //====================

   public boolean hasNext()
   {  return itsCurrent <= itsLast;
   }  //====================

   public void moveOn()
   {  itsCurrent++;
   }  //====================
}
```

Listing 5.11 Prototype Network and Node classes of objects

```
public class Network extends Object
{
   public static final int NUM_NODES = 100;
   ////////////////////////////////////

   public Position nodes()
   {  return new Position (0, NUM_NODES - 1);
   }  //====================
}
//#################################################################

public class Node extends Object
{
   private int itsID;     // ranges from 0 to NUM_NODES - 1

   public Node (int index)
   {  super();
      itsID = index;
   }  //====================

   public Position nodes()
   {  return new Position (itsID + 1, itsID + 4);
   }  //====================

   public boolean equals (Node par)
   {  return this.itsID == par.itsID;
   }  //====================
}
```

*5.9 Recursion (*Enrichment)*

The following method definition is at the beginning of Chapter Three. It uses a while-statement to put a CD in each slot of the executor's sequence of slots, by moving one slot forward each time until there are no more slots to fill:

```
public void fillSlots()
{  while (seesSlot())
   {  putCD();
      moveOn();
   }
}  //======================
```

When you think about what a while-statement means, you can see this is the same logic as the following. Of course, that last line is a comment instead of a Java statement, so the effect is not the same. But it describes what the while-statement does.

```
public void fillSlots()
{  if (seesSlot())
   {  putCD();
      moveOn();
      // repeat this if-statement
   }
}  //======================
```

This logic can be expressed a third way. The comment has been replaced by a call of the method that contains the if-statement. This is called **recursion**.

```
public void fillSlots()
{  if (seesSlot())
   {  putCD();
      moveOn();
      fillSlots();  // i.e., repeat this if-statement
   }
}  //======================
```

Execution does not go on forever for any of these logics. For each of them, the executor moves one slot forward each time through the loop. Eventually it comes to the end of the sequence. Then the `seesSlot()` condition is `false` and execution stops.

Recursive version of fillSlots(int)

Consider this task: We want to fill in the first four slots in a sequence, or the first six slots or whatever is specified in a variable `numToFill`. Afterward we want to back up to the starting position. But if there are less than `numToFill` in the sequence, we just fill in all there are and then back up to the original position.

That logic can be written fairly clearly using recursion: When `numToFill` is positive and you see a slot, then you can fill `numToFill` slots if you (a) put a CD in the first slot, then (b) move on, then (c) fill `numToFill-1` slots, then (d) back up by one slot. The following is a line-for-line translation of this logic:

```
public void fillSlots (int numToFill)
{  if (seesSlot() && numToFill > 0)
   {  putCD();                              // (a)
      moveOn();                             // (b)
      fillSlots (numToFill - 1);            // (c)
      backUp();                             // (d)
   }
}  //=======================
```

The recursive logic makes it unnecessary to keep track of the starting position in the sequence. The next-to-last statement in the method just means: Repeat this if-statement but with `numToFill` having a value 1 less than the previous time.

Recursive version of getNumSlots()

The `getNumSlots` method in Section 4.5 had the executor move down its sequence, adding 1 to a counter for each slot it saw. When it came to the end, it backed up to the starting point (because it is a query method) and then returned the final value of the counter. A recursive solution to the request to count your slots and report how many you have is to (a) report zero if you have no slots, otherwise (b) move on to the next slot and count how many there are from that point on, then (c) back up one slot and report 1 more than you found in step (b). The coding is as follows:

```
public int getNumSlots()
{  if ( ! seesSlot())                       // (a)
      return 0;                             // (a)
   moveOn();                                // (b)
   int num = getNumSlots();                 // (b)
   backUp();                                // (c)
   return 1 + num;                          // (c)
}  //=======================
```

Figure 5.6 should give you some idea of how recursion works for the method call `sam.getNumSlots()` with two slots left in `sam's` sequence. The key is, each method call has the runtime system create a new Method object to carry out the process given by the method definition. Think of it this way: The method definition for `getNumSlots` is a college course that Method objects can take to learn how to do something. Calling the method when `sam` has two slots left has the runtime system create a graduate of that course, labeled #1 in the figure, who is to carry out the process studied in the course.

Part of the process Method object #1 carries out is a method call that creates a totally different graduate of the course, labeled #2 in the figure, to carry out that same process when the sequence has one slot. Part of the process Method object #2 carries out is a method call that creates a third graduate of the course, labeled #3 in the figure, to carry out the same process when the sequence has no slots. So #3 returns 0 to #2, who stores 0 in his own variable `num`, thus returns 1 to #1, who stores the 1 in his own totally separate variable `num`, and thus returns 2 to the point where it was originally called.

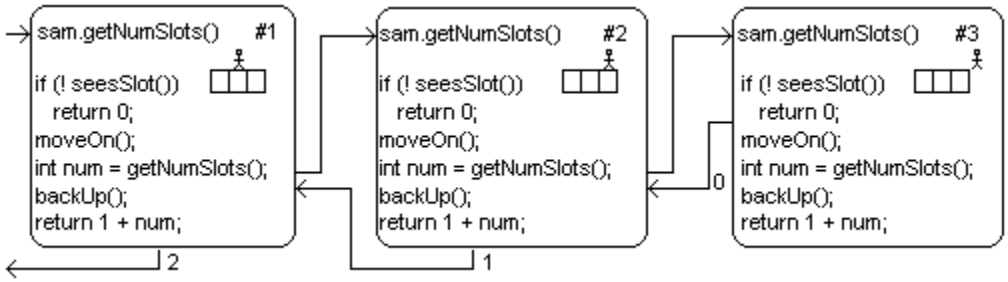


Figure 5.6 Call of getNumSlots with two additional recursive calls

People new to recursion sometimes ask, "How can a method call itself?" The answer is, "That doesn't happen; what happens is, one Method object calls a totally different Method object that graduated from the same course."

Turing Machine programs, as described at the end of Chapter Three, conventionally use recursion to the exclusion of while-statements. With recursion, methods in a subclass of the Tum class can always be coded as a single multi-selection statement.

Recursion applied to Networks

The `allReachableFrom` method in the earlier Listing 5.9 for Networks is much easier and much more efficient when you use recursion. You can eliminate all but the first and last statements of the `allReachableFrom` method if you just replace one statement in the `checkOut` method, as shown in Listing 5.12. Specifically, checking out a node `par` consists in marking it 2, then checking out every node marked 0 you can reach directly from `par`. In effect, to mark all the nodes you can reach from `par`, you first mark `par` and then you simply mark all the nodes you can reach from a node you can reach from `par`.

Listing 5.12 Recursive allReachableFrom method in SmartNet

```
/** Rewrite of SmartNet's allReachableFrom for efficiency. */

public boolean allReachableFrom (Node source)
{  checkOut (source);
   // all of these lines have been deleted
   return allNodesAreMarked();
}  //=====================

private static void checkOut (Node par)
{  par.setMark (2);
   for (Position p = par.nodes();  p.hasNext();  p.moveOn())
   {  Node current = p.getNext();
      if (current.getMark() == 0)
         checkOut (current);  // this is the only line changed
   }
}  //=====================
```

The **Traveling Salesman Problem** for a network of airports is to answer the question of whether you can visit every airport without going through the same airport twice, starting from a given airport. A general solution can be obtained by calling `canTravelFrom (givenAirport, getNumNodes())` for the `canTravelFrom` method in Listing 5.13. This method is based on the logic in the accompanying design block.

Question: Can one travel through n different nodes marked zero starting from base?
Answer: If n is 1, then...
Of course it is possible, since base itself is the one.
Otherwise...
Mark 1 on base.
If there is any node such that
(a) you can reach it from base in one direct step, and
(b) it is marked zero, and
(c) you can travel through n-1 different nodes marked zero starting with it, then...
It is obviously possible.
Otherwise...
It is not possible.

Listing 5.13 Recursive NodeOp method for the Traveling Salesman

```
/** Tell whether it is possible to travel through n nodes,
 *  all different and all marked 0, starting from Node base.
 *  Precondition:  base is marked 0.  */

public static boolean canTravelFrom (Node base, int n)
{  if (n <= 1)
      return true;
   base.setMark (1);
   for (Position p = base.nodes();  p.hasNext();  p.moveOn())
   {  Node current = p.getNext();
      if (current.getMark() == 0
                  && canTravelFrom (current, n - 1))
      {  base.setMark (0);   // restore original state
         return true;
      }
   }
   base.setMark (0);         // restore original state
   return false;
}  //=====================
```

Language elements
A method can call any method in its class, even itself.

Exercise 5.43 Rewrite the `fillSlots` method in Listing 3.4 recursively, where the executor returns to its original position.
Exercise 5.44 Rewrite the `seesAllFilled` method in Listing 3.5 recursively.
Exercise 5.45* Rewrite the `clearSlotsToStack` method in Listing 3.4 recursively.
Exercise 5.46* Rewrite the `hasAsManySlotsAs` method in Listing 3.6 recursively.
Exercise 5.47** Rewrite the `lastEmptySlot` method in Listing 3.8 recursively.
Exercise 5.48** Write a recursive method `public boolean canFillAllSlots()` for a subclass of Vic: The executor tells whether there are enough CDs in the stack to fill all of its empty slots. When the method terminates, the executor must be in the same state it was in when the method began. Note that a non-recursive solution is too hard.

*5.10 More On JOptionPane (*Sun Library)*

This section describes methods from the `javax.swing.JOptionPane` class that can be quite useful for major projects or in later courses, though they are not mentioned elsewhere in this book. Look at your documentation to see additional possibilities (a folder named something like `jdk1.3\docs\api\javax\swing\JOptionPane.html` on your hard disk).

The `showMessageDialog` method call has the following more general form:

```
showMessageDialog (null, someMessage, "title at the top",
                   JOptionPane.someIntName)
```

The first parameter is a Component value; if it is not `null`, the dialog is displayed in the frame for that Component object and usually positioned directly below the Component.

The second parameter `someMessage` is typically a String to be displayed, on several lines if it includes the newline character `'\n'`. However, the parameter type is specified as Object, so you may make it any of several kinds of displayable objects.

The third parameter is a String value to replace the default title "Confirm". For the fourth parameter, replace `someIntName` by one of the following to specify the icon:

- PLAIN_MESSAGE (no icon at all)
- ERROR_MESSAGE (a horizontal bar inside an octagon)
- WARNING_MESSAGE (an "!" inside a triangle)
- QUESTION_MESSAGE (a "?" inside a rectangle)
- INFORMATION_MESSAGE (an "i" inside a circle, which is the default icon)

The `showInputDialog` method call has the default title "Input", the "?" icon, and two buttons saying "OK" and "Cancel". It is also overloaded with a four-parameter version having the same four parameters with the same meaning as for `showMessageDialog`.

The `showConfirmDialog` can return any of several int values to tell what button the user clicked: YES_OPTION, NO_OPTION, OK_OPTION, CANCEL_OPTION, and CLOSED_OPTION (meaning that the user clicked the X-shaped closer icon in the top right of the window). The name `showConfirmDialog` is also overloaded; the more general form of it has two variants:

```
showConfirmDialog (null, messageString, titleString,
                   JOptionPane.YES_NO_OPTION)
showConfirmDialog (null, messageString, titleString,
                   JOptionPane.YES_NO_CANCEL_OPTION)
```

The first three parameters are as for `showMessageDialog` (the default title here is "Select an option"). The fourth parameter names an int value that specifies what buttons (either two or three of them) are displayed for the user to click. The default option (when only the first two parameters are used) is the YES_NO_CANCEL_OPTION.

5.11 Review Of Chapter Five

Listing 5.2 and Listing 5.3 illustrate almost all Java language features introduced in this chapter other than recursion.

About the Java language:

- You may call a **class method** (declared using `static`) with the name of its class in place of the executor. By contrast, an instance method requires a reference to an object of the class as the executor.
- You can use `this` inside an instance method as a reference to the executor of the method call. By contrast, a class method does not have an executor, so you cannot use `this` inside a class method.
- If you call a method without an executor and without a class name in place of the executor, the compiler uses the **default**. For calling a class method, the default is the class containing the method call. For calling an instance method, the default is `this` of the method containing the method call (i.e., it is `this` instance of the class).
- You may declare a variable in a class outside of any method, which makes it a **field variable**. Then each object of the class has its own value for the variable if it is an instance variable. But there is only one value for the whole class if it is a **class variable** (i.e., declared with the `static` modifier).
- If the declaration of a field variable assigns it a value, that takes effect for an object's instance variable when the constructor is called, for a class variable when the program begins. The runtime system supplies a default value if you do not: zero or `null` or `false`, as appropriate.
- The word **final** is allowed in any kind of variable declaration. It means that the first assignment of a value to that variable is the last one.
- If a method call returns an object, it can be used as the executor of another method call. This is a cascading or **chaining** of method calls.
- `someString.length()` returns the number of characters in the `someString`.
- `someString.substring(startInt, endInt)` returns the substring of `someString` running from position `startInt` to just before position `endInt`. Numbering is **zero-based** (starts from zero). You must have `0 <= startInt <= endInt <= someString.length()`
- In the grammar summary of Figure 5.7, the `Type` is int, boolean, or a ClassName; `ParameterList` is one or more `Type VariableName` combinations separated by commas, with no assignments to those parameters; and the `Modifier` is either `public` or `private`. Italicized words are optional elements.

`Modifier static` *`final`* `  Type VariableName = Expression;`	**declaration** of class variable with initial value

`Modifier static Type MethodName`
`  (`*`ParameterList`*`)  { StatementGroup }`
`Modifier static void MethodName`
`  (`*`ParameterList`*`)  { StatementGroup }`
	declaration of class method that accepts input initially assigned to its formal parameters

Figure 5.7 Declarations added in Chapter Five

Other vocabulary to remember:

- If a class does not have any instance methods or instance variables or main method, we call it an **utilities class**. If it has instance methods or instance variables and no main method, we call it an **object class**. The only other kind of class we use in this book is a class that has a main method and no other method, called an **application program**. A class method is **independent** if it could be in any class.
- String values are **immutable**, i.e., the attributes of these objects cannot be changed.

Answers to Selected Exercises

5.1 Insert before the first if: if (count == 0) return 0;

5.2 Insert before the for-statement: if (expo > 30) expo = 30;

5.3
```
public static int power (int base, int expo)
{   if (base <= 0 || expo < 0)
        return 0;
    int limit = 2147483647 / base;
    int power = 1;
    for ( ; expo > 0 && power <= limit;  expo--)
        power = power * base;
    if (expo == 0)
        return power;
    else
        return -1;
}
```

5.7 Put this outside of any method: private static String theLatestName = "none so far";
Put this statement at the end of the coding for the constructor: theLatestName = itsFirstName;
Add this method to the Person class:
```
public static int getNameOfLatestCreated()
{   return theLatestName;
}
```

5.8
```
java.util.Random randy = new java.util.Random();
public static int range (int one, int two)
{   if (one <= two)
        return one + randy.nextInt (two - one + 1);
    else
        return two + randy.nextInt (one - two + 1);
}
```

5.11 Put this declaration as the first statement of toString: final int BASE = 10;
Replace the phrase "itsHour < 10" by "itsHour < BASE".

5.14
```
private static char getSub (String sequence, int position)
{   return sequence.substring (position, position + 1);
}
```
Replace the following four parts of Listings 5.3-5.5:
```
return ! getSub (itsSequence, itsPos).equals (NONE);  // last 2 statements of seesCD
theStack = theStack + getSub (itsSequence, itsPos);  // statement in takeCD
+ getSub (itsStack, atEnd)  // fourth line of the body of putCD
itsSequence = getSub (LETTERS, k) + itsSequence;  // in the constructor
```

5.16 The new set of strings replaces the old set of strings provided by the previous call of reset.
This is no actual change in values, assuming that the same args was used each time.

5.17 Replace "Vic# " in the trace method by theStack + "; Vic# ".

5.21
```
public static void main (String[ ] args)
{   Network airline = new Network();
    String lastBlue = "no blues";
    for (Position pos = airline.nodes();  pos.hasNext();  pos.moveOn())
        if (pos.getNext().isBlue())
            lastBlue = pos.getNext().getName();
    JOptionPane.showMessageDialog (null, lastBlue);
}
```

5.22 It is the same coding as for seesOnlyBlue except remove the ! operator in the if-condition.

5.23
```
public static int getNumNodes (Node par)
{   int count = 0;
    for (Position p = par.nodes();  p.hasNext();  p.moveOn())
        count++;
    return count;
}
```

5.24
```
public static int bidirectional (Node par)
{   for (Position p = par.nodes();  p.hasNext();  p.moveOn())
        if ( ! connected (p.getNext(), par)
            return false;
    return true;
}
```
5.28 Replace the line beginning with "if" by:
```
if ( ! current.equals (par) && ! NodeOp.connected (current, par))
```
5.29
```
public boolean noLoners()
{   for (Position pos = nodes(); pos.hasNext();  pos.moveOn())
        if ( ! pos.getNext().nodes().hasNext())
            return false;
    return true;
}
```
5.30
```
public boolean atLeastOneNonBlue()
{   for (Position pos = nodes(); pos.hasNext();  pos.moveOn())
        if ( ! pos.getNext().isBlue())
            return true;
    return false;
}
```
5.37 Replace the for-statement by the following:
```
for (Position pos = nodes();  pos.hasNext();  pos.moveOn())
    if (pos.getNext().getMark() == 1)
    {   foundNodeToCheckOut = true;
        process (pos.getNext());
    }
```
5.38 Next after G, H will be checked out, which changes its 1 to 2. Next, the boolean variable is tested and the do-while repeats. E is checked out, which changes its 1 to 2 and its 0 to 1.

5.39
```
public boolean isBlue()
{   return itsID % 2 == 1;
}
public String getName()
{   return "Darryl";
}
```
5.40
```
private int itsMark = 0;
public void setMark (int newMark)
{   itsMark = newMark;
}
public int getMark()
{   return itsMark;
}
```
5.41 Node #6 connects to Nodes 7, 8, 9, 10.
Node #42 connects to Nodes 43, 44, 45, 46.
Node #97 connects to Nodes 98, 99, 0, 1.

5.43
```
public void fillSlots()
{   if (seesSlot())
    {   putCD();
        moveOn();
        fillSlots();
        backUp();
    }
}
```
5.44
```
public boolean seesAllFilled()
{   if ( ! seesSlot())
        return true;
    else if ( ! seesCD())
        return false;
    moveOn();
    boolean valueToReturn = seesAllFilled();
    backUp();
    return valueToReturn;
}
```

Review: Overall Java Language So Far

This summary presents a description of all of the language elements seen so far except for the `String[] args` in a main method heading. It even includes some elements from the next chapter relating to `double`. Several special notations are used to make the descriptions compact yet reasonably clear:

- A phrase in italics is optional.
- Words beginning with a small letter are reserved words (the keywords plus `true`, `false`, and `null`); they must be written exactly as is.
- Words beginning with a capital letter and ending in "Name" can be replaced by any identifier of a class, method, or variable as indicated. An identifier is a name that the writer of the definition makes up. It can be made up of letters, digits, and underscores. It cannot start with a digit and it cannot be a reserved word.
- Other words beginning with a capital letter represent phrases defined elsewhere in this section to be any of several different things.

A **CompilationFile** is a file you may compile in Java, structured as shown below. If it has the optional `extends` phrase, then every public declaration in the superclass named is indirectly in the defined class by inheritance, except for constructors and those declarations that the subclass overrides by giving a new declaration with the same signature (i.e., the same name and the same parameter structure).

```
ImportDirectives
public  class  ClassName  extends SuperclassName
{  DeclarationGroup
}
```

You may have import directives in a compilable file, as long as they come before any class definition in the file. The **ImportDirectives** consist of one or more lines structured as shown below. The first allows the use of the named class from another package. The second allows the use of any class from that other package. The **PackageName** is several identifiers separated by dots, e.g., `javax.swing` and `java.awt.event`.

```
import PackageName . ClassName ;
import PackageName . * ;
```

A **DeclarationGroup** is any number of consecutive Declarations. A **Declaration** can be one of the five forms listed below (lines beginning with a left brace are a continuation of the preceding line). The first two are field variable declarations; the last three are method declarations. If the assignment to the variable is not present, the variable is initialized to zero, `null`, or `false` (whichever is appropriate). Methods defined in the same class can have the same MethodName if they have different signatures. For the constructor declaration (listed last), the ClassName has to be that of the class the constructor is in. You may omit `super(...);` which will then default to `super();` .

```
ModifierPhrase  Type  VariableName ;
ModifierPhrase  Type  VariableName = Expression ;
ModifierPhrase  Type  MethodName ( ParameterList )
   { StatementGroup }
ModifierPhrase  void  MethodName ( ParameterList )
   { StatementGroup }
public  ClassName ( ParameterList )
   { super ( ArgumentList ) ;  StatementGroup }
```

Examples An object of the following class represents a TV show that has a ranking on a scale of 1 to 10. The top line is an import directive whose PackageName is java.util (a package containing Random). The TVShow class has two declarations of VariableNames with assignments and two without. These variable declarations have three different ModifierPhrases: "public static final", "private static final", and "private". The Type is int in two cases; it is Random and String in the other cases. The TVShow class also has a declaration of a constructor with the ParameterList "String name, int rank" and two declarations of non-constructor methods.

```
import java.util.Random;

public class TVShow extends Object
{
   public static final int MAX_RANK = 10;  // two class variables
   private static final Random randy = new Random();
   ////////////////////////////
   private String itsName;              // two instance variables
   private int itsRank;


   /** Create an object for the given name and rank. */

   public TVShow (String name, int rank)           // constructor
   {  super();
      itsName = name;
      if (rank >= 1 && rank <= MAX_RANK)
         itsRank = rank;
      else
         itsRank = 1;
   }  //======================

   /** Return a new TVShow object with the given name but a
    *  randomly chosen value for its rank. */

   public static TVShow randomShow (String name)  // class method
   {  return new TVShow (name, 1 + randy.nextInt (MAX_RANK));
   }  //======================


   /** Return the rank for this particular TVShow object. */

   public int getRank()                         // instance method
   {  return itsRank;
   }  //======================
}
```

A **ModifierPhrase** can have one of the following four forms. The modifier `public` says any other class can access the name; `private` says only definitions within the current class can access the name; `final` says it cannot be changed later (overridden or reassigned); and `static` says access can be with the class name in place of an instance of the class (thus it is a class method or a class variable). A non-constructor declaration without the `static` modifier is an instance method or instance variable.

```
public static final
private static final
public final
private final
```

The only **Type** discussed through the first part of Chapter Six is one of the following. The boolean type is for values that are either `true` or `false`. The int type is for numbers without decimal points. The double type is for numbers that have a decimal point and digits after it. The ClassName is an object type.

```
boolean
int
double
PackageName.  ClassName
```

An **ArgumentList**, used in method calls, consists of one or more Expressions with commas between two consecutive expressions. So it has one of the following two forms.

```
Expression
Expression , ArgumentList
```

Examples The following are statements containing method calls with 2, 1, and 0 arguments, respectively.

```
TVShow t1 = new TVShow ("Law & Order", 9);
TVShow t2 = TVShow.randomShow ("NYPD Blue");
System.out.println (t2.getRank());
```

A **ParameterList**, used in method headings, describes the kinds of arguments that must be passed as input to the execution of the method. It has one of the following two forms.

```
Type VariableName
Type VariableName , ParameterList
```

Examples The following method headings in the TVShow class have 2, 1, and 0 parameters, respectively.

```
public TVShow (String name, int rank)
public static TVShow randomShow (String name)
public int getRank()
```

A **StatementGroup** is any number of consecutive Statements. A **Statement** can be any of the following nine forms. If an `else` could be matched with more than one `if`, then the `else` is to be matched with the most recent such `if`. The parentheses shown after `if`, `while`, and `for` are not optional.

```
Type VariableName ;
MethodCall ;
Initializer ;
return Expression ;
{  StatementGroup  }
if ( Condition )  Statement
if ( Condition )  Statement  else  Statement
while ( Condition )  Statement
do  Statement  while ( Condition ) ;
for ( Initializer ; Condition ; Updater )  Statement
```

An **Initializer** is something that can be used in the first part of a for-statement, to assign a value to the loop control variable at the beginning of the looping process. It can have any of the following four forms.

```
Type VariableReference = Expression
VariableReference = Expression
VariableReference ++
VariableReference --
```

An **Updater** is something that changes the value of the loop control variable, as follows.

```
MethodCall
VariableReference = Expression
VariableReference ++
VariableReference --
```

Examples The following are possible for-statement headings.

```
for (int k = 5;  k < size;  k++)
for (k--;  k > 0;  k--)
for (itsMin = min;  itsMin < 0;  itsMin = itsMin + 60)
for (p = nodes;  p.hasNext();  p.moveOn())
```

An **Expression** is a phrase for which the runtime system can compute a value. It has one of the following three forms.

```
ObjectExpression
NumericExpression
Condition
```

An **ObjectExpression** has one of the following forms. The `this` reference is only allowed within an instance method or constructor. The VariableReference must be an object type and the MethodCall must return an object type.

```
this
null
VariableReference
MethodCall
StringValue
new ClassName ( ArgumentList )
```

A **VariableReference** has one of the following forms. Option: You may omit `this.` or `ClassName.` at the beginning of a VariableReference, to have it default to the executor of the current instance method or to the class the VariableReference is in, respectively.

```
VariableName
PackageName.  ClassName . VariableName
this . super . VariableName
ObjectExpression . VariableName
```

Examples The following statements assign various kinds of ObjectExpressions to variables.

```
TVShow a = null;                // a refers to no object
Object b = System.out;          // assign a VariableReference
String c = sam.getPosition();   // assign a MethodCall
String u = "Friends";           // assign a StringValue
TVShow x = new TVShow ("24", 8); // assign a new value
```

A **MethodCall** has one of the following forms. Option: You may omit `this.` or `ClassName.` at the beginning of a MethodCall, to have it default to the executor of the current instance method or to the class the MethodCall is in, respectively.

```
PackageName.  ClassName . MethodName ( ArgumentList )
this . super . MethodName ( ArgumentList )
ObjectExpression . MethodName ( ArgumentList )
```

A **NumericExpression** has one of the following forms. The MethodCall must return an int or double type and the VariableReference must be an int or double type. The last form simply means that, if you already have a NumericExpression, you can put parentheses around it and it will still be a NumericExpression.

```
IntegerLiteral
DoubleLiteral
VariableReference
MethodCall
NumericExpression * NumericExpression
NumericExpression / NumericExpression
NumericExpression % NumericExpression
NumericExpression + NumericExpression
NumericExpression - NumericExpression
( NumericExpression )
```

A **Condition** is a kind of Expression. It has one of the following forms. The MethodCall must return a boolean type and the VariableReference must be a boolean type.

```
true
false
VariableReference
MethodCall
ComparisonOfTwoValues
! Condition
Condition && Condition
Condition || Condition
( Condition )
```

Examples The following assign various NumericExpressions and Conditions to variables.

```
int e = 47;               // assign an IntegerLiteral
int f = e;                // assign a VariableReference
int k = x.getRank();      // assign a MethodCall
int m = f * (e - 5);      // assign result of operator
boolean p = true;         // assign a boolean literal
boolean q = s.equals (t); // assign a MethodCall
boolean r = e <= f;       // assign a ComparisonOfTwoValues
boolean ok = p && ! q;    // assign result of operator
```

A **ComparisonOfTwoValues** is a special kind of Condition. It has one of the following eight forms.

```
NumericExpression <  NumericExpression
NumericExpression <= NumericExpression
NumericExpression >  NumericExpression
NumericExpression >= NumericExpression
NumericExpression == NumericExpression
NumericExpression != NumericExpression
ObjectExpression == ObjectExpression
ObjectExpression != ObjectExpression
```

A **StringValue** has one of the following foms. The only operator that can be used with object references is the plus sign, where at least one of the operands is a String value. It concatenates the two String values. If the other operand is a numeric value, the plus sign concatenates the string of characters that form the numeral with the string of characters in the String value.

```
StringLiteral
VariableReference
MethodCall
StringValue + StringValue
StringValue + NumericExpression
NumericExpression + StringValue
```

A **StringLiteral** is a pair of quotes containing any characters other than a backslash or a quote, except you use one of the \n or \\ or \b or \t combinations.

An **IntegerLiteral** is a sequence of one or more digits, optionally preceded by a negative sign.

A **DoubleLiteral** is a sequence of one or more digits, then a decimal point, then a sequence of one or more digits. The whole is optionally preceded by a negative sign.

The twenty **reserved words** seen so far in this book are the following (`true`, `false`, and `null` are technically not keywords). Those in the first line can only be used outside of a method body, and those in the last line can only be used inside a method body.

```
private  public  class  extends  static  void
int  boolean  new  true  false  null
if  else  while  do  for  return  this  super
```

The twelve **Sun standard library** methods seen so far in this book are the following.

```
Object:       equals (someObject).
String:       equals (someString),
              length(),
              substring (start, end).
Random:       new Random(),
              nextInt (limitInt).
System:       System.exit (0),
              System.out.println (someString).
Integer:      Integer.parseInt (someString).
JOptionPane:  JOptionPane.showMessageDialog (null, someString),
              JOptionPane.showInputDialog (promptString),
              JOptionPane.showConfirmDialog (null, someString).
```

Examples The main method of the following class can be executed from the command line using `java Puzzler`. It illustrates the use of some of these library methods.

```
public class Puzzler
{
   public static void main (String[] args)
   {  String s = JOptionPane.showInputDialog ("How many?");
      int number = Integer.parseInt (s);
      for (int k = 0;  k < s.length();  k++)
         System.out.println (s.substring (0, k));
      JOptionPane.showMessageDialog (null, "all done");
      System.exit (0);
   }  //======================
}
```

6 Basic Data Types and Expressions

Overview

In this chapter you will learn about decimal number values, character values, long values, and more String methods, to help develop software for managing a car repair shop. You will also have a light introduction to graphical components. The first five sections complete the coverage of all the language features you need for the first half or so of each of Chapters Eight through Twelve. In particular, if you want to use disk files for input and output, you can read and understand the first three sections of Chapter Ten and of Chapter Twelve on disk files after you complete Section 6.5.

- Sections 6.1-6.2 discuss the double type of value (numbers with decimal points).
- Sections 6.3-6.4 introduce more String methods using chars (character values).
- Sections 6.5-6.6 give some details on conversions and casts between types of values, as well as an introduction to the Math class and JTextArea objects.
- Sections 6.7-6.8 complete the development of the RepairShop software, with the main application class depending on six other classes: Queue, IO, RepairOrder, View, String, and System (only the last two are from the Sun standard library).

6.1 Double Values, Variables, And Expressions

Suppose you have a job developing a program to help a car repair shop schedule each day's repair jobs. The shop accepts appointments and walk-ins on a first-come-first-served basis. The shop foreman enters each repair job into the computer as it is booked, putting it at the end of a list of such jobs. When a time slot becomes available to work on a car, the next job at the front of the list is removed from the list and worked on. Walk-ins during the day are added to the end of the list. This kind of list is called a **queue**.

Your RepairShop program is to report on the total number of jobs waiting and the total estimated time to complete those jobs. An updated report is to be made after each change in the list. So this software will deal with several different kinds of objects, including:

1. A virtual repair job, representing a single work order.
2. A virtual input device, representing the keyboard.
3. A virtual output device, representing the screen.
4. A virtual queue, storing data for several repair jobs on a first-in-first-out basis.

You will need to store the number of hours for a single work order as a decimal number, e.g., 1.25 hours for 1 hour 15 minutes. And you will need to read in a line of input containing several words and separate out each word from the rest. So you need more information on working with decimal numbers and Strings. This will be supplied in the next few sections, then we will come back to the RepairShop software and complete it.

Decimal Numbers

You may declare a variable as type **double**. It can then hold decimal numbers such as 53.172 and -0.00005. Java sets aside a 64-bit storage space for its value, stored in scientific notation. That is enough space for 15-decimal-place precision, with some space left over to store an exponent of 10 up to about the 307th power. In short, you can store a 306-digit number with 15-digit precision. The five numeric operators for doubles are `+` for plus, `*` for times, `-` for minus, `/` for divided-by, and `%` for the remainder.

For example, `13.0 / 4.0` is 3.25, the result of "long division". `8.0 % 2.5` is 0.5 and `(-8.0) % 2.5` is -0.5, similar to int values.

The six comparison operators can be used for both int values and double values: The expression `x < y` means that x is less than y. Similarly, `x > y` means that x is greater than y, `x <= y` means that x is less than or equal to y, and `x >= y` means that x is greater than or equal to y. `x != y` means that x is not equal to y and `x == y` means that x is equal to y (the double equals sign is needed to distinguish it from the assignment operator).

If you want to obtain a double value from the user, you can convert the string of characters `s` that `showInputDialog` returns to a double value with the following statement. It uses the `parseDouble` class method from the Double class in the `java.lang` package, which is analogous to `Integer.parseInt`. This class method accepts either normal decimal form (e.g., -4176000.0 or 0.000000275) or **scientific notation** (their equivalents -4.176E+6 or 2.75E-7). Note that the 'E' in this form stands for "times ten to the power": 6.3E+4 is 6.3 times ten to the power 4, i.e., 63000.

```
double x = Double.parseDouble (s);
```

The program can crash if the user's input contains letters or is otherwise ill-formed. However, the `Double.parseDouble` method (which is new with Java version 1.2) will tolerate blanks before the numeral, whereas `Integer.parseInt` will not.

Promotions and casts

If you combine an int value with a double value using an operator, the runtime system automatically **promotes** the whole number to the corresponding decimal form. The same thing happens if you assign an int value to a double variable. So if you have two int variables `x` and `y`, then `(1.0 * x) / y` gives a precise answer for the quotient.

You cannot assign a double value to an int variable without saying that you are doing it. You do this with a **cast**, which is the type name in parentheses. For instance, if `dub` is a double variable and `ent` is an int variable, then `dub = ent` is legal, but `ent = dub` is not legal. However, `ent = (int) dub` is legal. This assignment discards the part after the decimal point in `dub`'s value (3.8 is changed to 3, and -3.8 is changed to -3). In general, you are allowed to make a cast from any numeric value to any other kind of numeric value, but you may lose some of the value.

Sentinel logic

A standard pattern for user interaction is to ask for a particular kind of value over and over again, using that value in a new calculation. When the user wants to stop the program, the user enters a special signal value such as 0 or -1 that cannot occur as a value to use in the calculation. Such a signal is called a **sentinel value**.

The GrowthRates application program in Listing 6.1 illustrates this **sentinel-controlled loop pattern**. It asks the user for an interest rate (such as 6% or 15%, but without the percent sign). Then it calculates how many years it would take for money to grow at that interest rate to be twice as much and reports the answer. The sentinel values are all the nonpositive numbers (i.e., an "interest rate" that is not positive terminates the program).

The computation of the number of years it takes to double the money is independent of any object, since it works only with numbers. The MathOp class in the earlier Listing 5.1 is intended to collect together class methods that work with numbers. So this computation of the doubling time should be a method in the MathOp utilities class.

Listing 6.1 Application program using doubles

```
import javax.swing.JOptionPane;

public class GrowthRates
{
   /** Calculate time to double your money at a given rate. */

   public static void main (String[ ] args)
   {  JOptionPane.showMessageDialog (null,
                "Calculating growth for various interest rates");
      String input = JOptionPane.showInputDialog
                ("Annual rate? 0 if done:");
      double rate = Double.parseDouble (input);
      while (rate > 0.0)
      {  JOptionPane.showMessageDialog (null,
                 "It takes " + MathOp.yearsToDouble (rate)
                 + " years for \nyour money to double.");
         input = JOptionPane.showInputDialog
                ("Another rate (0 when done):");
         rate = Double.parseDouble (input);
      }
      System.exit (0);
   }  //=====================
}

// a method for the MathOp utilities class
   /** Precondition:  interestRate is positive. */

   public static int yearsToDouble (double interestRate)
   {  double balance = 1.0;
      int count = 0;
      while (balance < 2.0)
      {  balance = balance * (1.0 + interestRate / 100.0);
         count++;
      }
      return count;
   }  //=====================
```

The calculation requires repeatedly multiplying one year's balance by 1.06 if the `rate` is 6%, by 1.15 if the `rate` is 15%, etc. The command `count++` means that `count` is increased by 1. This logic illustrates the common **count-cases looping action**: If you initialize a counter variable to zero before the loop begins, and you increment it each time through the loop, then its value when you exit the loop will be the number of times that the body of the loop was executed.

The following is the sequence of phrases you will see on the screen when you run this program and enter the boldfaced values:

```
Calculating growth for various interest rates
Annual rate? 0 if done: 6
It takes 12 years for your money to double.
Another rate (0 when done): 9.1
It takes 8 years for your money to double.
Another rate (0 when done): 12.3
It takes 6 years for your money to double.
Another rate (0 when done): -2
```

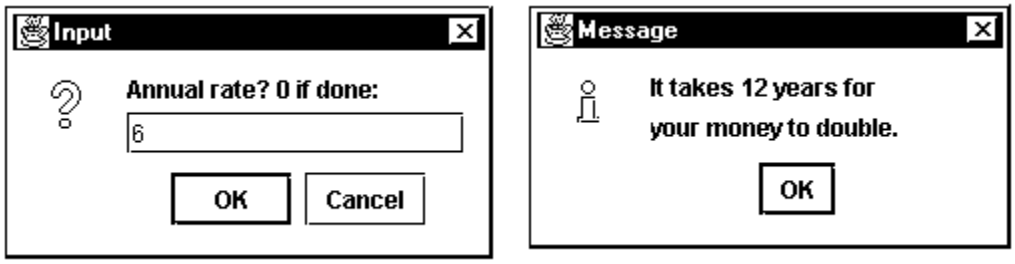


Figure 6.1 SCREEN SHOTS of dialog boxes for GrowthRates

Programming Style When you have a program that interacts with the user by keyboard and screen, it is good style to begin with a line or two on the screen indicating the purpose of the program. That way, the user knows right away if he/she is running the desired program. The first statement of Listing 6.1 illustrates this principle.

Special assignment operators

The assignment to the `balance` variable in Listing 6.1 can be written as follows:

```
balance  *=  1.0 + interestRate / 100.0;
```

This is an example of a **special assignment operator**. You can write **`x *= y`** instead of `x = x * y`. The former executes somewhat faster and seems a little easier to understand once you get used to it. Similarly, **`x += y`** means `x = x + y`, **`x -= y`** means `x = x - y`, **`x /= y`** means `x = x / y`, and **`x %= y`** means `x = x % y`. For instance, the following two statements have exactly the same effect:

```
total = total + someGuy.getBirthYear();
total += someGuy.getBirthYear();
```

Language elements
Five additional assignment operators are +=, -=, *=, /=, and %=.
A numeric type of variable in Java is double. These values are not objects; they are normally written as two sequences of digits separated by a decimal point, with an optional minus sign. You may use the arithmetic, assignment, and comparison operators that ints have.
If you want to assign a double value to an int variable, you must put (int) in front of the double expression. This truncates the value after the decimal point.

Exercise 6.1 Write an application program to ask the user for three decimal numbers and then print their average.
Exercise 6.2 Write an application program to ask the user for the exchange rate for German marks and the German price per liter (0.264 gallons) of gasoline, then say what that is in dollars per gallon.
Exercise 6.3 Find three phrases in the Interlude before Chapter Four that can reasonably be rewritten to use a special assignment operator.
Exercise 6.4 Write an application program to ask the user for a (double) number of days and print the number of seconds in that many days. Also print the number of minutes and the number of hours. Do not put any numeric value greater than 60 in your program.
Exercise 6.5* Revise GrowthRates to compound monthly. For instance, if the rate is 6%, multiply by 1.005 each month, then report the number of years and months it took to double.
Exercise 6.6* Write an application program to ask the user for a (double) temperature in Kelvin degrees and convert it to Fahrenheit. Also print the Fahrenheit temperature at which water is inert. Note: It helps to know that water boils at 373.15 degrees Kelvin, freezes at 273.15 degrees Kelvin, and is inert at absolute zero Kelvin.

Exercise 6.7* Write an application program to ask the user for the ages of three people and then print the age of the youngest.
Exercise 6.8** Write an application program to ask the user for the amount paid for a purchase and also for the amount of money offered in payment. Print out the number of pennies, nickels, dimes, and quarters that the user receives in change. The change should produce the least possible number of coins. But print "can't do that" if the change is not in the range 0 through 99 cents. Hint: `quarters = (offered - paid) / 25`.

6.2 Creating Your Own Library Classes

The JOptionPane method calls and the string-to-number conversions obscure the basic logic of a program. It is difficult enough for a programmer to work out complex logic, as in Listing 6.1 with its two while-statements, and difficult enough for other readers to make sense of the logic, without the intrusion of long phrases that perform a simple task.

Programming Style A broadly-accepted rule of good style in Java is, if you find that several lines of logic appear in several different places in your programs, and if those lines act together to accomplish a single well-defined task, then write a method to contain those lines and use calls of that method instead. Put the method in your personal library of classes if you expect it to be needed in several programs. It is good to treat even one line of logic this way if it is used very frequently and is rather lengthy or arcane.

The repair shop client wants the name of the shop displayed on each message dialog box instead of just "Message". Also, the client does not like the information icon; you are to get rid of it. A variant of the `showMessageDialog` method lets you do these things, but it makes the method call even wordier: You add two parameters, one the title and one having the value `JOptionPane.PLAIN_MESSAGE` (which omits the icon).

If you were to make this change throughout the RepairShop software, and then sell this software to another repair shop, you would have to make many changes in your software. Clearly you need to define methods that encapsulate keyboard input and messages.

The IO class

The solution is to develop a class that adapts the general-purpose JOptionPane methods especially for use with this RepairShop software. You put the `showMessageDialog` call in a method in this library class of your own, which you name perhaps IO. Now you can just change a few words in one place to use it for another repair shop. You also put three kinds of `showInputDialog` method calls in this IO class: one to get the String input and parse it as a double, one to do the same for ints, and one to simply get a String as input. The result is in Listing 6.2 (see next page).

The heading for the `say` method may be puzzling. The `showMessageDialog` permits any displayable Object as its second parameter, not just a String value. You may assign a String value to an Object parameter because String is a subclass of Object (as is every class in Java). You will see the advantage later in this chapter of being able to pass a different kind of displayable Object to the second parameter.

The JOptionPane method that gets a String input returns `null` if the user clicks the Cancel button. This is usually inconvenient. So the `askLine` method returns the empty String in that case. The entire main method of the earlier Listing 6.1 could be written with IO more clearly and compactly as follows (compare the two):

Listing 6.2 The IO class

```
import javax.swing.JOptionPane;

public class IO
{
   /** Display a message to the user of Jo's Repair Shop. */

   public static void say (Object message)
   {  JOptionPane.showMessageDialog (null, message,
                "Jo's Repair Shop", JOptionPane.PLAIN_MESSAGE);
   }  //=====================

   /** Display the prompt to the user; wait for the user to enter
    *  a string of characters; return that String (not null). */

   public static String askLine (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      if (s == null)
         return "";
      else
         return s;
   }  //=====================

   /** Display the prompt to the user; wait for the user to enter
    *  a number; return it, or return -1 if ill-formed.  */

   public static double askDouble (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      return new StringInfo (s).parseDouble (-1);
   }  //=====================

   /** Display the prompt to the user; wait for the user to enter
    *  a whole number; return it, or return -1 if ill-formed.  */

   public static int askInt (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      return (int) new StringInfo (s).parseDouble (-1);
   }  //=====================
}
```

```
   public static void main (String[ ] args)
   {  IO.say ("Calculating growth for various interest rates");
      double rate = IO.askDouble ("Annual rate? 0 if done:");
      while (rate > 0.0)
      {  IO.say ("It takes " + MathOp.yearsToDouble (rate)
                 + " years for \nyour money to multiply by 2.");
         rate = IO.askDouble ("Another rate (0 when done):");
      }
      System.exit (0);
   }  //=====================
```

IO's `askDouble` and `askInt` methods do not use the standard parsing methods that can crash if the numeral is ill-formed. Instead, they use a parsing method developed later in this chapter that returns -1 (or whatever is supplied as the parameter of `parseDouble`) whenever the numeral is ill-formed. The -1 result signals to the calling method that the user made a mistake in keyboard entry when only positive values are expected. This means that the RepairShop software is robust -- as it should be.

The class box for the IO class is in Figure 6.2. In the figure, the words in parentheses describes what kind of values you have to supply to call that method. The word after the colon tells what kind of value the method sends back to you (this is the UML standard notation for the type of value returned by a method, used in a class box; it is not the same notation that Java uses in a method heading).

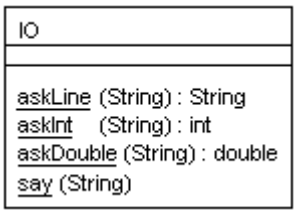


Figure 6.2 The IO class

The conditional operator

The if-else statement in the `askLine` method of Listing 6.2 can be written this way:

```
return s == null  ?  ""  :  s;
```

The `?:` operator replaces an if-else statement in certain cases: `condition ? x : y` is an expression whose value is `x` if the `condition` is true, but `y` if the `condition` is false. The expressions on each side of the colon `:` must be the same type of value. This book recommends you only use this **conditional operator** for a `return` value or for a value assigned to a variable. If you choose to use it in any other way, you should put parentheses around the entire three-part expression.

In the MathOp class of Listing 5.1, the body of the `average` method can be written as a single statement using the conditional operator:

```
return sum >= 0  ?  (sum + count / 2) / count
                 :  (sum - count / 2) / count;
```

An independent class method that could be in MathOp to round off a given double value to the closest int value is as follows. The trick is to add 0.5 and see if that makes the number big enough to be rounded up rather than down. For instance, 3.4 rounds to 3 because `3.4+0.5` is 3.9, but 3.6 rounds to 4 because `3.6+0.5` is 4.1:

```
public static int roundOff (double par)     // in MathOp
{  return par >= 0  ?  (int) (par + 0.5) : (int) (par - 0.5);
}  //======================
```

This trick generalizes to round off values to any specific number of decimal places; the following would round off to hundredths (two places right of the decimal point):

```
return par >= 0  ?  (int) (par * 100 + 0.5) / 100
                 :  (int) (par * 100 - 0.5) / 100;
```

Some examples of how the conditional operator could be used for a value assigned to a variable are as follows:

```
// assign to lowPay the smaller of hisPay and herPay
lowPay =   hisPay < herPay  ?  hisPay  :  herPay;

// assign to halfDay the AM/PM marker for military time
String halfDay =   military >= 1200  ?  "PM"  :  "AM";
```

The reason for using the conditional operator is not to save space, it is to increase the clarity of the logic. If the two preceding statements were written as if-else statements, it would obscure the fact that the purpose of the statements is to assign a value to `lowPay` and to `halfDay`. With the conditional operator, that purpose is brought to the fore. In any case, any coding written with the conditional operator can be restated without it, so it is not an essential part of the Java language.

Exceptions

A method **throws an Exception object** with a statement similar to the following:

```
throw new RuntimeException ("Your message here");
```

This notifies the calling method of a problem that crashes the program unless the calling method has special statements that are discussed in Chapter Ten. For instance, if a method contains an int expression involving division, and the divisor has the value zero, that method throws an **ArithmeticException** object. Or if you evaluate `s.length()`, it throws a **NullPointerException** object when `s` has the value `null`. You should always verify that the Java logic you write can never do this.

The `Double.parseDouble` and `Integer.parseInt` methods can throw a **NumberFormatException** object if the user enters a badly-formed numeral, such as one with two minus signs or with a comma or letter in it (commas are not allowed within numbers in a Java program). You cannot easily check for this before you call a parsing method. So programs such as those in the earlier Listing 6.1 should have several more statements to handle the Exception object when it is thrown, as described in Chapter Ten, if they are to be used commercially.

Nested classes

One class can be declared inside another class with a heading such as the following:

```
private static class InsideClass
```

The InsideClass is a **nested class**. The **members of a class** X are the variables, methods, and classes whose declarations are inside X but not inside any member of X.

The effect of putting such a declaration within class X instead of outside class X is only on visibility: Private members of X can be mentioned within the InsideClass; public members of the InsideClass can be mentioned anywhere in X; and no class outside X can mention the InsideClass. The primary purpose therefore is to maintain encapsulation (i.e., so classes outside the "top-level class" X cannot modify variables inside X). This language feature is used in several later chapters.

Language elements
An expression of this form can be used judiciously: Condition ? Expression : Expression
The two values on each side of the colon must be of the same type, which will be the result's type.
You may declare one class inside another with this heading: private static class Whatever
Such a class declaration only affects visibility of identifiers.

Exercise 6.9 Write an IO method `public static int askNonNeg (String prompt)`: It repeatedly asks for an int value until it gets a non-negative value. Use a while-statement.
Exercise 6.10 Rewrite one part of each of Listing 4.5 and Listing 4.6 to use the conditional operator appropriately.
Exercise 6.11* Rewrite parts of Listing 5.5 and Listing 5.10 to use the conditional operator appropriately.
Exercise 6.12* Draw the full UML class diagram for the IO class.
Exercise 6.13** Replace the logic in `askInt` so that it accepts a double value and rounds it down before returning it. Hint: Use `(int)`, but note that `(int)` applied to a negative non-integer rounds up. You need an input of `3.8` returned as `3`, and you need an input of `-3.8` or `-3.2` returned as `-4`.

6.3 Basic String Methods; The Comparable Interface

Java provides the standard library class String in the `java.lang` package. A String object represents a string of characters. You can write a String literal in a program as a string of characters enclosed in quotes. So if you declare the variable `String line`, you can assign `line = "hello"`. That means that the variable `line` refers to an object that contains the five characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`, in that order.

Review of String features seen so far

The only operator for String values is the plus sign. Between two String values, the plus sign attaches the second String value to the end of the first; this is **concatenation**. So `"car" + "ted"` is the String value `"carted"`. Between a String value and a numeric value, the plus sign attaches the string of characters that represent the number. So `"car" + 54` is the String value `"car54"`. This plus operator is evaluated left-to-right in the absence of parentheses, so `"car" + 5 + 4` is `"car54"` but `"car" + (5 + 4)` is `"car9"`. The special assignment operator `+=` can be used for Strings, e.g., `s += "it"` means the same as `s = s + "it"`.

You have seen three String methods so far in this book, all of which are illustrated in the following coding to find the word "the" in a sentence:

```
for (int k = 0;  k < sentence.length() - 2;  k++)
   if ("the".equals (sentence.substring (k, k + 3)))
      return true;
```

- `s.equals(t)` is true when `s` and `t` have the same characters in the same order.
- `s.length()` is the int value that tells how many characters `s` has.
- `s.substring(start, end)` is the String value that contains the characters of `s` beginning at position `start` and going up to but not including position `end`. Position numbers are **zero-based**, i.e., the first character is at index 0. This method call throws an **IndexOutOfBoundsException** if `start < 0` or `end > s.length()`.

None of Java's String methods change the state of a String value; they at most return new String values. So Strings are **immutable**. A defect in the String class is that it does not have a method that tells whether the executor can be safely parsed as an int or a double, so you can call `parseInt` or `parseDouble` for it. We will develop a way around this problem later in this chapter.

Caution You might think you do not need to test `s.equals(t)` to see whether two Strings are equal; you could just use the `==` operator. But the `==` operator between two object references only tests whether they are exactly the same object. You usually want to know whether they have the same sequence of characters, even though those sequences may be stored in different String objects (i.e., in different places in RAM). If you are ever tempted to use the `==` operator, stop and think: String objects are like boxes, and characters are what are in the boxes, so do you want to know whether two Strings are the same box or whether they have the same contents? General principle: If you want to be safe, never use `==` or `!=` between two object values unless one of them is `null`.

The backslash character

If you want a quote character to be part of a String literal, you have to write `\"` inside the String literal's quotes. This **backslash character** within a String literal signals to the compiler that the quote character that comes directly after it is a character in the string, not the character that marks the end of the string. For instance, you could print `the game of "Guess My Number"` with the statement:

```
JOptionPane.showMessageDialog (null,
             "the game of \"Guess My Number\"");
```

In general, Java uses the backslash character in a String literal to signal that the character that follows it has a different meaning from what it would otherwise have. If you want a tab or newline to be part of the String literal, you have to write `\t` or `\n`, respectively. If you want to backspace within the String literal, you have to write `\b`. And if you want the backslash itself in the String literal, you have to write `\\` inside the quotes. These "metacharacters" are often called **escape sequences**.

The compareTo String method

The method call `s.compareTo(t)` compares the two String values `s` and `t`. The number it returns tells which comes earlier alphabetically, if all characters are capital letters or if all are lowercase letters. In other cases, the order of the two String values is determined by a standard set of numerical codes for the characters, as described in the next section (on character values). Technically, it is called **lexicographical ordering**, which is a more general term than "alphabetical ordering."

Figure 6.3 gives details on the `compareTo` method. The numeric result returned by the `compareTo` method is analogous to the value of `n - p` for numbers (since `n - p` is negative if `n < p` and positive if `n > p`).

s.compareTo(t)
returns 0 if they are equal, a negative integer if s comes before t, and a positive integer if s comes after t. s and t are Strings.

Figure 6.3 The compareTo String method

Examples: `"bid".compareTo("bob")` is a negative integer, and so is `"AL".compareTo("JO")`, so of course `"bob".compareTo("bid")` is a positive integer and so is `"JO".compareTo("AL")`. `"sam".compareTo("sam")` is zero.

As you would expect, if two different Strings `s` and `t` have the same characters up until `s` ends, the shorter one comes before the longer one -- `s.compareTo(t)` is negative. For instance, `"apple".compareTo("applet")` is a negative number.

The **empty String**, i.e., the String with no characters in it, comes before all others in the standard ordering, since it is shorter than all others. The empty String is written in a program as two quote marks `""` right next to each other.

Comparable objects

The String class in the Sun standard library has the phrase `implements Comparable` in its heading. That means that String objects can be passed to method parameters of the Comparable type. The Sun standard library has many classes that implement Comparable, including the File class and the Date class.

Comparable is not a class, it is a category of classes, called an **interface**. A class is **Comparable** if it has a `compareTo` method with an Object parameter and it returns an int value. The advantage of having the Comparable category of classes is that it promotes reusable software. When you write a method with parameters that are Comparable rather than simply String parameters, you can use that method for Strings, Files, Dates, and other kinds of objects.

Interfaces are used heavily in event-driven programming, discussed in Chapter Nine. For instance, standard library classes have methods that tell an ActionListener kind of object to respond to the click of a button. If you declare a class that `implements ActionListener`, those library methods tell an object of your class to carry out the actions you choose.

A method with Comparable parameters

The logic of the method in Listing 6.3 produces a message saying whether three given String values are in order. But it can also be used for File objects, Date objects, and other kinds of objects, because the parameters are of Comparable type rather than String type. The following statement is an example of how it can be called:

```
System.out.println ("The strings are " + ordered (s, t, u));
```

Listing 6.3 An independent class method to test Comparable values

```
/** Tell what kind of order the 3 values are in. */

public static String ordered (Comparable first,
           Comparable second, Comparable third)
{  if (first.compareTo (second) == 0)
      return second.compareTo (third) >= 0
            ?  "in ascending order"  : "in descending order";
   else if (first.compareTo (second) < 0)
      return second.compareTo (third) > 0
            ?  "not in order"  :  "in ascending order";
   else  // first comes after second
      return second.compareTo (third) < 0
            ?  "not in order"  :  "in descending order";
}  //======================
```

This logic illustrates the use of the `compareTo` method. It first tests whether the `first` value equals the `second` because if so, all three are in order. Otherwise, the logic splits into two parts, depending on whether the `first` value comes before the `second`. If it does, you need to have the `second` be less than the `third` for ascending order; if it does not, you need to have the `second` value be greater than the `third` for descending order.

Programming Style The comment after the second `else` in the `ordered` method is intended to make it clear to the reader that, at this point in the logic, `first` is known to come after `second`. There is of course no need to inform the runtime system; it is not sentient. It is good style to avoid testing a condition at a point where you know what the result of the test will be. In particular, it would be pointless to write `else if (first.compareTo (second) > 0)` in place of that commented `else`.

Class casts

You may put `implements Comparable` in the heading of one of your classes if the class has an int-returning method with the signature `compareTo(Object)`. For instance, you could add such a method to the Time class of Listing 4.5, so you could use the `ordered` method and others for Time objects.

The Time instance variables are `itsHour` and `itsMin`. One Time value is "larger" than another if it has a larger value of `itsHour` or, with equal values of `itsHour`, it has a larger value of `itsMin`. So the obvious coding for the `compareTo` method is as follows (because the exact value of the integer returned is not material; only its sign counts):

```
public int compareTo (Object anyTime)  // defective
{  return this.itsHour != anyTime.itsHour
          ? this.itsHour - anyTime.itsHour
          : this.itsMin - anyTime.itsMin;
}  //=====================
```

You would of course call the method as `x.compareTo(y)` with two Time variables `x` and `y`. But the method heading has `anyTime` as an Object parameter, not a Time parameter, so the compiler will not let the method body refer to `anyTime.itsHour` or `anyTime.itsMin`. The compiler will complain that Objects do not have instance variables named `itsHour` and `itsMin`. And as usual, the compiler is right.

You cannot fix the problem by making the parameter a Time parameter, because `compareTo` is required to have an Object parameter. What you need is a way to tell the compiler that `anyTime` will be a Time sort of object even though it looks like an Object sort of object. The solution that Java offers is to replace the method definition by the following, in which only the first two lines have been changed:

```
public int compareTo (Object ob)  // in the Time class
{  Time anyTime = (Time) ob;   // promises that ob is a Time
   return this.itsHour != anyTime.itsHour
          ? this.itsHour - anyTime.itsHour
          : this.itsMin - anyTime.itsMin;
}  //=====================
```

The first statement of the method uses the **class cast** expression `(Time) ob`, which tells the compiler that at runtime `ob` will refer to a Time object. That allows you to assign it to the Time variable `anyTime`. And that in turn allows you to use `itsHour` and `itsMin` as instance variables. Now the heading of the Time class can be as follows:

```
public class Time extends Object implements Comparable
```

The general principle is that, when you write `(C)x` for some class C, that tells the compiler that `x` will at runtime refer to an object of class C. The compiler accepts such a phrase if C extends the class declared for `x` or implements the interface declared for `x` (either directly or indirectly). But if at runtime the reference in `x` is not to an object of class C or of a subclass of C, a **ClassCastException** is thrown.

By contrast, if you want to assign an object value of class C to a variable of a superclass of C or of an interface that C implements, you do not have to use a cast. The assignment will never throw an Exception. This is in fact what you are doing when you pass `y` to `ob` by the method call `x.compareTo(y)`. And it is what you are doing when you pass the String `s` to `first` by the method call `ordered(s,t,u)`.

The corrected `compareTo` method for the Time class could have used the phrase `((Time) ob)` in three places instead of assigning the value to the `anyTime` variable. But that would execute a bit slower and be harder to read. The parentheses around the whole expression are needed; the phrase `(Time) ob.itsHour` would be interpreted by the compiler as getting the `itsHour` value from `ob` and then casting that int value to a Time object, which does not make sense. You need to use this expression:

```
((Time) ob).itsHour
```

The calls of `compareTo` in the `ordered` method of Listing 6.3 are polymorphic: If you call `ordered(s,t,u)` for three String values, the runtime system uses the `compareTo` method of the String class. If you call it for three Time values, the runtime system uses the `compareTo` method of the Time class.

The equals method for the Time class

Any class that has a `compareTo` method should also have an `equals` method for which two objects that are equal have a `compareTo` value of zero. A good `equals` method for the Time class would tell whether two different Time objects have the same values for `itsHour` and `itsMin`. The method could have the heading `public boolean equals (Time given)`. The following method definition is a suitable one for this purpose:

```
public boolean equals (Time given)  // in the Time class
{  return this.itsHour == given.itsHour
               && this.itsMin == given.itsMin;
}  //======================
```

Language elements
This following form of expression tells the compiler that the class of the object will be a subclass or an implementation of the expression's type: (ClassName) ObjectExpression
You may append the following to a class heading: implements Comparable
if it has a method with this heading: public int compareTo (Object ob)
A string literal can have \t or \n or \" or \b or \\ within the quotes that start and end it.

Exercise 6.14 What changes would you make in the `ordered` method in Listing 6.3 to print the string of characters within the method instead of returning it?
Exercise 6.15 Write a single statement to print just the number stored in the int variable `x` using `JOptionPane.showMessageDialog`.
Exercise 6.16 Write a statement that prints the substring consisting of the fourth, fifth, and sixth characters in the value stored in the String variable `s`.
Exercise 6.17 Write an independent method `public static boolean sameFirstFive (String one, String two)`: It tells whether the two given Strings have the same first five characters. Return `false` if either has less than five characters.
Exercise 6.18 Write an independent method `public static int indexOf (String big, String little)`: It returns the index in `big` of the first substring of `big` that equals `little`. It returns -1 if `little` is not equal to any substring of `big`.
Exercise 6.19* Rewrite the `compareTo` method for the Time class without any local variable such as `anyTime`.
Exercise 6.20* Write an `equals` method and a `compareTo` method for the Person class in Listing 5.2. One Person object comes before another if it is younger (has a later `itsBirthYear`) or, for Persons with the same birth year, it has a lexicographically earlier `itsLastName`.
Exercise 6.21* Write an application program that reads in three strings of characters from the keyboard and then prints the one that comes first as determined by `compareTo`.

6.4 Character Values And String's CharAt Method

The car repair software will require you to work with individual character values. Java denotes a single character by putting it in apostrophes, as in `'x'` or `'3'` or `'*'`. A variable to store a single character is of **char** type, as in `char firstLetter = 'a'`.

Each char value has a unique whole-number numerical equivalent we call its **Unicode** value. These Unicode values range from 0 to 65,535 inclusive. The main points to remember about the order of the Unicode values are:

- The capital letters are together and in their normal order, that is, `'A'` directly precedes `'B'` which directly precedes `'C'`, etc. through `'Z'`.
- The lowercase letters are together and in their normal order, that is, `'a'` directly precedes `'b'` which directly precedes `'c'`, etc. through `'z'`.
- The digits are together and in their normal order, that is, `'0'` directly precedes `'1'` which directly precedes `'2'`, etc. through `'9'`.
- Lowercase letters come after capital letters which come after digits which come after a blank.

If you combine a char value with an int value using an operator, the system automatically promotes it to the int value corresponding to its Unicode. The same thing happens if you assign a char value to an int variable. You cannot assign an int value or a double value to a char variable without a **cast**, e.g., `chr = (char) ent` is legal when `chr` is a char variable and `ent` is an int value, and so is `ent = chr`, but not `chr = ent`.

Two more String methods

Figure 6.4 shows the only other methods in the String class that you need to know for this book. They allow you to find out what the character at a particular position is and to obtain the ending portion of a given String value.

`s.charAt(k)`
is the character at the kth position in the String object referred to by s. An Exception is thrown unless 0 <= k <= s.length() - 1.

`s.substring(n)`
returns a new String object consisting of the characters of s starting at position n and continuing to the end. An Exception is thrown unless 0 <= n <= s.length().

Figure 6.4 Two String methods

Java counts positions in a String value starting from 0, so `"x z".charAt(0)` is the letter `'x'`, `"x z".charAt(1)` is a blank, `"x z".charAt(2)` is the letter `'z'`, and evaluation of `"x z".charAt(3)` throws an **IndexOutOfBoundsException**. Figure 6.5 illustrates the meaning of these String methods.

Caution If you use the `charAt` method with an index outside the range 0 to `length()-1`, the program throws an IndexOutOfBoundsException. Similarly, `substring(a,b)` requires that `0 <= a <= b <= length()`, otherwise it throws an IndexOutOfBoundsException. Make sure your coding verifies that all index values are within range.

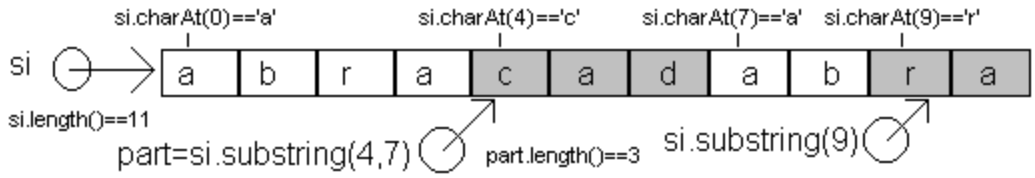


Figure 6.5 Use of String methods

You are perhaps thinking that it is illogical and confusing for characters in a String object to be numbered from 0 on up instead of from 1 on up. But every language has quirks. Just as you won't be speaking correct Spanish unless you learn to use a double negative to actually mean a negative, you won't be writing no correct Java unless you learn to number from 0.

These new String methods allow a more natural rewrite of Listings 5.3 through 5.5. For instance, Listing 5.3 defines `String NONE = "0"`. It could instead define `char NONE = '0'` and then use NONE as follows, since a plus sign between a char value and a String value concatenates the char onto the string:

```
private static final char NONE = '0';

// replace the last two lines of seesCD by:
   return itsSequence.charAt (itsPos) != NONE;

// replace the body of the if-statement in takeCD by:
   itsStack += itsSequence.charAt (itsPos);
   itsSequence = itsSequence.substring (0, itsPos) + NONE
               + itsSequence.substring (itsPos + 1);
```

Class methods versus instance methods

As a general rule, you should strongly prefer to make your methods instance methods rather than class methods. It is easier to complete large software projects successfully if you think in terms of having some object do a job you need to have done (by calling one of its instance methods) rather than doing it yourself (by calling a class method, i.e., with no executor other than you the programmer). But there are several kinds of situations in which a class method is appropriate:

1. You want information about a class of objects as a whole rather than an individual object. For instance, in Listing 5.2, `Person.getNumPersons()` tells how many Person objects have been created.
2. You want information about numbers or characters. For instance, if you want the number 2 to the power `x`, where x is an int value, you cannot use `x.powerOf2()`, because numbers cannot be executors. Thus you have MathOp in Listing 5.1.
3. The main method of an application program is required to be a class method.

The StringInfo class, developed next, follows the principle that instance methods are preferable to class methods.

Algorithms involving Strings

The String class in the Sun standard library has many useful methods, but sometimes you need more than it offers. For instance, it is useful to be able to determine whether all characters in a given String are between two given characters `lo` and `hi`. Then the method call `someString.inRange('0', '9')` could test whether `someString` has nothing but digits. However, the String class does not have such a method.

The StringInfo class presented next has several useful methods needed for the car repair software, and you may add more if you wish. For instance, if `info` is a StringInfo value, the following statement would see whether it is either all lowercase letters or all capitals:

```
if (info.inRange ('A', 'Z') || info.inRange ('a', 'z'))
   System.out.println (info + " is acceptable.");
```

A **StringInfo** object is basically a wrapper around a String value -- it has one instance variable of type String, named `itself`. This is the use of **composition** rather than inheritance. The implementation of `inRange` requires the common **All-A-are-B looping action**: You look at each character of the String one at a time, from left to right. If you see a character that is not in the specified range, you return `false`. If you get to the end without seeing such a character, you return `true`:

```
public boolean inRange (char lo, char hi)  // for StringInfo
{  for (int k = 0;  k < itself.length();  k++)
      if (itself.charAt (k) < lo || itself.charAt (k) > hi)
         return false;
   return true;
}  //=====================
```

By contrast, if you wanted a method to tell you whether at least one character in `itself` is in the range from `lo` to `hi` inclusive, the logic would be the **Some-A-are-B looping action**: Return `true` if you see any character in the specified range; return `false` if you get to the end of the String without seeing one. This gives you the following coding (which you should compare closely with `inRange` above):

```
for (int k = 0;  k < itself.length();  k++)
   if (itself.charAt (k) >= lo && itself.charAt (k) <= hi)
      return true;
return false;
```

The upper part of Listing 6.4 (see next page) contains the constructor, `toString` method, and `compareTo` method for the StringInfo class. The standard name for a method that gives a String equivalent of an object is `toString` (as you saw for the Time class in Chapter Four). The `compareTo` method, which justifies `implements Comparable` in the class heading, requires that you cast its parameter to a StringInfo so you can access `itself`.

The trimFront, firstWord, and initialDigits methods

The RepairShop software will read in a line containing several words, and you will need to be able to separate out the individual words. A quite useful method is one that discards the first few characters plus any additional **whitespace** (blanks, tabs, page breaks, or other characters whose Unicode value is less than that of a blank). A parameter gives the starting index. You then increment that index until you come to a character whose Unicode value is larger than that of a blank or until you come to the end of the string. The implementation of this `trimFront` method is in the middle part of Listing 6.4.

Once you have a string of characters with no leading blank, you need logic to move down the string until you see a blank, then return the characters up to that blank. This is the `firstWord` method in Listing 6.4. An example of how these two methods might be used to strip off the first word of a given StringInfo value named `info` is as follows:

```
info.trimFront (0);
String nextWord = info.firstWord();
info.trimFront (nextWord.length());
```

Listing 6.4 The StringInfo class of objects

```
public class StringInfo extends Object implements Comparable
{
   private String itself;


   public StringInfo (String given)
   {  super();
      itself =   (given == null)  ?  ""  :  given;
   }  //======================

   public String toString()
   {  return itself;
   }  //======================

   public int compareTo (Object ob)
   {  return itself.compareTo (((StringInfo) ob).itself);
   }  //======================

   /** Remove all characters before index n, plus any
    *  whitespace immediately following those characters.  */

   public void trimFront (int n)
   {  while (n < itself.length() && itself.charAt (n) <= ' ')
         n++;
      itself = n > itself.length()  ?  "" : itself.substring (n);
   }  //======================

   /** Return the first word of the String, down to but not
    *  including the first whitespace character.  */

   public String firstWord()
   {  for (int k = 0;  k < itself.length();  k++)
         if (itself.charAt (k) <= ' ')
            return itself.substring (0, k);
      return itself;  // since the string has no whitespace
   }  //======================

   /** Strip out all non-digits from the StringInfo object. */

   public void retainDigits()
   {  String result = "";
      for (int k = 0;  k < itself.length();  k++)
         if (itself.charAt(k) >= '0' && itself.charAt(k) <= '9')
            result += itself.charAt (k);
      itself = result;
   }  //======================
}
```

You may also need a method that discards all extra spaces, commas, and dollar signs from a numeral, leaving only the digits in the string. The `retainDigits` method in the lower part of Listing 6.4 finds only the digits in the StringInfo executor. It initializes a string `result` to have no characters, then adds any digits it sees to `result`.

The last three methods in this listing use the **count-controlled loop pattern**: Execute the body of the loop a number of times that is known at the time the loop begins, except you may terminate early if you find what you are looking for. That known number of times

is `itself.length()` in each method. Note: You saw the sentinel-controlled loop pattern in Section 6.1, and you will see other loop patterns elsewhere in this book.

The `firstWord` method is an example of **sequential search**: You go through a sequence of values to see if you can find a particular value. In this case, you are searching for a blank.

Parsing a string to obtain a number

The IO methods `askInt` and `askDouble` in the earlier Listing 6.2 do not use the Sun standard library `Integer.parseInt` or `Double.parseDouble` methods directly on input from the user. Otherwise they would throw an Exception if the user accidentally types a letter among the digits, or clicks the CANCEL button, or presses the ENTER key without typing any characters, or gives any of several other kinds of bad inputs. The body of the `askDouble` method of the IO class is the following:

```
String s = JOptionPane.showInputDialog (prompt);
return new StringInfo (s).parseDouble (-1);
```

This method call returns the double value that the first part of the string represents, even if there are some unacceptable characters after that acceptable first part. For instance, if the string is "53x" or "-53,217" or ".24 ", it returns 53 or -53 or 0.24 respectively. If the initial part of the string does not form a valid numeral, this `parseDouble` method call returns -1, the parameter value. This special return value allows you to check whether the conversion from a numeral to a number succeeded, since the RepairShop software does not expect a negative number for input. The body of IO's `askInt` method is the same as its `askDouble` method except for an `(int)` cast before the return value.

Listing 6.5 (see next page) has StringInfo's `parseDouble` method. The first step is to return the parameter value if the string has no characters. Otherwise you set `pos` to the first index that might be a digit (index 1 if the first character is a minus sign, otherwise index 0). Then you advance `pos` past any digits that occur. If the first nondigit is a decimal point, you advance `pos` past any digits that occur after the decimal point. As you advance, you make a note of whether you saw at least one digit either before or after the decimal point (since both "44." and ".44" are acceptable numerals). If you saw at least one digit, you can then safely call `Double.parseDouble` on the substring up to the first invalid character, otherwise you return the parameter value.

Language elements
A char value represents a single character. Each character has a Unicode value 0 to 65,535.
If you assign a char value to a numeric variable or use a numeric operator with it, it is converted to the int value corresponding to its Unicode. Assigning a double or int value to a char variable requires you put the cast operator (char) in front of the numeric expression.

Exercise 6.22 If `s` is the String value "algorithm", what is `s.charAt(2)`? What is `s.substring(2)`? What is `s.substring(2, 6)`?
Exercise 6.23 Write a StringInfo method `public void trimRear()`: The executor discards all the whitespace at the end of itself.
Exercise 6.24 The `firstWord` method is only supposed to be called when the given String does not begin with whitespace. What happens if it does begin with whitespace?
Exercise 6.25 Write a StringInfo method `public int countRange (char lo, char hi)`: the executor tells how many characters in itself are in the range `lo` to `hi` inclusive. For instance, `info.countRange('A','Z')` counts the capital letters.
Exercise 6.26 Write a StringInfo method `public char biggestChar()`: The executor returns the character in itself that has the highest Unicode value. If the string is empty, return `(char) 0`.

Listing 6.5 The parseDouble method in the StringInfo class

```
/** Return the numeric equivalent of itself.
 *  Ignore the first invalid character and anything after it.
 *  If the part of the string before the first invalid
 *  character is a numeral, return the double equivalent,
 *  otherwise return the value of badNumeral. */

public double parseDouble (double badNumeral)
{  if (itself.length() == 0)
      return badNumeral;
   boolean hasNoDigit = true;  // until a digit is seen
   int pos =   (itself.charAt (0) == '-')  ?  1  :  0;
   for ( ;  hasDigitAt (pos);  pos++)
      hasNoDigit = false;
   if (pos < itself.length() && itself.charAt (pos) == '.')
      for (pos++;  hasDigitAt (pos);  pos++)
         hasNoDigit = false;
   return hasNoDigit  ?  badNumeral
          :  Double.parseDouble (itself.substring (0, pos));
}  //======================

private boolean hasDigitAt (int pos)
{  return pos < itself.length() && itself.charAt (pos) >= '0'
                                && itself.charAt (pos) <= '9';
}  //======================
```

Exercise 6.27 Write a StringInfo method `public int lastBiggie (StringInfo par)`: The executor returns the last position number at which it has a character that has a larger Unicode than the character of the parameter in the same position. It returns -1 if there is no such character.
Exercise 6.28 Write a StringInfo method `public boolean hasWhitespace()`: The executor tells whether it contains at least one character less than or equal to a blank.
Exercise 6.29 Write a StringInfo method `public int indexOf (char par)`: The executor returns the position number of a given character in itself; it returns -1 if the character is not there. In case the character is in itself several times, it returns the position of the first instance of it.
Exercise 6.30 Write a StringInfo method `public String initialDigits()`: The executor returns the initial portion of itself consisting of digits (up to the first non-digit).
Exercise 6.31* Write a StringInfo method `public String signedNumber()` that does what the preceding exercise does except that it includes an initial negative sign if it is directly followed by a digit. Call the `initialDigits` method as part of the logic.
Exercise 6.32* Revise the method of the preceding exercise to allow at most nine digits, or ten if the first digit is a 1 (due to the limits on the size of an int value).
Exercise 6.33** Write a StringInfo method `public StringInfo reverse()`: The executor returns a new StringInfo value of the same length but with the characters in reverse order. Hint: You cannot assign a new value at a particular index in a string, but you can concatenate characters on to a string one at a time to get what you want.
Exercise 6.34** Write a StringInfo method `public void replace (char lo, char hi)`: The executor replaces every instance of the char value `lo` by the char value `hi` and makes no other change in itself.
Exercise 6.35** Write a StringInfo method `public void numWords()`: The executor tells how many words it has (a word is non-whitespace followed by either whitespace or the end of the string).

6.5 Long Integers; Casts And Conversions; The Math Class

You may declare a variable as type **long**. That means that it can store an integer value with up to 63 binary digits, in addition to a negative sign if needed. So the range of values is plus or minus slightly over eight billion billion, which is a 19-digit number.

One advantage of long values is that you can store money amounts up to 80 million billion dollars, exactly to the penny. An int value has 31 binary digits plus a possible negative sign, so an int variable cannot store more than about 20 million dollars exactly to the penny. The only disadvantages of using long values versus int values are that they take twice the space in RAM and operations with long values take at least twice as long.

Reading the system clock

The expression `System.currentTimeMillis()` gives the current time, measured in the number of milliseconds that have passed since midnight of January 1, 1970. There are 1000 milliseconds in a second, and 86,400 seconds in a day (multiplying 60*60*24), for a total of 86,400,000 milliseconds in a day. So if the current time were returned as an int value, it would give the wrong answer after a little less than a month. That is why the current time is returned as a long value.

If you have a method call `doStuff(x)` that works with an Object x, and if you want to know how much time it takes to execute that method call, you could call the following method. It checks the time just before the method executes, then again after it finishes execution, and reports the difference in the two times.

```
public static int getElapsedTime (Object x)
{  long start = System.currentTimeMillis();
   doStuff (x);
   return (int) (System.currentTimeMillis() - start);
}  //======================
```

Conversions

The char, int, long, double, and boolean types are called **primitive types** of values, since they are not objects. Java has three other primitive types, all numeric. These are discussed in Chapter Eleven, but they are not used in this book.

You may assign a long value to a double variable without an explicit cast. But you cannot assign a long value to an int or char variable without an explicit cast, as in `intVariable = (int) longVariable`. The hierarchy

```
char -> int -> long -> double
```

summarizes casting requirements: You must cast a value of one of these types to assign it to a variable of a type further to the left. You do not need to cast if you assign a value to a variable of a type further to the right. Note that boolean is not in this hierarchy; you cannot cast a boolean value to or from any other primitive type of value.

One **byte** of space is 8 bits, corresponding to 8 base-two digits. So you may store up to 256 (2^8) different values in one byte of space. Unicode values (for chars) require two bytes of storage space, since they range from 0 up to 65,535 (2^{16}). int values require four bytes, since they range up to plus or minus 2^{31}. long and double values require eight bytes (long values range up to plus or minus 2^{63}).

Using a char or int value as if it were "wider" (by combining it with a "wider" number using an operator or assigning it to a "wider" variable) is a **widening conversion**, or **promotion**, and does not lose information (except a little bit with longs to doubles). So

the compiler does this automatically. Assigning to a "smaller" variable (e.g., double to long or long to int or int to char) is a **narrowing conversion**. The compiler requires you to acknowledge the possibility of a loss of information by using a cast for a narrowing conversion. Combining a value with a "wider" value is called "arithmetic conversion"; assigning it to a "wider" variable is called "assignment conversion".

This is analogous to the rule for class casts (discussed in Section 6.3): If Sub is a subclass of Super, and you have declared `Sub x; Super y;` then you may make the assignment `y = x` but assigning the other way requires a cast: `x = (Sub) y`. The latter is analogous to narrowing, since `x` is more specific than `y`; the `y = x` assignment is analogous to widening.

Note: Subtracting an int value from a char value converts the character to its Unicode value. If `ch` is a char variable that contains a capital letter, then `(ch - 'A')` will be an int value in the range from 0 to 25. If `ch` contains a lowercase letter, then `(ch - 'a')` will be in the range from 0 to 25. And if `ch` contains a digit, then `(ch - '0')` will be in the range from 0 to 9 (inclusive in each case).

The Math class

The Sun standard library has a class of methods that perform some basic calculations. Some of the methods in this **Math** class are as follows. Each of these methods is a class method, as you can see from the fact that the class name appears in place of a reference to the executor:

- `Math.sqrt(x)` returns the square root of x, e.g., `Math.sqrt(25)` is 5. If you try to calculate the square root of a negative number, it returns the value **NaN**, which stands for "not a number".
- `Math.abs(x)` returns the absolute value of x, e.g., `Math.abs(-12)` is 12.
- `Math.min(x, y)` is the smaller of x and y, e.g., `Math.min(5,-12)` is -12.
- `Math.max(x, y)` is the larger of x and y, e.g., `Math.max(5,-12)` is 5.
- `Math.pow(x, y)` returns x-to-the-power-y, e.g., `Math.pow(2,5)` is 32.
- `Math.random()` returns a "random" number `r` such that `0 <= r < 1`.
- `Math.cos(x)` returns the cosine of x.
- `Math.sin(x)` returns the sine of x. Other trigonometric functions are available.
- `Math.log(x)` is the natural logarithm of x, that is, to base e. If you try to calculate the logarithm of a negative number, it returns the value NaN.
- `Math.exp(x)` is e to the power x, so `Math.exp(Math.log(x))` is x.

Each of these Math methods returns a double value calculated from one or two double parameters, except that `abs`, `min`, and `max` will instead return an int value when the parameter(s) are int values, or a long value when the parameters are long. The Math class also contains two constants: `Math.PI` is the area of a circle of radius 1, roughly 3.14159, and `Math.E` is the natural base of logarithms, roughly 2.71828.

Crash-guards

The following three if-statements have something in common. Do you see what it is?

```
if (sue != null)        s = sue.getName();
if (k < par.length())   s = "" + par.charAt (k);
if (x != 0)             y = 6 / x;
```

The subordinate statement in each case would throw an Exception if it were executed when the condition is false, e.g., `sue.getName()` throws a **NullPointerException** if sue contains the `null` value. But the if-statement guards against such a "crash".

The following three conditions also have something in common. Do you see what it is?

```
sue != null && sue.getName().equals ("bob")
k < par.length() && par.charAt (k) > ' '
x != 0 && 6 / x < 3
```

The part after the and-operator in each case would throw an Exception if it were evaluated when the part before the and-operator is false. But the short-circuit property of the and-operator guards against such a crash.

In each case, the condition at the beginning of the phrase is a **crash-guard** for the expression or statement in the latter part of the phrase. A crash-guard prevents the evaluation of an expression or statement in a case when the expression or statement is impossible to evaluate sensibly.

A crash-guard can also be a condition that, when true, prevents the evaluation of a phrase that would throw an Exception. This happens when the guarded part is subordinate to `else` or comes after a conditional return, as follows:

```
if (x == 0)  { /* misc. statements */ }  else  y = 6 / x;
if (x == 0)  return 0;  y = 6 / x;
```

And of course, the short-circuit property of the or-operator can be used to crash-guard, as in the following conditions:

```
k >= par.length() || par.charAt (k) <= ' '
x == 0  ||  6 / x < 3
```

Programming Style As a general principle, you should crash-guard all expressions and statements that could possibly throw an Exception under certain circumstances, when those circumstances can be guarded against with one or two simple tests of conditions. However, you do not need to guard against circumstances that are expressly prohibited by the precondition of a method. The precondition states what crash-guarding has already been done before the method is called, so further crash-guarding would be superfluous.

Language elements
The narrow-to-wide hierarchy char => int => long => double for primitives determines casting: You must use a cast to have a value treated as a "narrower" value, but not to have a value treated as wider. A long integer value ranges up to plus or minus 2^{63} - 1.

Exercise 6.36 Write an independent method `public static char toUpperCase (char par)`: It returns the capitalized form of the char parameter. It returns the parameter without change if the parameter is not lowercase.
Exercise 6.37 Write an independent method `public static double largestOf4 (double x, double y, double z, double w)`: it returns the largest of four double parameters. Have just one statement. Use `Math.max` three times.
Exercise 6.38 Write an application program that reads in a double value and prints its square root, fourth root, and eighth root.
Exercise 6.39 State which of the expressions or statements in Listing 2.8 has a crash-guard, and state which of the four types of crash-guard it is.
Exercise 6.40* Java has its own analog of the Y2K bug, in that times and dates will be messed up when the current time exceeds the capacity of a long value. In approximately what year will that happen?
Exercise 6.41* Same question as the previous exercise, but for Listing 3.10.
Exercise 6.42* Same question as the previous exercise, but for Listing 5.5.

6.6 Formatted Output To A JTextArea In A JScrollPane

The car repair software requires the display of a large amount of output in a graphical window. `IO.say` is not appropriate for this part of the output, since you will print some of the data at one point in the program, then more at another point, etc. Also, the user may want to review the output from previous steps. You could use `System.out.println`, but the alternative presented in this section is often preferable.

The JTextArea class

You may construct a rectangular area for the output of information with the following statement:

```
JTextArea area = new JTextArea (10, 25);
```

The area is big enough for 10 lines (the first parameter) of about 25 characters each (the second parameter). That is, the parameters tell the height and width in that order. **JTextArea** is in the `javax.swing` package. The text area has a text string that it displays, initially of length zero. You may then add a string of characters to the information that appears in the text area with an `append` statement:

```
area.append ("testing");
area.append ("\n a JTextArea");
```

The text area will then have two lines to display, "testing" and "a JTextArea". The following statement displays the text area in a JOptionPane window:

```
JOptionPane.showMessageDialog (null, area);
```

Note that you may supply a JTextArea object in place of the String object you have previously used. All that is required is that the second parameter of `showMessageDialog` be a displayable Object. You may then append a few more lines, so that the next time you execute the `showMessageDialog` statement, all the lines appear, including those that appeared earlier. But if you want to start fresh with a new string of characters, use something like `area.setText("new stuff")`.

The JScrollPane class

The user will want to be able to scroll up and down in a JTextArea when the amount of information becomes quite large. You may add a scrollbar to the JTextArea object named `area` with the following statement:

```
JScrollPane scroll = new JScrollPane (area);
```

JScrollPane is in the `javax.swing` package. The JScrollPane constructor takes a JTextArea object as its parameter so the scrollbar object knows what to wrap itself around. When you are ready to display it, you use the following statement:

```
JOptionPane.showMessageDialog (null, scroll);
```

Example of using a text area

Listing 6.6 illustrates the use of these standard library methods. It creates a scrolled text area 10 characters tall and 15 'm' characters wide (i.e., using the width of the letter 'm' to measure). It then prints all of the integers from 1 to 100, each with its square one tab position to the right (that is the meaning of the `"\t"` part). Note that `import javax.swing.*;` saves writing the phrase `javax.swing` in five different places.

Listing 6.6 The PrintSquares application program

```
import javax.swing.*;

public class PrintSquares
{
   /** Print a table of integers 1..10 and their squares. */

   public static void main (String[ ] args)
   {  JTextArea area = new JTextArea (10, 15);               // 1
      JScrollPane scroller = new JScrollPane (area);         // 2
      String table = "Table of integers and their squares"; // 3
      for (int k = 1;  k <= 100;  k++)                       // 4
         table += "\n" + k + "\t" + k * k;                   // 5
      area.append (table);                                   // 6
      JOptionPane.showMessageDialog (null, scroller);        // 7
      System.exit (0);                                       // 8
   }  //=====================
}
```

Figure 6.6 shows what the dialog box could look like at the end of execution of the PrintSquares program. The scroll bar on the right indicates that it displays only about 10% of the total output. The width was deliberately chosen to be somewhat smaller than needed so you can see where it cuts off (at about 32 smaller characters).

You have now seen two ways a program can send data outside itself: terminal window (`System.out.println`), and modal graphic (`showMessageDialog`). The dialog box method is called modal because the program pauses until the user clicks a response.

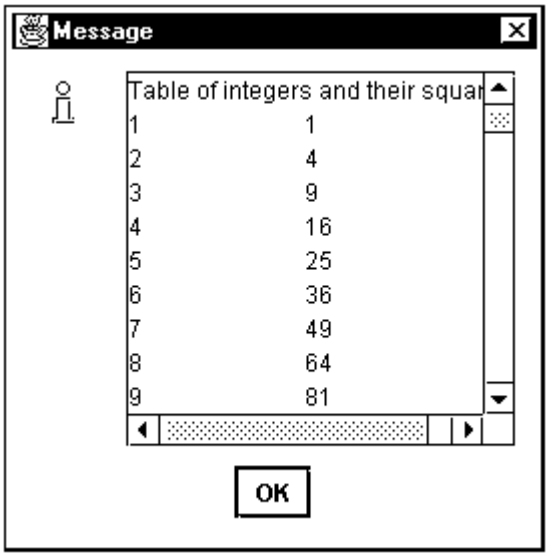


Figure 6.6 Screen shot of JTextArea

The text area and the scroll pane work together to accomplish a single task, displaying to the user a view of the data produced by the program. So these two objects should be logically represented as a single object, which we call a View object. The RepairShop software will require such an object, as defined in Listing 6.7.

Listing 6.7 The View class of objects

```
import javax.swing.*;

public class View extends Object
{
   private JTextArea area;
   private JScrollPane scroller;


   public View (int rows, int columns)
   {  area = new JTextArea (rows, columns);
      scroller = new JScrollPane (area);
   }  //======================

   /** Show the message in a scrolled text area of a dialog. */

   public void display (String message)
   {  area.append (message);
      IO.say (scroller);
   }  //======================
}
```

This class definition lets you replace lines 1 and 2 in the earlier Listing 6.6 by

```
   View output = new View (10,15);
```

and replace lines 6 and 7 by

```
   output.display (table);.
```

Exercise 6.43* Revise Listing 6.1 to use the View class for all output.
Exercise 6.44** Revise the Mastermind game in Listing 4.9 to display all output so far each time it shows the updated status, as well as remind the user of the number chosen.

6.7 Analysis And Design Example: The RepairOrder Class

Your initial analysis of the RepairShop software showed the need for decimal numbers and for objects that could handle sophisticated input and output, which you now have. It also showed the need for objects that represent work orders. So the next task is to create a RepairOrder class.

First you consider what kind of behavior a work order must have. You should be able to ask a work order object for the client's name. You should also be able to ask it for the kind of repair work to be done. The foreman will then look at the job to make an estimate of how much time the repair will take. So you need to be able to tell a work order to remember that estimate so you can retrieve it later. Finally, it will be useful to be able to tell a work order that has been completed to send a bill to the client.

Now that you know the behavior that a RepairOrder object should exhibit, you can see what knowledge a RepairOrder object needs, that is, its instance variables. You should store the client's first and last name and the repair work that has to be done. That is three String values. These values can be determined at the time the work order is first created. The estimate of the time can be a fractional number of hours (e.g., 1.75 for 1 hour and 45 minutes), so a double variable is suitable for storing that value.

Finally, most object classes should have an instance method that supplies all or most of the information stored in an object of the class in a printable form. Such a method often comes in handy. The name `toString` is the conventional Java name for the String analog of an object.

Stubbed documentation

This leads you to develop documentation for the RepairOrder class as the sketch shown in Listing 6.8 (see next page). The methods in this documentation are **stubbed**. That is, the body of each has the minimum necessary to make it compile -- nothing if a void method, a simple return otherwise (`null` if it returns an object).

A stubbed class such as this is just for discussion and development purposes. The only reasons for making it compilable are (a) it is a format understood by all Java programmers who might be in on the discussion, and (b) there is something satisfying about having it compile.

Often you put something more in stubbed methods, such as `IO.say ("now in method XXX")`. This would be because you want to actually run a program that calls on those methods, but you are not yet ready to develop their logic. All you would be doing is testing out some other methods, perhaps seeing how the user interface looks, that require that these exist.

Figure 6.7 shows how a typical RepairOrder object is represented. The RepairOrder variable named `ralph` is an object reference, so its value is shown as an arrow pointing to a rectangle with no name (because objects themselves are not named variables). That rectangle contains four variables, three of which contain String references. Since Strings are objects, their values are also shown as arrows pointing to rectangles.

Listing 6.8 Documentation for the RepairOrder class

```
public class RepairOrder extends Object  // stubbed documentation
{
   private String itsFirstName;
   private String itsLastName;
   private String itsRepairJob;
   private double itsEstimatedTime;


   /** Create a RepairOrder from a line of input.  */
   public RepairOrder (String par)                          { }

   /** Return the client who owns the car. */
   public String getClient()                    { return null; }

   /** Return the kind of repair job to be performed. */
   public String getRepairJob()                 { return null; }

   /** Return the estimate of the time to do the job. */
   public double getEstimatedTime()              { return 0.0; }

   /** Record the estimate of the time to do the job. */
   public void setEstimatedTime (double time)               { }

   /** Send a bill to the client. */
   public void sendBillToClient()                           { }

   /** Return a String value containing most or all of the
    *  RepairOrder's data, suitable for printing in a report. */
   public String toString()                     { return null; }
}
```

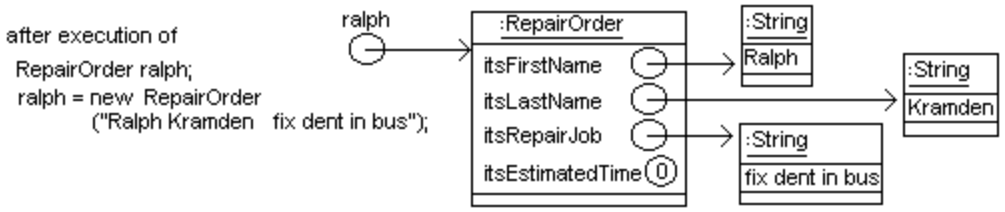


Figure 6.7 UML object diagram for a RepairOrder variable

A reasonable definition of the RepairOrder class is in Listing 6.9 (see next page). The constructor and the `sendBillToClient` methods are the only ones that cannot be done in a single statement. We will skip the `sendBillToClient` method in this development, since we do not need it for the key logic of scheduling repair jobs.

The `toString` form of an object typically gives the value of most or all of the instance variables of the object. It is reasonable for RepairOrder objects to supply all four of these attributes, separated by punctuation and blanks.

Listing 6.9 The RepairOrder class

```
public class RepairOrder extends Object
{
   private String itsFirstName = "";
   private String itsLastName = "";
   private String itsRepairJob = "";
   private double itsEstimatedTime = 0.00;


   public RepairOrder (String par)
   {  super();
      if (par != null)
      {  StringInfo si = new StringInfo (par);
         si.trimFront (0);  // remove any leading whitespace
         this.itsFirstName = si.firstWord();

         si.trimFront (this.itsFirstName.length());
         this.itsLastName = si.firstWord();

         si.trimFront (this.itsLastName.length());
         this.itsRepairJob = si.toString();
      }
   }  //======================

   public String getClient()
   {  return itsFirstName + " " + itsLastName;
   }  //======================

   public String getRepairJob()
   {  return itsRepairJob;
   }  //======================

   public double getEstimatedTime()
   {  return itsEstimatedTime;
   }  //======================

   public void setEstimatedTime (double time)
   {  if (time > 0.0 && time < 20.0)
         itsEstimatedTime = time;
   }  //======================

   public String toString()
   {  return itsLastName + ", " + itsFirstName + ": "
            + itsRepairJob + ", time= " + itsEstimatedTime;
   }  //======================
}
```

Development of the RepairOrder constructor

To develop the logic for the constructor, you first need to make sure that you have not been passed no String at all; if you have, you leave the instance variables with their initial values.

If the String parameter is not `null`, you need to strip off any initial blanks (and other whitespace). Then you get the first word and store it as the first name of the client for this repair job. You should use a StringInfo object to do this (as defined in the earlier Listing 6.4). Then you can use the `trimFront` and `firstWord` instance methods from that class.

What remains is the part of the String parameter that comes after that first word; you can find that easily by taking the substring of `par` that begins with the character after the first word. You repeat the process to get the second word, which is the client's last name. Whatever is left must be the description of the repair job.

You may be thinking that the RepairOrder constructor could produce some kind of error if the given string of characters does not have any non-blank characters. However, there is really no problem in this case. After the first name is removed, the string that is left is the empty string, `trimFront` will leave it the empty string, and then `firstWord` applied to the empty string will return the empty string. The body of the looping statement in each of those two methods is not executed at all, since `itself.length()` is zero.

Exercise 6.45 Revise the `setEstimatedTime` RepairOrder method: The executor asks the user for an estimated time and updates itself accordingly, but only if its current estimated time is zero.
Exercise 6.46 What changes should be made in Listing 6.9 to give each RepairOrder object a unique ID, which is a single capital letter? Provide a method for retrieving the ID of a RepairOrder object. Assume that no more than 26 RepairOrders will ever be created.
Exercise 6.47* What changes would you make in Listing 6.9 to add an instance variable `itsEstimatedCost` that is obtained analogously to `itsEstimatedTime`?
Exercise 6.48* Add a class variable to RepairOrder that keeps track of the total of the estimated times of all RepairOrder objects that have been created. Also add a class method that allows outside classes to access that total estimated time.
Exercise 6.49* Draw the UML class diagram for the RepairOrder class.

6.8 Analysis And Design Example: Model/View/Controller

The software that schedules the repair jobs needs to maintain a queue that contains all the repair jobs left to do. A queue is just a list of things for which the primary operations are adding a new value and removing an old value; the latter operation always removes the value that has been on the list for the longest period of time.

Analysis

You talk with the client to clear up some details of how the RepairShop software is to behave. For one thing, you need to know how many jobs to allow space for. The client tells you that it could be anywhere from 5 to 20 jobs in one day, so you decide to make sure the queue has room for at least 100 (just to be safe).

You also ask how the client wants to indicate whether the next operation to process is (a) to enter a new repair job or (b) to remove an existing repair job from the queue for processing. The client tells you that it will be convenient to enter the letter `'A'` to indicate adding a new job. Since clients are often not sensitive to the fact that capital letters are not interchangeable with lowercase letters, you check back and verify that either `'A'` or `'a'` is to indicate adding a new job.

Logic Design

This all leads to the design of the main logic shown in the accompanying design block, after extensive thought about how to go about it.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic, first draft

1. Create a queue named `jobQueue` that can hold up to 100 jobs (far more than might be required).
2. Ask the user for the first repair job and add it to the `jobQueue`.
3. Repeat the following until the user decides to quit...
    - 3a. If the user wants to add a new job at this point then...
        - 3aa. Ask the user for that repair job and add it to the `jobQueue`.
    - 3b. Otherwise, if the `jobQueue` is not empty...
        - 3ba. Remove the next job to do from the `jobQueue` and display it.
    - 3c. Report the total estimated time for all jobs still in the `jobQueue`.

After you develop this design, you study it carefully before going further, to make sure it is right. When you do, you can see a defect: It makes no provision for calculating the total estimated time. You cannot report to the user values that you have not calculated. So you expand the main logic to allow for these calculations. You also did not create an object to display information to the user. The View kind of object from Listing 6.7 is intended for this purpose. So the revised plan is as shown in the larger accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic, second draft

1. Create a View object to display the information to the user.
2. Create a queue named `jobQueue` that can hold up to 100 jobs (far more than might be required).
3. Ask the user for the first repair job and add it to the `jobQueue`.
4. Initialize a running total of estimated times to that first job's estimated time.
5. Repeat the following until the user decides to quit...
    - 5a. If the user wants to add a new job at this point then...
        - 5aa. Ask the user for that repair job and add it to the `jobQueue`.
        - 5ab. Add to the running total to include that new job.
    - 5b. Otherwise, if the `jobQueue` is not empty...
        - 5ba. Remove the next job to do from the `jobQueue` and display it.
        - 5bb. Subtract from the running total to allow for that removed job.
    - 5c. Report to the user the running total estimated time for all jobs remaining.

Object Design

You already have a description of RepairOrder objects, IO objects, and View objects. Clearly you need a Queue object as well, with the capabilities of adding a value, removing a value, and telling you whether or not it is empty. The Sun standard library does not contain a Queue class. Fortunately a friend of yours has a Queue class handy, thoroughly debugged and tested, that you can use (the full implementation of that Queue class is at the end of the next chapter). Its available operations include the following:

- `new Queue()` creates a Queue object that can hold any number of jobs.
- `jobQueue.enqueue (x)` adds the job `x` to the queue.
- `jobQueue.dequeue()` removes and returns the next available job on a first-in-first-out basis.
- `jobQueue.isEmpty()` tells whether the queue is empty.

The overall **object design** for the RepairShop software is the information in Figure 6.8, together with a description of what services each of these methods provides.

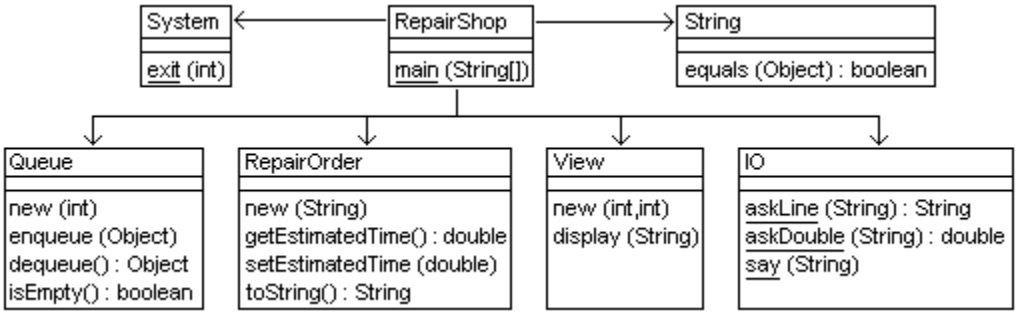


Figure 6.8 UML class diagram for RepairShop

Your friend did not know, when she wrote this Queue class, that you would want it for storing RepairOrder objects. It would be of limited value and not reuseable if it only stored RepairOrder objects anyway. She wrote it to store Object objects, which is the most general (and therefore the most useful) kind of object to store. Reminder: The Object class is the superclass of every class.

The only problem with this is that the method call `jobQueue.dequeue()` produces an Object value, and the compiler will not let you store an Object value in a RepairOrder variable (you cannot mix up object types like that). It would be like putting an ordinary person on a rodeo bull when the rules say that only a cowboy can ride a rodeo bull (if you think this is another metaphor coming up, you are mistaken; it is a simile).

If `x` is a reference to an object and ClassName is the name of some class, then the class cast `(ClassName) x` is a reference to an object of type ClassName. It is as if you put a cowboy hat on it so the compiler will think of it as a cowboy.

There is, however, a caveat: If during execution of the program the runtime system evaluates the phrase `(ClassName) x` and finds that `x` is not actually of type ClassName, then it throws a ClassCastException. In other words, not everyone who wears a cowboy hat is a cowboy.

Consequently, `(RepairOrder) jobQueue.dequeue()` produces a RepairOrder value from an Object value. Although this is analogous to the expression `(int) someDouble`, it is substantially different in two ways: The `(int)` cast causes the runtime system to modify the value it gets from `someDouble` by chopping off (truncating) the part after the decimal point; it does not throw an Exception. The class cast does not modify anything, but it causes the runtime system to throw an Exception if the object returned by `jobQueue.dequeue()` is not in fact a RepairOrder object.

The Model/View/Controller pattern

The Java logic for this RepairShop application program follows fairly directly from the previous discussion. Listing 6.10 gives the result. It illustrates the Model/View/Controller pattern as a way of organizing a program:

Listing 6.10 The RepairShop application program

```
public class RepairShop
{
   private static final String ENTER
         = "Enter first name, last name, and description of job";
   private static final String TIMEPROMPT
         = "Estimate the time to complete the repair job";
   private static final String CHOICES = "Enter A to add another"
         + " job, X to exit, anything else to process a job";


   /** Repeatedly get repair jobs from the foreman or display
    *  the next repair job to the foreman, until there are no
    *  more jobs to be done. */

   public static void main (String[] args)
   {  View output = new View (40, 25);                          //1
      Queue jobQueue = new Queue();                             //2
      RepairOrder nextJob = new RepairOrder (IO.askLine (ENTER));
      double totalTime = IO.askDouble (TIMEPROMPT);             //4
      nextJob.setEstimatedTime (totalTime);                     //5
      jobQueue.enqueue (nextJob);                               //6

      String input = IO.askLine (CHOICES);                      //7
      while ( ! input.equals ("X") && ! input.equals ("x"))     //8
      {  if (input.equals ("A") || input.equals ("a"))          //9
         {  nextJob = new RepairOrder (IO.askLine (ENTER));     //10
            nextJob.setEstimatedTime (IO.askDouble (TIMEPROMPT));
            jobQueue.enqueue (nextJob);                         //12
            totalTime += nextJob.getEstimatedTime();            //13
            output.display ("\nHours remaining: " + totalTime);
         }                                                      //15
         else if (jobQueue.isEmpty())                           //16
            IO.say ("No jobs currently in the queue!");         //17
         else                                                   //18
         {  nextJob = (RepairOrder) jobQueue.dequeue();         //19
            totalTime -= nextJob.getEstimatedTime();            //20
            output.display ("\nNext job: " + nextJob.toString()
                        + "\nHours remaining: " + totalTime);
         }                                                      //23
         input = IO.askLine (CHOICES);                          //24
      }                                                         //25

      System.exit (0);                                          //26
   }  //======================
}
```

- One kind of object, both Queue and RepairOrder in this case, stores the data during execution of the program, modifies that data when told, and provides information about the data when asked. This is the **Model** aspect. It does not provide input or output. In more complex software, you may have several different kinds of models
- Another kind of object, **View** in this case, displays the information about the model for the user to see. In more complex software, there may be several views of the data structure (such as a histogram and a numerical table).
- A third kind of object, or in this case just the main method, serves the **Controller** function. Its job is to accept user input (by the IO methods in this case), then send messages to the Model (asking it to update itself or to give information about itself) and to the View (asking it to update the display) as needed.

In Chapter Nine, when you learn about event-driven programming, the Controller function will be fulfilled by the objects that "listen" for the user's click of a button or entry of information in a textfield and respond as described above. This Model/View/Controller pattern is quite common in programming that uses such objects. In general, we delegate the responsibilities of the program to three distinct kinds of objects whose purposes are (a) maintaining the model of an aspect of the real world, (b) displaying information to the user, and (c) obtaining input from the user.

In the typical case, a Controller either tells the View how it changed the model, or it simply informs the View that it changed the Model and lets the View ask the Model how. Either way, the Model is not even "aware" that it participates in an MVC triad.

Programming Style Keep your main method (and all others) down to where they will fit comfortably on one screen. Create objects to carry out significant subtasks so you can avoid excessive complexity in your main logic. Define separate classes of objects to do this. In general, you should rarely have other methods in the class with the main method; this book never does.

Exercise 6.50 Write a second RepairOrder constructor to add to the earlier Listing 6.9: It has two parameters, the string that describes the repair order plus a double value that specifies the estimated time for that job. Revise Listing 6.10 to use this constructor.
Exercise 6.51* Rewrite the first if-condition in Listing 6.10 to use `charAt`. Allow for the empty String.
Exercise 6.52* Revise Listing 6.10 so that the View prints the number of jobs still on the queue. Extra credit: Have a new kind of object keep track of the `totalTime` and the number of jobs, as well as of the queue itself (a smarter kind of queue will do).
Exercise 6.53* Revise Listing 6.10 so that it forces the user to enter again each numeric input that is not a valid numeral or is negative.

6.9 More On Debugging Your Program: Tracing And Type-Checking

Sometimes your program produces the wrong answer, and sometimes it produces no answer at all (because it is stuck in an "infinite loop" or because it crashed). The problem is always either that (a) some variable has a different value from what you expected it to have, or (b) you do not really know what to expect from your own logic.

The latter can only be cured by diligent study and instruction. But in the former case, you can set up a manual trace of the state of every relevant variable. This is a table of the values those variables should have after each step of the program, for one particular set of input values.

Manual tracing

Figure 6.9 is a manual trace of the RepairShop program. You use a small amount of input, just enough to locate the error. For this sample run, assume you first enter a repair job for `sue` of 80 minutes, then a repair job for `bill` of 60 minutes, then process the next available repair job (which should be `sue`).

You put the name of each relevant variable or value at the top of the columns. You need not have a column for a variable that is only for output; instead you put lines in the table (boldfaced in Figure 6.9) showing what values are displayed at which points.

ACTION	TEST	nextJob	jobQueue	totalTime	input
jobQueue=			empty		
nextJob=		sue 1.33	<sue,1.33>		
totalTime=				1.33	
jobQueue=					
input=					A
test isn't "X"	true				
test is "A"	true				
nextJob=		bill 1.0			
jobQueue=			<sue,1.33>, <bill,1.0>		
totalTime=				2.33	
display				**2.33**	
input=					D
test isn't "X"	true				
test is "A"	false				
test jobQueue	false				
nextJob=		sue 1.33	<bill,1.0>		
totalTime=				1.0	
display				**1.0**	

Figure 6.9 Manual trace of RepairShop

Next you fill in the table one line at a time, applying the next step of the program and recording the new value of any variable that changed in that step, until you see that one variable has the wrong value. Figure 6.9 uses for the Queue's values <x,y> to denote a repair job for person `x` of `y` hours and lists the front of the Queue to the left. Note how the boldfaced lines show what you should see on the screen at that point, so you can easily check it against the actual run of the program.

Automated tracing

Sometimes you do not see the error with this process. The problem is that you are misreading some statement in the program or that you do not really understand what some statement does in this context. So your table's final answer is not the one the program produces. Once you find out just which one statement it is, you can usually see how you are misreading or misunderstanding it.

You may need an automated trace instead of a manual one. You should insert the following five statements in the program in Listing 6.10 right after the line whose number begins the `println` parameter. The first statement uses the fact that string concatenation converts the object to `nextJob.toString()`:

```
System.out.println (3 + " job=" + nextJob);
System.out.println (4 + " time=" + totalTime);
System.out.println (7 + " input=" + s);
System.out.println (11 + " job=" + nextJob);
System.out.println (24 + " input=" + s);
```

These five statements print out all the relevant values with a note of the point in the program where the message was written. They print to the terminal window so you do not have to keep clicking the OK button. If you compare these values with your manual table, you should see the first point at which your table went wrong. Then you can see whether the input contained unexpected values or you misunderstood some statement (just before the debug message that disagreed with your table). Note that there is no need to print the values of the variables changed in lines 13, 19 and 20, because the very next statement displays them anyway.

Two difficulties with this technique are (a) the number of debugging message may be so large that they scroll off the screen before you can read them, and (b) you have to put `//` before each debugging line before you release the program for general use. A solution is to have a Debug class with a class method you call to print the debugging output. Then the Debug class can (a) have a counter that does showMessageDialog after each fifteen lines of output, to give you a chance to read the messages, and (b) have a boolean variable you set `true` or `false`, with `true` required before any debugging messages appear. In the latter case, you simply set the boolean to `false` to disable all debugging messages before releasing the program for general use. The development of this class is an exercise.

Type-checking an expression

You should also know how to **type-check** the expressions in your program to make sure that they are acceptable to the compiler. This type-checking process consists of replacing each value or variable by its type and then applying the operators in the proper order to see what type each subexpression produces. The following two examples of type-checking boldface the subexpression to be evaluated next.

The `firstWord` method in the earlier Listing 6.4 has the phrase:

```
if (itself.charAt (k) <= ' ')
```

1. Substitute for each variable and value in the expression its type to get:

```
if (String.charAt (int) <= char)
```

2. Reduce the `charAt` call by first verifying that it is in fact a method in the String class and it really does have a single parameter of int type. You may then substitute for the call its return type, which is char:

```
if (char <= char)
```

3. Reduce the inequality expression by first verifying that the values on both sides of the operator are the same type. You may then substitute for the expression its result type, which for `<=` is boolean. Since an if-statement requires a boolean value inside its parentheses, this is legal:

```
if (boolean)
```

Another example of type-checking

Consider the following statement that assigns a value to a boolean variable. You start by putting in the default executor wherever appropriate when doing type-checking:

```
valueToReturn =  ! this.seesSlot() || numLeft == 0;
```

1. Substitute for each variable and value in the expression its type to get:

```
boolean =  ! Vic.seesSlot() || int == int;
```

2. Reduce the equality expression by first verifying that the values on both sides of the operator are the same type. You may then substitute for the expression its result type, which is boolean:

```
boolean =  ! Vic.seesSlot() || boolean;
```

3. Reduce the `seesSlot` method call by first verifying that it is in fact a method in the Vic class with no parameter. You may then substitute for the call its return type, which is boolean:

```
boolean =  ! boolean || boolean;
```

4. Reduce the not-expression by first verifying that `!` is applied to a boolean value as required. You may then substitute for the expression its result type, which for the not-operator is boolean:

```
boolean =  boolean || boolean;
```

5. Reduce the or-expression by first verifying that `||` is applied to two boolean values as required. You may then substitute for the expression its result type, which for the or-operator is boolean:

```
boolean =  boolean;
```

Since this is an assignment to a variable of the same type, it is legal.

If none of this works, it is time to call your instructor at his home and ask for help. Hopefully this is not on a Sunday evening when the assignment is due Monday and you put off doing it until the last day.

Exercise 6.54* Write the Debug class described in this section and describe the changes in Listing 6.9 that allow you to use the Debug class properly.

*6.10 More On Random, NumberFormat, and DecimalFormat (*Sun Library)*

This section describes methods from the Random, NumberFormat, and DecimalFormat standard library classes that can be quite useful, though they are not mentioned elsewhere in this book. Since this book gives only a selection from the thousands of methods in the hundreds of classes of the standard library, you should research additional features of Java not covered in this book at `http://java.sun.com/docs`. The most important is the **API** (Application Programming Interface) link on that page.

Random objects (from the java.util package)

If you create a Random object using `randy = new Random(someLongValue)`, you set a 32-bit long "seed" value to be used in calculating pseudorandom numbers. It gives the same sequence of random numbers on each run of the program, which is nice for debugging. If you use `new Random()`, it sets the seed from the current time using `System.currentTimeMillis()`.

The basic method used internally by the Random class is `next(anInt)`, which calculates a new seed from the old one and then returns the number formed by the first `anInt` bits of the seed. You may use the following instance methods to obtain pseudorandom numbers, all uniformly distributed except for `nextGaussian`, where `randy` could be any Random object:

- `randy.nextInt()` returns a 32-bit number `next(32)`, since int values have 32 bits.
- `randy.nextInt(limitInt)` returns essentially `next(31) % limitInt`, with minor adjustments.

- `randy.nextLong()` returns a 64-bit number `next(32) * 2^32 + next(32)`.
- `randy.nextDouble()` returns a number from 0.0 to 0.9999..., namely `next(26) / 2^26 + next(27) / 2^53` since the digit part of a double value has 53 bits (the other 11 bits are used to keep track of the exponent part).
- `randy.nextFloat()` returns `next(24) / 2^24`, since a float value is a decimal number stored with 32 bits and the digit part of that value has 24 bits.
- `randy.nextGaussian()` returns a double value that has a standard normal distribution with mean of 0.0 and standard deviation of 1.0, for use in statistics.

NumberFormat objects (from the java.text package)

You sometimes want to print a number rounded to a particular number of decimal places. For instance, if you were to print 0.1 + 0.2, you would not see 0.3; you would see 0.30000000000000004. This is because numbers are stored in binary form. Therefore, Java variables do not store fractional numbers exactly unless the only division they involve is by a power of two. 0.2 is 1/5, and 5 is not a power of two.

You could use a standard library class named **NumberFormat** from the `java.text` package. You would first have to create a NumberFormat object as follows:

```
NumberFormat form = NumberFormat.getNumberInstance();
```

Then you need to set the maximum and minimum number of digits after the decimal point: `setMaximumFractionDigits` causes the NumberFormat object to round off if the String representation of the number has too many digits after the decimal point; and `setMinimumFractionDigits` causes it to add extra zeros if the String representation of the number does not have enough digits after the decimal point. The actual formatting is done by calling the `format` instance method, which has a double parameter and returns a String value. For instance, to have the int or double value in `x` written with exactly two digits after the decimal point, you could use these three statements:

```
form.setMaximumFractionDigits (2);
form.setMinimumFractionDigits (2);
System.out.println (form.format (x));
```

You could modify Listing 6.6 to print the reciprocals of the integers 1 to 100, correct to four decimal places. Put the following among the first few statements of Listing 6.6:

```
NumberFormat form = NumberFormat.getNumberInstance();
form.setMaximumFractionDigits (4);
form.setMinimumFractionDigits (2);
```

If you then replace the for-statement by the following, you will see 1.00 as the reciprocal of 1, 0.50 as the reciprocal of 2, 0.25 for 4, 0.125 for 8, and all other outputs will be rounded to exactly four digits after the decimal point:

```
for (int k = 1;  k <= 100;  k++)
   table += "\n" + k + "\t" + form.format (1.0 / k);
```

You may also have the part of the number before the decimal point be separated by commas into groups of three digits by calling the instance method `setGroupingUsed` with a boolean parameter: `true` adds commas, `false` omits them:

```
form.setGroupingUsed (true);
System.out.println (form.format(4175238)); // prints 4,175,238
```

DecimalFormat objects (from the java.text package)

DecimalFormat is a subclass of NumberFormat. You may use it to easily specify the number of digits in the string representation of a number. You could create three DecimalFormat objects as follows:

```
DecimalFormat minOfTwo = new DecimalFormat ("00");
DecimalFormat withThree = new DecimalFormat ("0.000");
DecimalFormat dollars = new DecimalFormat ("$0.00");
```

- `minOfTwo.format(x)` for a given double or long value `x` returns a String value that is rounded to the nearest integer and has at least two digits, i.e., a one-digit number has a leading zero. If you used `"0000"` in the constructor, each number from -999 to 999 would be printed with enough leading zeros to fill it out to four digits.
- `withThree.format(x)` returns a String value that has at least one digit before the decimal point and exactly three digits after the decimal point (i.e., rounded if it naturally has more than three digits, with trailing zeros as needed to pad it out to three digits). For instance, it returns `"4.700"` when `x` is 4.7 and returns `"-23.625"` when `x` is -23.62547.
- `dollars.format(x)` returns a String value that has exactly two digits after the decimal point and a "floating dollar sign" immediately before the first digit of the number; for instance, it returns `"$14.42"` when `x` is 14.423.

6.11 Review Of Chapter Six

Listing 6.1, Listing 6.3, and Listing 6.4 illustrate most of the Java language features introduced in this Chapter.

About the Java language:

- Variables of **double** type store numbers with decimal points in scientific notation with about 15-decimal-digit accuracy, using 8 bytes (1 **byte** can store 8 binary digits).
- In the sequence of primitive types char => int => long => double from "narrow" to "wide", you may assign a narrower value to a wider variable, which **promotes** the narrower value to a wider one. But assigning a wider value to a narrower variable (such as a double value to an int variable) requires that you cast it, e.g., put an `(int)` **cast** in front of the double value to convert it to an int value. If you combine two different kinds of these values with an operator, the narrower one is promoted. The promotion of a char value is to its **Unicode** value (0 to 65,535).
- `x += y` means `x = x + y`, and similarly for `-=`, `*=`, `/=`, and `%=`.
- An expression of the form `testing ? oneThing : anotherThing` uses the **conditional operator** to obtain a value. Both the question mark and the colon are required. If `testing` is true then `oneThing` gives the value; otherwise `anotherThing` gives the value. Those two parts on either side of the colon have to be of the same type (int or boolean or Worker or whatever).
- The **members of a class** X are the variables, methods, and classes declared inside of X (except of course those declared inside of any method or class that is inside X). A class declared inside another class with the heading `private static class Whatever` is a **nested** class of the containing class. This affects nothing but the visibility of various class members, so it is primarily done to maintain encapsulation.
- A method **throws an Exception** when it notifies the runtime system of a problem that crashes the program (unless you have special statements that are discussed in Chapter Ten). Examples are an **ArithmeticException** (caused by trying to divide an integer by zero), a **NumberFormatException** (thrown by an attempt to parse a string of characters as a numeral when it is not), an **IndexOutOfBoundsException** (thrown

by e.g. `s.charAt(k)` if it is false that `0 <= k < s.length()`), a **ClassCastException** (thrown by `(C)x` when `x` is not in class C or in a subclass of C), and a **NullPointerException** (thrown by e.g. `x.getName()` when `x` is `null`).

- **Comparable** is a category of classes, called an **interface**. It specifies you may have `implements Comparable` in your class heading if your class or its superclass has a method with the heading `public boolean compareTo(Object ob)`. You may then assign objects of your class to a Comparable variable. In general, you may have `implements X` in your class heading if X is an interface for which your class has all the methods that the interface specifies.
- If Sub is a subclass of Super, you may assign a Sub value to a Super variable. At runtime, a call of a method in the Super class with that Sub value as the executor will actually call the method in the Sub class that overrides it (if any).
- To tell the compiler that an object variable or value will contain at runtime a reference to an object from a subclass Sub of the variable's declared class, put a `(Sub)` cast in front of the variable or value. This **class cast** operation can also be used to tell the compiler that the object is of a class Sub that implements a specific interface.
- Variables of **char** type store individual characters. Character literals have apostrophes, as in `'x'`. Five special ones are the newline `'\n'`, tab `'\t'`, quote `'\"'`, backspace `'\b'`, and backslash `'\\'` characters.
- The Unicode encoding assigns consecutive integers to `'A'` through `'Z'`, also to `'a'` through `'z'`, and also to `'0'` through `'9'`. Characters with Unicode values less than or equal to that of a blank are called **whitespace**.
- A **long** value is a whole-number value stored using 64 bits (8 bytes), so it is roughly plus-or-minus 8 billion billion. You may assign a long value to a double variable without an explicit cast, but not to an int or char variable. The boolean, char, int, long, and double categories of values are called **primitive types**.

Other vocabulary to remember:

- A **sentinel-controlled loop** is a loop that repeats for each value in a sequence of values until you reach a special value, different from normal values, signaling the end of the sequence. A **count-controlled loop** repeats until a counter reaches a specified value.
- Some common actions to take inside a loop, illustrated in this chapter, are the Count-cases looping action, the All-A-are-B looping action, and the Some-A-are-B looping action. The latter two perform a **sequential search** for a value or set of values.
- A **String literal** is a sequence of characters enclosed in quotes, e.g., `"5 \t x"`.
- Assigning a char value to an int variable or an int value to a double variable causes a **widening conversion**; the compiler does this automatically. The opposite is a **narrowing conversion**; it requires a cast.
- A class of objects is formed by **composition** if its only or its primary instance variable is an object of another class. Some authors define "composition" to mean that at least one instance variable is an object, including classes where a String is an instance variable.
- A **crash-guard** is a condition that is tested to avoid evaluating an expression that would crash the program if the crash-guard were not tested.
- A **queue** is a list of things for which the primary operations are adding a new value and removing an old value; the latter operation always removes the value that has been on the list for the longest period of time.
- The **object design** for a piece of software is a list of the major object classes it uses and the services that each such class offers. These services can be described by giving the headings of the public methods together with comments saying what each does. Developing the main logic of the software often helps you decide what objects you need.
- The **Model/View/Controller pattern** is explained in Section 6.10.
- A **trace** of a program is a listing of the values that the most important variables have at various points in the execution of the program. It is used for studying and debugging the program.

About the java.lang.String class:

- `someString.compareTo(aString)` is a String query that returns an int value. If `someString` comes after/before `aString` in **lexicographical order** (based on the Unicode values of their characters), then `someString.compareTo(aString)` will be some positive/negative integer, respectively.
- `someString.charAt(indexInt)` is a String query that returns the character at position `indexInt` (numbered from 0 up). `indexInt` is an int value for which `0 <= indexInt < someString.length()`.
- `someString.substring(startInt)` produces the substring of characters starting at index `startInt` in `someString` and going to the end. `startInt` is an int value for which `0 <= startInt <= someString.length()`.

About the java.lang.Math class:

- `Math.PI` is the ratio of the circumference of a circle to its diameter.
- `Math.E` is the natural base of logarithms, roughly 2.718281828.
- `Math.sqrt(x)` returns the square root of the double value `x`.
- `Math.abs(x)` returns the absolute value of `x`. It returns an int if `x` is an int value, a double when `x` is double, a long when `x` is long.
- `Math.min(x, y)` is the smaller of `x` and `y`. It returns an int value when both parameters are ints, a double when both are double, a long when both are long.
- `Math.max(x, y)` is the larger of `x` and `y`. It returns an int value when both parameters are ints, a double when both are double, a long when both are long.
- `Math.pow(x, y)` returns x-to-the-power-y, where `x` and `y` are doubles.
- `Math.random()` returns a "random" double number `r` such that `0 <= r < 1`.
- `Math.cos(x)` returns the cosine of the double radian value `x`.
- `Math.sin(x)` returns the sine of the double radian value `x`.
- `Math.log(x)` is the natural logarithm of the double value `x`, that is, to base e.
- `Math.exp(x)` is e to the power `x`, so `Math.exp(Math.log(x))` is `x`.

About other Sun standard library classes:

- `Double.parseDouble(someString)` returns the double equivalent of the numeral (which can be in ordinary integer or decimal form, or in **scientific notation** such as 6.3E+4). It throws a NumberFormatException if the numeral is ill-formed.
- `System.currentTimeMillis()` returns the long current time in milliseconds.
- `new JTextArea(rowsInt, colsInt)` creates a new JTextArea object, a rectangular output region with a specified number of rows and columns of characters.
- `someJTextArea.setText(someString)` makes the information that appears in a JTextArea executor be the specified String value.
- `someJTextArea.append(someString)` adds text to the information.
- `new JScrollPane(someJTextArea)` creates a new JScrollPane object wrapped around a given JTextArea. A JScrollPane or a JTextArea can be the second parameter of a `JOptionPane.showMessageDialog` method call.

Answers to Selected Exercises

6.1

```
public class FindAverage
{   public static void main (String[ ] args)
    {   JOptionPane.showMessageDialog (null, "The average of three numbers");
        double first = Double.parseDouble (JOptionPane.showInputDialog ("Enter #1:"));
        double second = Double.parseDouble (JOptionPane.showInputDialog ("Enter #2:"));
        double third = Double.parseDouble (JOptionPane.showInputDialog ("Enter #3:"));
        JOptionPane.showMessageDialog (null, "Their average is "
                                        + ((first + second + third) / 3));
        System.exit (0);
    }
}
```

6.2
```
public class Gasoline
{   public static void main (String[ ] args)
    {   JOptionPane.showMessageDialog (null, "Convert marks/liters to dollars/gallon");
        String s = JOptionPane.showInputDialog ("Enter price of one DeutschMark:");
        double marks = Double.parseDouble (s);
        s = JOptionPane.showInputDialog ("Enter price of one liter of gas in DM:");
        double gasPrice = Double.parseDouble (s);
        double cost = marks * gasPrice / 0.264;
        JOptionPane.showMessageDialog (null, "The dollar cost per gallon is " + cost);
        System.exit (0);
    }
}
```

6.3 total = total + turn can be written as total += turn.
power = 2 * power can be written as power *= 2.
given = given / 2 can be written as given /= 2.

6.4
```
public class ConvertDays
{   public static void main (String [] args)
    {   String input = JOptionPane.showInputDialog ("Enter a number of days");
        double days = Double.parseDouble (input);
        double hours = 24.0 * days;
        double minutes = 60.0 * hours;
        double seconds = 60.0 * minutes;
        JOptionPane.showMessageDialog (null, days + " days equals " + hours
                + " hours, or " + minutes + " minutes, or " + seconds + " seconds.");
    }
}
```

6.9
```
public static int askNonNeg (String prompt)
{   int valueToReturn = askInt (prompt);
    while (valueToReturn < 0)
        valueToReturn = askInt (prompt);
    return valueToReturn;
}
```

6.10 in Listing 4.5: return itsHour < 10 ? "0" + itsHour + itsMin : "" + itsHour + itsMin;
in Listing 4.6: itsUsersNumber = (s == null || s.equals ("")) ? -1 : Integer.parseInt (s);

6.14 Replace "return" by "IO.say" in five places and put parentheses around the phrase that followed "return"; then replace "String" by "void" in the heading.

6.15 JOptionPane.showMessageDialog (null, "" + x), because showMessageDialog(null, x) will not compile, since x is an int value, not a String or other object value.

6.16 IO.say (s.substring (3,6)), since the fourth character is numbered 3 and the seventh is numbered 6.

6.17
```
public static boolean sameFirstFive (String one, String two)
{   return one.length() >= 5 && two.length() >= 5
                && one.substring (0, 5).equals (two.substring (0, 5));
}
```

6.18
```
public static int indexOf (String big, String little)
{   int len = little.length();   // to speed up execution
    for (int k = 0;  k <= big.length() - len;  k++)
        if (big.substring (k, k + len).equals (little))
            return k;
    return -1;
}
```

6.22 s.charAt(2) is 'g', s.substring(2) is "gorithm", and s.substring(2,6) is "gori".

6.23
```
public void trimRear()
{   int k = itself.length() - 1;
    while (k >= 0 && itself.charAt (k) <= ' ')
        k--;
    itself = itself.substring (0, k + 1);  // up to and including k
}
```

6.24 itself.charAt (0) is whitespace, so the value returned is itself.substring (0,0), which is "".

6.25
```
public int countRange (char lo, char hi)
{   int count = 0;
    for (int k = 0;  k < itself.length();  k++)
        if (itself.charAt (k) >= lo && itself.charAt (k) <= hi)
            count++;
    return count;
}
```

6.26
```
public char biggestChar()
{   char valueToReturn = (char) 0;
    for (int k = 0;  k < itself.length();  k++)
        if (itself.charAt (k) > valueToReturn)
            valueToReturn = itself.charAt (k);
    return valueToReturn;
}
```
6.27
```
public int lastBiggie (StringInfo par)
{   int length = this.length() < par.length()  ?  this.length()  :  par.length();
    for (int k = length - 1;  k >= 0;  k--)
        if (this.charAt (k) > par.charAt (k))
            return k;
    return -1;
}
```
6.28
```
public boolean hasWhitespace()
{   for (int k = 0;  k < itself.length();  k++)
        if (itself.charAt (k) <= ' ')
            return true;
    return false;
}
```
6.29
```
public int indexOf (char par)
{   for (int k = 0;  k < itself.length();  k++)
        if (itself.charAt (k) == par)
            return k;
    return -1;
}
```
6.30
```
public String initialDigits()
{   for (int k = 0;  k < itself.length();  k++)
        if (itself.charAt (k) < '0' || itself.charAt (k) > '9')
            return itself.substring (0, k);
    return itself;  // since the string has nothing but digits
}
```
6.36
```
public static char toUpperCase (char par)
{   return (par < 'a' || par > 'z')  ?  par  :  (char) (par - 'a' + 'A');
}
```
6.37
```
public static double largestOf4 (double x, double y, double z, double w)
{   return Math.max (Math.max (x, y), Math.max (z, w));
}
```
6.38
```
public class PrintRoots
{   public static void main (String[ ] args)
    {   double input = IO.askDouble ("Enter a number to find many roots of");
        double fourth = Math.sqrt (Math.sqrt (input));
        IO.say ("square: " + Math.sqrt (input) + ", fourth: " + fourth
                    + ", eighth: " + Math.sqrt (fourth));
        System.exit (0);
    }
}
```
6.39 In the seesTwoEmpty method, the statement if (seesCD()) return false; is guarded by the initial if (! seesSlot()), a Type 3 crash guard. The test if (seesSlot()) is a Type 1 crash guard for the statement beginning if (! seesCD()). hasTwoOnStack has no guards.

6.45
```
public void setEstimatedTime()
{   if (itsEstimatedTime == 0)
        itsEstimatedTime = IO.askInt ("Enter a new estimated time:");
}
```
6.46 Add these two statements to the constructor: itsID = theID; theID = (char) (theID + 1);
Also add the following declarations to the RepairOrder class:
```
private static char theID = 'A';
private char itsID;
public static char getID()
{   return itsID;
}
```
6.50
```
public RepairOrder (String description, double time)
{   super();
    if (description != null)
        getTheInfo (description);  // method whose body is the body of the if-statement in the
    itsEstimatedTime = time;       // other constructor.  Have the other constructor call getTheInfo too.
}
```
Replace the two lines after the if condition by the following:
```
nextJob = new RepairOrder (IO.askLine (ENTER), IO.askDouble (TIMEPROMPT));
```
Replace the third and fourth statements of the main method by the following:
```
RepairOrder nextJob = new RepairOrder (IO.askLine (ENTER), IO.askDouble (TIMEPROMPT));
double totalTime = nextJob.getEstimatedTime();
```

7 Arrays

Overview

In this chapter you will learn about arrays in the context of a personnel database system for a commercial company. An array lets you store a large number of values of the same type. Arrays make it easy to implement the Vic class for Chapters Two and Three (since a sequence stores many CDs), the Network classes for Chapter Five (since a Position object stores many Nodes), and the Queue class for Chapter Six (since a queue stores many Objects). You will also see an elementary use of disk files for input.

- Sections 7.1-7.2 design the Worker class of objects and develop two of the many small programs that might be in the Personnel Database software suite.
- Sections 7.3-7.4 introduce arrays and apply them to another program using Workers and to the implementation of the Worker class.
- Section 7.5-7.6 discuss the partially-filled array concept and develop many instance methods for an object that contains a list of Workers. The list varies in size.
- Section 7.7-7.11 describe implementations of the Vic, Network, and Queue classes used in earlier chapters and develop an elementary sorting method for Workers.

You only need to study Sections 7.1-7.6 to be able to understand the rest of the book. The other sections are to give you thorough practice working with arrays.

7.1 Analysis And Design Of The Worker Class

A company offers your consulting firm a contract to develop software to handle all their personnel matters. You head a software engineering team at your firm that will design and build the software. The others on your team are all junior programmers, so you will have to create the basic design and delegate the details to the others. Your talks with the company executives tell you that the software must, among other things:

- Maintain a data file of workers, accepting additions and deletions.
- Go through the data file and print various reports, some with the workers in order of names, some in order of paycheck size, some in order of birth year, etc.
- Input hours worked for each worker and print pay checks, with taxes withheld.

When you look over your notes from your first meeting with the company's management, you realize that this software can be mostly a number of different small to medium main programs working with a common set of objects, including:

- A virtual worker, representing a single employee.
- A virtual input device, representing the keyboard or a data file.
- A virtual output device, representing the screen or printer.
- A virtual file cabinet, storing data for many workers.

Description of the Worker class

You decide to design the Worker class first. You need to be able to perform at least the following operations with Worker objects. As you develop the main logic of various programs, you expect to see the need for additional operations.

- Create a Worker object from a string of characters received as input from a disk file or from the keyboard.
- Update the hours worked each week, so a Worker can compute its pay for the week.
- Access one Worker's data: name, week's pay, birth year, and others.
- Compare two Workers to see which comes first alphabetically by name.
- Obtain a printable form of a Worker for use in reports.

This leads you to develop documentation for the Worker class as the sketch shown in Listing 7.1, with more to be added later by your subordinates. The methods in documentation are **stubbed**. That is, the body of each has the minimum necessary to make it compile -- nothing if a void method, a simple return otherwise (`null` if it returns an object). The name `toString` is the conventional Java name for the method that returns the String representation of an object. If the constructor is given a `null` or the empty String, it signals this by having `getName` return `null`.

Listing 7.1 Documentation for the Worker class, first draft

```
public class Worker implements Comparable //stubbed documentation
{
   /** Create a Worker from an input String, a single line with
    *  first name, last name, year, and rate in that order.
    *  If the String value is bad, the name is made null. */
   public Worker (String input)                                { }

   /** Return the first name plus last name of the Worker.
    *  But return null if it does not represent a real Worker. */
   public String getName()                          { return null; }

   /** Return the Worker's birth year. */
   public int getBirthYear()                          { return 0; }

   /** Return the Worker's gross pay for the week. */
   public double getWeeksPay()                        { return 0; }

   /** Record the hours worked in the most recent day. */
   public void setHoursWorked (double hoursWorked)             { }

   /** Return 0 if equal, negative if the executor is before
    *  ob (executor has an earlier last name or else the same
    *  last name and an earlier first name), positive otherwise.
    *  Precondition:  ob.getName() returns a non-null value. */
   public int compareTo (Object ob)                   { return 0; }

   /** Tell whether one Worker has the same name as another. */
   public boolean equals (Worker ob)               { return false; }

   /** Return a String value containing most or all of the
    *  Worker's data, suitable for printing in a report. */
   public String toString()                          { return null; }
}
```

The condition `x.compareTo(y) < 0` for Worker objects tests whether `x` comes before `y` in a reasonable ordering, and `x.compareTo(y) >= 0` tests whether `x` equals or comes after `y`. They are analogous to `x < y` and `x >= y` for numeric variables. The presence of this method in the Worker class allows it to `implement Comparable` (discussed in Section 6.3). Both `compareTo` and `equals` have the same name and meaning as for String objects.

Reading from a BufferedReader

Information about hundreds or thousands of workers is quite naturally stored in a file on a hard disk, so this Personnel Database software needs a class that provides methods for reading from a disk file. We will call this the Buffin class (Buffin is a subclass of the BufferedReader class from the Sun standard library). The Buffin class is developed in Sections 10.1-10.6, but you do not have to study that material now. All you need for this chapter is how to use the following two Buffin methods that encapsulate reading data from a text file. The statement

```
Buffin file = new Buffin ("A:\\someFile.txt")
```

opens a channel to the data file named `someFile.txt` on the floppy disk and creates a special BufferedReader object. The `\\` sequence is what you must write inside quotes to indicate a single backslash character. Then the statement

```
String value = file.readLine();
```

requests the next available line from that text file. If you have already read all the information from the file, the `readLine` method returns the `null` value as a signal.

If the runtime system cannot open the file for some reason, or if you call `new Buffin` with the empty String as the parameter, all input will be obtained from the keyboard. This feature of the Buffin class lets you write a multipurpose method with a Buffin parameter that gets data from either a file or the keyboard.

If, after reading some data from the file you opened, you decide you want to start over from the beginning of the file, you only need execute `file = new Buffin ("A:\\someFile.txt")` again. Opening a channel again is the only way to go back to read again what you have already read from the file.

Finding the last of three workers

Listing 7.2 (see next page) is an application program that uses Worker objects. The main method creates three Worker objects with input from a file named `workers.txt`. Then it prints out the Worker with the alphabetically last name. The task of finding and printing the last of the three is sufficiently complex that it deserves a separate method. This method is put in a separate utilities class for operations on Comparable objects. The `ordered` method of Listing 6.3 would be another candidate for this CompOp class. The class diagram is in Figure 7.1.

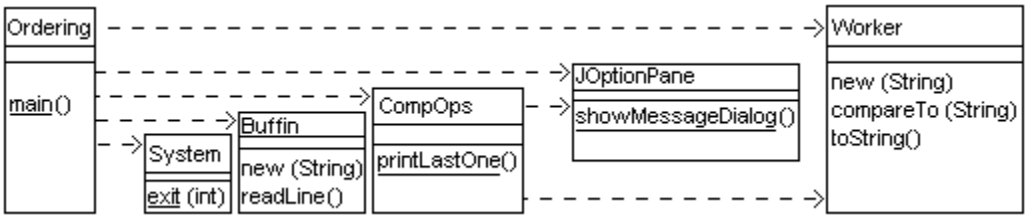


Figure 7.1 UML class diagram for the Ordering program and printLastOne

Listing 7.2 Application program using Workers

```
public class Ordering
{
   /** Ask the user for three Workers.  Print the one that is
    *  last in alphabetical order of names.  Precondition:  the
    *  user enters three useable values at the keyboard. */

   public static void main (String[ ] args)
   {
      javax.swing.JOptionPane.showMessageDialog (null,
                  "reading 3 workers, printing the first.");

      //== READ THREE WORKER VALUES FROM THE KEYBOARD
      Buffin infile = new Buffin ("workers.txt");
      Worker first  = new Worker (infile.readLine());
      Worker second = new Worker (infile.readLine());
      Worker third  = new Worker (infile.readLine());

      //== PRINT THE ALPHABETICALLY LAST
      CompOp.printLastOne (first, second, third);
      System.exit (0);
   }  //=====================
}

//#############################################################3


public class CompOp
{
   /** Return the alphabetically last of three Comparables. */

   public static void printLastOne (Comparable first,
                        Comparable second, Comparable third)
   {  Comparable last =  first.compareTo (second) >= 0
                        ?  first  :  second;
      if (last.compareTo (third) < 0)
         last = third;
      javax.swing.JOptionPane.showMessageDialog (null,
               "The alphabetically last is " + last.toString());
   }  //=====================
}
```

The `printLastOne` method has three parameters of Comparable type. When called from Ordering's main method, the method queries the `compareTo` method in the Worker class twice to decide which of the three is alphabetically last, since the Comparable objects are instances of the Worker class. Then it prints the result.

Polymorphism in the Ordering class

You may wonder how `printLastOne` can compile correctly when it calls `last.toString()` for one of the Comparable objects but `toString` is a Worker method. However, the Object class has a `toString` method to produce a textual representation of an object. The compiler accepts the phrase `last.toString()` because every Comparable object is in a subclass of Object. Therefore, the runtime system can use Object's `toString` method definition unless the object referred to by `last` has its own `toString` method to override the one from the Object class.

The method call `last.toString()` in the `printLastOne` method may call the method in the Worker class or the method in the String class or in some other class, depending on the kind of Comparable objects passed as parameters. This is an example of polymorphism -- a method call that could invoke any of several different methods at different times during the execution of the program.

Key point The compiler does not look at the object referenced by the variable; it only looks at the variable. The object does not exist yet (since it is created at runtime).

Suppose you run a program that calls `CompOp.printLastOne` for three String values. When the runtime system evaluates `last.toString()` in that program, it sees that `last` refers to a String object. So it calls the `toString` method in the String class. That prints the characters in the string.

Key point The runtime system does not look at the variable; it only looks at the object referenced by the variable. The variable does not exist anymore (since it is in the source file `CompOp.java`, not in the compiled file `CompOp.class`).

When the runtime system decides which method to call, it does not look at the type of the variable, it looks at the type of the object. The Comparable variable `last` in the source file is a "polymorph" -- it will sometimes refer to a Worker object and sometimes to a String object (or some other subclass). "Polymorph" comes from the Greek, meaning a thing that takes on several different forms (a sort of shape-changer).

Caution If a class contains a method with the same signature as a method in its superclass, then the two methods must have the same return type. Also, you cannot have one be an instance method and the other be a class method.

Exercise 7.1 Write an independent method `public static void averageDailyPay (Worker karl)`: It prints the given Worker's name and average daily pay (assuming 5 days in a work week).
Exercise 7.2 Rewrite Listing 7.2 by replacing the four statements of the main method beginning with "Worker first" by a single statement. Do not change the effect.
Exercise 7.3 Revise the `printLastOne` method in Listing 7.2 to print the youngest of three Workers.
Exercise 7.4 How would you rewrite Listing 7.2 to have `printLastOne` return the value of `last` instead of printing it? Have one statement in the main method that calls that revised `printLastOne` and prints the value that it returns. Do not change the overall effect of the program.
Exercise 7.5* Write an application program to read three Worker values from the keyboard and then print the average age of those three Workers.
Exercise 7.6* Revise the body of the `printLastOne` method in Listing 7.2 to print the name of that one of the three Workers who has the lowest weekly pay. Also print the amount of that Worker's pay.

7.2 Analysis And Design Example: Finding The Alphabetically First

Problem A data file named `workers.txt` lists thousands of workers. Write an application program to find and print the one with the alphabetically first name.

Analysis (clarifying what to do) You verify that the data file follows a standard format for worker data, so you can retrieve worker information from the data file as a single line to be processed by a Worker constructor. Then the `getName` method applied to that worker information produces the worker's name in the form needed to decide on alphabetical ordering, except it produces `null` when there was no Worker value to read.

No, wait! The `getName` produces names with the first name first, so a comparison will find the person with the earliest first name. That is probably not what the client wants. You check back with the client and find that she wants workers ordered by last name, with the first name only to break ties. So you need to use the `compareTo` method in the Worker class.

A program should be **robust**, which means that it should handle unexpected or unusual situations in a reasonable manner without crashing (at least terminate gracefully). For this program, the file may be empty, in which case the first Worker object will have no name. In that case, you could just have the program print an appropriate message.

Logic design (deciding how to do it) An initial plan for solving this problem could be as shown in the accompanying design block. It uses a sentinel-controlled loop pattern.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic

1. Create the virtual data file.
2. Attempt to read in the first worker value.
3. If that first worker value does not even exist, then...
   Print an appropriate message.
4. Otherwise...
   4a. Make `answerSoFar` equal to that Worker object.
   4b. For each remaining worker you read from the data file, do...
      If its name is before `answerSoFar`'s, then...
         Replace `answerSoFar` by that Worker object.
   4c. Print the name of the ending value of `answerSoFar`.

Object design and implementation Each step of this design translates to one or two Java statements, using classes you have already defined, so no further analysis is needed. The logic as expressed in Java is shown in Listing 7.3 (see next page).

Reminder: `Worker data = new Worker (file.readLine());` is the same as the following two statements, but it avoids having an extra variable used in only one line:

```
String input = file.readLine();
Worker data = new Worker (input);
```

Review of Structured Natural Language Design features The SNL approach is that the entire design should be expressed in ordinary natural language (English or whatever you are most comfortable in), with three exceptions:

1. Storage locations should be named where it clarifies the logic (e.g., `answerSoFar`).
2. Use `If whatever then... <stuff> Otherwise... <stuff>` where a choice is made (sometimes you do not need the `Otherwise` part). Indent for the `<stuff>` in each case.
3. Use `For each whatever do... <stuff>` for repetition (or perhaps use `repeat`). Indent for the `<stuff>`.

Test Plans

You will need to test your program. Most people tend not to think seriously about the tests until they have finished the implementation. That is too late for maximum efficiency in the development of software.

A **test set** is a set of input values used for one run of the program, together with the expected output values (what you believe should be produced by the program for that input). You test your program by giving it the input values and then making sure that the output it produces is what you expected.

Listing 7.3 Application program using a data file

```
import javax.swing.JOptionPane;

public class FindEarliestWorker
{
   /** Read a file and display the name of the Worker
    *  that is alphabetically first. */

   public static void main (String[ ] args)
   {  JOptionPane.showMessageDialog (null, "finding the first");
      Buffin file = new Buffin ("workers.txt");
      Worker answerSoFar = new Worker (file.readLine());

      if (answerSoFar.getName() == null)
         JOptionPane.showMessageDialog (null, "No workers!");
      else
      {  Worker data = new Worker (file.readLine());
         while (data.getName() != null)
         {  if (data.compareTo (answerSoFar) < 0)
               answerSoFar = data;
            data = new Worker (file.readLine());
         }
         JOptionPane.showMessageDialog (null,
                     answerSoFar.getName() + " is first of all.");
      }
      System.exit (0);
   }  //====================
}
```

A test set for the FindEarliestWorker program could have the following eight Worker names. At the right of each name is the value that `answerSoFar` should have after the Worker is read. The two lines of output from the program are boldfaced:

```
Finding the first
George Meaney        answerSoFar is George Meaney
Utah Phillips        answerSoFar is George Meaney
Joe Hill             answerSoFar is Joe Hill
Mother Jones         answerSoFar is Joe Hill
Walter Reuther       answerSoFar is Joe Hill
Marcus Garvey        answerSoFar is Marcus Garvey
Bill Heywood         answerSoFar is Marcus Garvey
Samuel Gompers       answerSoFar is Marcus Garvey
Marcus Garvey is first of all.
```

A **Test Plan** is a large enough number of test sets that you can be confident that almost all of the bugs are out of the program. Begin to develop your Test Plan when you are finishing the Analysis stage of the development. Do not even start the Design stage until you have your Test Plan at least partially planned. Three reasons why it is important to consider the Test Plan before the Design stage are:

1. As you make up a test set, you are reviewing the results of your Analysis stage. You are likely to find many of the errors that are in your analysis. And it is far more efficient to remove the errors before you design rather than after.
2. You get a deeper understanding of the specifications, which means that you will find it easier to design the software.
3. If you have much difficulty making up the test sets, that shows that you do not understand the specifications well enough, so you could not possibly make up a good design anyway.

Exercise 7.7 Modify Listing 7.3 to print the alphabetically last name.
Exercise 7.8 Add to Listing 7.3 coding to also print the total amount paid to all Workers this week.
Exercise 7.9 Add to Listing 7.3 coding to tell whether every worker was born in the same century.
Exercise 7.10* Modify Listing 7.3 to print the earliest birth year. Do not store the worker with the earliest birth year, only the year itself.
Exercise 7.11* Modify Listing 7.3 to print both the name and the birth year of the person with the alphabetically first name.
Exercise 7.12** Add to Listing 7.3 coding to tell whether the workers are in increasing order of names as determined by `compareTo`.
Exercise 7.13* Draw the UML class diagram for `FindEarliestWorker`.

7.3 An Array Of Counters, An Array Of Strings

Problem A data file named `workers.txt` lists thousands of workers, all with birth years in the 1960s. Write an application program to tell how many were born in each one of the 10 years.

Analysis (clarifying what to do) You verify that the data file follows the standard format for worker data, so you can retrieve worker information from the data file as a single line to be processed by a Worker constructor. You should print ten lines after reading through the entire file. The first line should say "1960 had XX workers", the second should say "1961 had XX workers", etc., where XX is filled in by the non-negative count of workers.

What if some birth years in the data file are wrong? You decide to treat the assertion that all birth years are in the 1960s as a prediction to be verified rather than as a precondition to be trusted. That means that your program will be more robust because it will not crash if the prediction is wrong. This more than compensates for being less efficient, caused by spending time checking that the assertion is true.

To handle the exceptional workers, you can print those whose birth years are not in the 1960s as they arise, or print the total number of such exceptions after finishing the file, or simply say whether such exceptions exist. The robustness-check is left as an exercise.

Design (deciding how to do it) You need to have a variable that counts the number born in 1960, a variable that counts the number born in 1961, a variable that counts the number born in 1962, etc. Read the data file one worker at a time using `readLine`. For each worker you read, add 1 to the counter for his or her year. An initial plan for solving this problem is in the accompanying design block. It uses the standard sentinel-controlled loop pattern you saw for Listing 7.3.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic
1. Create the virtual data file.
2. Create ten counters, one for each year, all initially zero.
3. For each worker that you read in from the file, do...
    3a. Add 1 to the counter corresponding to her or his birth year.
4. Print out the list of 10 counters with explanatory comments.

Using an array of counters

You could have 10 int variables named `count0`, `count1`, `count2`, etc. Then you would use the last digit of the birth year to decide which counter to increment. So you would increment `count4` if the birth year is 1964, `count7` if the birth year is 1967. But this makes the logic extremely clumsy.

You can instead use an **array** of 10 int variables, one for each year. When you execute

```
int[ ] count;
count = new int[10];
```

you create 10 counter variables at once (the `[ ]` characters are called **brackets**). Inside the program, you may refer to the one for 1960 as `count[0]`, the one for 1961 as `count[1]`, the one for 1962 as `count[2]`, etc. The `int[]count` part declares `count` to be an object reference. The `count = new int[10]` part creates an object having 10 int variables, each initialized to zero, and has `count` refer to that object. The array appears as shown in Figure 7.2 (the initialized values and also after counting seven years).

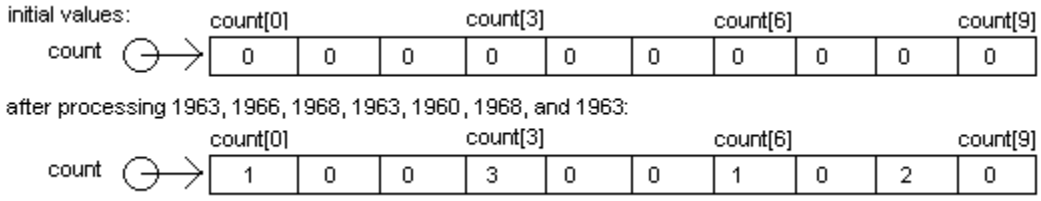


Figure 7.2 An array of 10 counters

Development of the sub-logic

Step 3a of the main logic, finding the right counter to add 1 to, is the step that is most complex. It can be designed as shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for step 3a
1. Get the year that the current worker was born.
2. Subtract 1960 from it to get a number ranging from 0 to 9.
3. Add 1 to the counter indexed by that number.

The implementation of this logic is in Listing 7.4 (see next page). It uses constants for 1960 and 10 to make the logic clearer to the reader and to make the logic easier to modify in the future.

Programming Style Listing 7.4 declares `lastDigit` inside the loop where it is assigned instead of outside it. It is preferable to not declare a local variable any more broadly than it has to be. Some would say this wastes time redeclaring the variable each time through the loop. This logic is faulty for two reasons: (a) a commercial-strength runtime system allocates all space for local variables when the method is called, not during execution of the method, so no time is actually wasted; (b) if that logic were cogent, you would be compelled to also "save time" when you code the Worker constructor by making all its local variables into class variables, since it is called once each time through that same loop. But no one does that.

Listing 7.4 Application program using an array of counters

```
import javax.swing.JOptionPane;

public class CountBirthYears
{
   private static final int TIME_SPAN = 10;
   private static final int YEAR = 1960;


   /** Read a file and display the number of Workers born
    *  in each of the years 1960 to 1969. */

   public static void main (String[ ] args)
   {
      //== INITIALIZE VARIABLES
      JOptionPane.showMessageDialog (null,
                  "Counting those born in the " + YEAR + "s.");
      Buffin buffer = new Buffin ("workers.txt");
      int[ ] count = new int [TIME_SPAN];  // all initially zero

      //== READ THE WORKER DATA AND UPDATE THE COUNTS
      Worker data = new Worker (buffer.readLine());
      while (data.getName() != null)
      {  int lastDigit = data.getBirthYear() - YEAR;
         count[lastDigit]++;  // error check left as exercise
         data = new Worker (buffer.readLine());
      }

      //== PRINT THE ANSWERS
      for (int k = 0;  k < TIME_SPAN;  k++)
         JOptionPane.showMessageDialog (null,
                     (YEAR + k) + " had " + count[k] + " workers");
      System.exit (0);
   }  //=====================
}
```

When you create an array, you must specify its length. For instance, `count = new int[10]` gives that new array a length of 10. The length cannot change later in the program. You get a **NegativeArraySizeException** if the specified length is negative. And you get an **ArrayIndexOutOfBoundsException** if you refer to `count[k]` when `k` is negative or greater than 9. ArrayIndexOutOfBoundsException is a subclass of IndexOutOfBoundsException. Another of its subclasses is **StringIndexOutOfBoundsException**, thrown e.g. by `s.charAt(-2)`.

You can retrieve the length of an array using `length`, a final instance variable for arrays. For instance, `count.length` has the value 10. Note that it has no parentheses, in contrast to `s.length()` for a String. The ten variables in the `count` array are called the array's components, e.g., `count[4]` is the **component** whose **index** is 4 and `count[6]` is the component whose index is 6. Java arrays are **zero-based**: The first component has index 0.

If you want to test-run the program in Listing 7.4 using input from the keyboard, you only have to make sure that the `workers.txt` file does not exist. Then the Buffin will automatically switch to keyboard input. When you want to stop entering data from the keyboard, you may use either CTRL/D or CTRL/Z (depending on the operating system you are using). That causes `readLine()` to return a `null` value.

Using an array of Strings

If you want to print the year in words, you will find it easiest to use an array of words for numbers. Here is one way to make the array:

```
String [ ] unit = new String [10];
unit[0] = "";
unit[1] = "-one";
unit[2] = "-two";
unit[3] = "-three";
unit[4] = "-four";
unit[5] = "-five";
unit[6] = "-six";
unit[7] = "-seven";
unit[8] = "-eight";
unit[9] = "-nine";
```

Then the next-to-last statement of Listing 7.4 could be replaced by the following:

```
JOptionPane.showMessageDialog (null, "Nineteen sixty"
        + unit[k] + " had " + count[k] + " workers.");
```

And the output would begin something like the following:

```
Nineteen sixty had 5 workers.
Nineteen sixty-one had 2 workers.
Nineteen sixty-two had 8 workers.
```

Initializer lists

An **initializer list** is a simpler way to define the values in the `unit` array. It can only be used at the time the array is declared. The following statement uses an initializer list to do the same for `unit` as what was just shown. The array is created with a length of 10 because 10 values are listed:

```
String [ ] unit = {"", "-one", "-two", "-three", "-four",
              "-five", "-six", "-seven", "-eight", "-nine"};
```

When you create a new array object, each component is initialized to zero (or `null` if an array of objects). Some people prefer to explicitly initialize the values in an array when it makes a difference what the values are. If you wanted to do that, you could just replace the declaration of `count` in Listing 7.4 by the following use of an initializer list:

```
int [ ] count = {0,0,0, 0,0,0, 0,0,0, 0};
```

Language elements	
You may declare an array reference using:	Type [ ] VariableName = new Type [IntExpression]
or:	Type [ ] VariableName = { ExpressionList }
A component of an array is:	ArrayReference [IntExpression]
And the capacity of the array is:	ArrayReference . length

Exercise 7.14 Improve Listing 7.4 to also guard against a worker with a birth year not in the 1960s by simply ignoring all such workers.
Exercise 7.15 Modify Listing 7.4 to count the number of workers whose name starts with `'A'`, the number for `'B'`, and so forth down to `'Z'`. Ignore workers whose name does not begin with a capital letter. Hint: `int index = something.charAt(0) - 'A'` could replace the assignment to `lastDigit`.
Exercise 7.16 Modify Listing 7.4 to count the number of workers born in every year 1920 through 1984, ignoring birth years outside that range. How many additional lines would

you have in your program if you did not use an array, using `count20`, `count21`, ..., `count84` instead?

Exercise 7.17* Modify the preceding exercise to print the year all in words. Hint: Use the `unit` array plus a similar array for the tens digit.

Exercise 7.18* Improve Listing 7.4 to also print a separate count of all workers whose birth years are not in the 1960s. Avoid a crash for such workers.

Exercise 7.19* Draw the UML class diagram for the CountBirthYears class.

7.4 Implementing The Worker Class With Arrays

Now that you know what kinds of messages you will be sending to Worker objects, you can decide what a Worker object needs to know in order to answer the messages correctly. It needs to store its first name, last name, hourly pay rate, hours worked in each of the past five workdays (one workweek), birth year, address, etc. You decide that the first draft of the Worker class will have just the first five of those values to establish the basic idea. The junior programmers on your team can add the rest later.

When a worker is hired, the employer will know the name, the birth year, and the hourly wage rate at which the worker starts. The hours paid in a given week change from week to week and are not necessarily known at the time you need to construct a Worker object. It makes sense to just initialize the hours worked to a default value of eight hours in each workday.

Based on this, most of the methods for the Worker class are fairly obvious. Listing 7.5 (see next page) contains the coding for four of them. The last two methods in Listing 7.5, whose logic is discussed next, use the constants WEEK and NUM_DAYS.

The setHoursWorked and getWeeksPay methods

A Worker is to keep track of only the five most recent workdays. When the `setHoursWorked` method adds to a worker's data the fact that he/she worked say 9 hours today, that means you should move the data for the previous four days back one slot in the array to make room for it. You thereby lose the data for five days ago. That is, copy `itsHoursPaid[3]` into `itsHoursPaid[4]`, then copy `itsHoursPaid[2]` into `itsHoursPaid[3]`, etc., finally copying the parameter value `hoursWorked` into `itsHoursPaid[0]`.

The `getWeeksPay` method needs to start by adding up the current five days' hours and multiplying that by `itsHourlyRate`. However, if the worker has worked more than forty hours in the past week, the client pays time-and-a-half for the overtime hours. For instance, if the worker has worked 50 hours in the past week, the worker is paid for 55 hours (5 bonus hours for the 10 hours of overtime). This is equivalent to paying the worker 1.5 times the total number of hours worked, then subtracting off 20 hours because the first 40 should not be multiplied by 1.5.

Pictorial representation of a Worker object

Figure 7.3 shows how a typical Worker object can be represented. The Worker variable is an object reference, so its value is shown as an arrow pointing to a rectangle. That rectangle has no name on it, because objects themselves are not named variables. That rectangle contains five values, two of which are Strings. Since Strings and arrays are objects, their values are also shown as arrows pointing to rectangles.

Listing 7.5 The Worker class of objects, part 1

```
public class Worker extends Object implements Comparable
{
   public static final double WEEK = 40.0; // regular work week
   public static final int NUM_DAYS = 5;   // days in 1 work week
   ////////////////////////////////
   private String itsFirstName = null;
   private String itsLastName = null;
   private int itsBirthYear;
   private double itsHourlyRate;
   private double [] itsHoursPaid = {8, 8, 8, 8, 8}; // 1 per day


   /** Return the first name plus last name of the Worker.
    *  But return null if it does not represent a real Worker. */

   public String getName()
   {  return (itsFirstName == null || itsLastName == null)
             ?  null  :  itsFirstName + " " + itsLastName;
   }  //=====================

   /** Return a String value containing most or all of the
    *  Worker's data, suitable for printing in a report. */

   public String toString()
   {  return itsLastName + ", " + itsFirstName + "; born "
             + itsBirthYear + ".  rate = " + itsHourlyRate;
   }  //=====================

   /** Record the hours worked in the most recent day. */

   public void setHoursWorked (double hoursWorked)
   {  for (int k = NUM_DAYS - 1;  k > 0;  k--)
         itsHoursPaid[k] = itsHoursPaid[k - 1];
      itsHoursPaid[0] = hoursWorked;
   }  //=====================

   /** Return the Worker's gross pay for the week. */

   public double getWeeksPay()
   {  double sum = 0.0;
      for (int k = 0;  k < NUM_DAYS;  k++)
         sum += itsHoursPaid[k];
      return sum <= WEEK  ?  itsHourlyRate * sum
             :  itsHourlyRate * (sum * 1.5 - WEEK / 2);
   }  //=====================
}
```

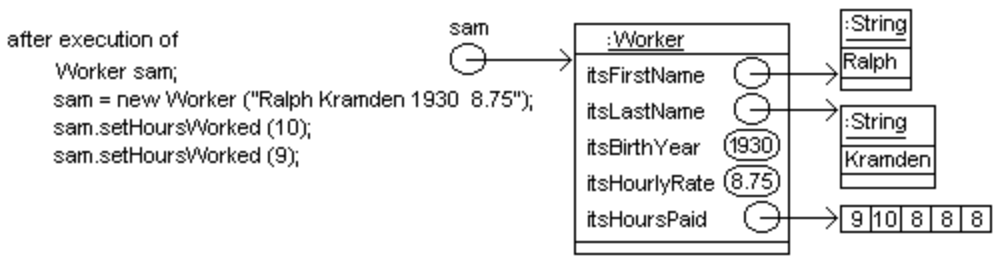


Figure 7.3 UML Object diagram for a Worker object

Programming Style It is good style to follow this Java naming convention: A method named getXXX obtains an attribute of an object without modifying that or any other object. The value may be stored in an instance variable (as for `getName` and `getBirthYear`), or it may be calculated from instance variables (as for `getWeeksPay`). Also, setXXX is conventionally a void method that modifies one or more attributes of the executor object to be the parameter of setXXX.

The remaining Worker methods

Your client wants workers listed in order of last name. Workers with the same last name are to be listed in order of first name. So a reasonable logic for the Worker `compareTo` method is to apply String's `compareTo` method to `itsLastName` for the two workers and use that result. However, if the result of that String comparison is zero, meaning that they have the same last name, apply String's `compareTo` method to `itsFirstName` for the two workers and use that result. The coding for the `compareTo` method is in Listing 7.6 (see next page), along with the coding for an `equals` method which will come in handy; it simply calls on the `compareTo` method to do all the work.

The constructor with a String parameter first makes sure the String value is acceptable. If it is `null` or contains nothing but whitespace, the constructor leaves the names with the `null` value as a signal to the caller of the method that there is no more valid input.

If input comes from the keyboard, the user indicates that all the Worker values have been entered by just pressing the ENTER key. This produces the empty String as a sentinel value (i.e., a value that stands at the end of the sequence of values to signal termination of the sequence). If input comes from a disk file, `readLine` returns `null` when it reaches the end of the file. Either way, this constructor leaves the names `null`.

If the string input has non-whitespace characters, it is split into four "words" (portions of non-whitespace characters separated from the other "words" by whitespace) using the StringInfo methods `trimFront` (remove leading blanks) and `firstWord` (the part down to the first blank) defined in Listing 6.4. The constructor uses the four "words" to fill in all instance variables except the work week (which defaults to 40 hours).

Caution No main method or other logic should call `compareTo` with `null` for the parameter. If you have a dot after a variable with the value `null`, you are asking an object that does not exist to carry out an action or look at its instance variables. This does not make sense, and the runtime system tells you so -- it throws a NullPointerException.

Exercise 7.20 Explain why it does not work to have the heading of the for-statement in the `setHoursWorked` method be `for (int k = 1; k < NUM_DAYS; k++)`.
Exercise 7.21 Revise the `getWeeksPay` method to pay time-and-a-half for any hours in excess of 8 in a day, summed over all five days.
Exercise 7.22* Revise the Worker class for six-day workweeks, retaining the hours worked in the most recent six days.

Listing 7.6 The other methods in the Worker class of objects

```
//public class Worker, continued

   /** Compare two workers based on their last names,
    *  except based on their first names if the same last name.
    *  Precondition:  ob.getName() returns a non-null value. */

   public int compareTo (Object ob)
   {  Worker that = (Worker) ob;
      int comp = this.itsLastName.compareTo (that.itsLastName);
      return  comp != 0  ?  comp
              : this.itsFirstName.compareTo (that.itsFirstName);
   }  //====================

   /** Create a Worker from an input String, a single line with
    *  first name, last name, year, and rate in that order.
    *  If the String value is bad, the name is made null. */

   public Worker (String par)
   {  super();
      StringInfo si = new StringInfo (par);
      si.trimFront (0);
      if (si.toString().length() > 0)
      {  itsFirstName = si.firstWord();

         si.trimFront (itsFirstName.length());
         itsLastName = si.firstWord();

         si.trimFront (itsLastName.length());
         itsBirthYear = (int) si.parseDouble (-1);

         si.trimFront (si.firstWord().length());
         itsHourlyRate = si.parseDouble (-1);
      }
   }  //====================

   public boolean equals (Worker par)
   {  return par != null && this.compareTo (par) == 0;
   }  //====================

   public int getBirthYear()
   {  return itsBirthYear;
   }  //====================
```

7.5 Analysis And Design Example: Finding The Average In A Partially-filled Array

One of the programs that the client needs for the Personnel Database Software is to list all workers who make more than fifty percent above the average. Discussion with the client clarifies the requirements: The workers are listed in the `workers.txt` file and the average is to be computed on the values that `getWeeksPay()` returns.

You need to keep an array of all the Workers in RAM in order to solve this problem efficiently. The number of workers changes with hirings and firings; the client says it is currently almost 1600. So you decide the program will use an array of capacity 5000 to

store the Worker objects. That should be more than enough for many years. You will of course keep track of the actual number of Workers stored in the array at any given time.

Design

The overall design is to first read all the workers into an array big enough to hold them, filling components 0, 1, 2, 3, etc., leaving the unneeded components at the end of the array. Then add up the week's pay for all workers to calculate the average pay (be careful not to divide by zero when the data file is empty). Finally, print out a description of each worker whose pay is greater than 1.5 times the average. If you name the array `item`, and you track the number of workers read in a variable named `size`, then the useable values are in the range `item[0]` through `item[size-1]` inclusive.

Implementation of the design

The key sublogic here is filling in the array. The overall structure of the loop to read many Workers is similar to the loop in Listing 7.4, namely:

```
Worker data = new Worker (file.readLine());
while (data.getName() != null)
{  // do whatever is appropriate
   data = new Worker (file.readLine());
}
```

The appropriate thing to do here is assign data to the next available component of the array. When the loop begins, `size` is initially zero. On the first iteration, we assign `item[0]=data`, so we increment `size` to 1. On the second iteration, we assign `item[1]=data`, so we change `size` to be 2. On the third iteration, we assign `item[2]=data`, which means that `size` is now 3. In general, the appropriate thing to do on each iteration is as follows:

```
item[size] = data;
size++;
```

Although 5000 components should be far more than enough for a few years, we cannot be sure that the situation will not change drastically in a few months. We should refuse to accept any more Worker values if we do not have room for them. So the while condition should be `(data.getName() != null && size < item.length)`.

Now that we have put values in the array, the logic is much the same as for "full arrays" as in Listing 7.5 (array length of NUM_DAYS) and Listing 7.4 (array length of TIME_SPAN) except that we use `size` in place of NUM_DAYS or TIME_SPAN. To add up the week's pay for all workers, we go through the components indexed from 0 to `size` (i.e., not including `size`), adding each one's `getWeeksPay()` to a running total. To print out the highly-paid Workers, we go through the components from 0 to `size`, printing each one for which `getWeeksPay()` is too large. The coding is in Listing 7.7 (see next page).

Programming Style The only reason for the `highlyPaid` variable in the last part of the logic of Listing 7.7 is to avoid evaluating `1.5 * averagePay` many times over inside the loop, each time getting the same result. In general, if your logic calls for evaluating the same expression three or more times, each time getting the same result, it is more efficient to evaluate it once and store it in a local variable for later use. In this particular situation, the logic shown "factors out" the expression, moving the calculation outside the loop.

Listing 7.7 Application program using an array of Workers

```
import javax.swing.JOptionPane;

public class HighlyPaid
{
   public static final int MAX_WORKERS = 5000;

   /** Read a file of up to 5000 Workers and display those who
    *  make more than fifty percent above the average pay. */

   public static void main (String[ ] args)
   {
      //== INITIALIZE VARIABLES
      JOptionPane.showMessageDialog (null, "highly paid people");
      Buffin file = new Buffin ("workers.txt");
      Worker[ ] item = new Worker[MAX_WORKERS];
      int size = 0;  // number of workers in the array

      //== READ THE WORKER DATA AND ADD EACH TO THE ARRAY
      Worker data = new Worker (file.readLine());
      while (data.getName() != null && size < item.length)
      {  item[size] = data;
         size++;
         data = new Worker (file.readLine());
      }

      //== CALCULATE THE AVERAGE WEEK'S PAY
      double totalPay = 0;
      for (int k = 0;  k < size;  k++)
         totalPay += item[k].getWeeksPay();
      double averagePay = size == 0  ?  0.0  :  totalPay / size;

      //== PRINT THOSE MAKING MORE THAN 50% OVER THE AVERAGE
      double highlyPaid = 1.5 * averagePay;
      for (int k = 0;  k < size;  k++)
         if (item[k].getWeeksPay() > highlyPaid)
            JOptionPane.showMessageDialog (null,
                        item[k].toString());
      System.exit (0);
   }  //=====================
}
```

A partially-filled array

The array in Listing 7.7 is a **partially-filled array** because the number of useable values changes each time you read the file. One conventionally keeps the useable data in components `0` through `size-1` of the array, as shown in Figure 7.4. Of course, the name of the variable that keeps count of the useable values does not have to be `size`.

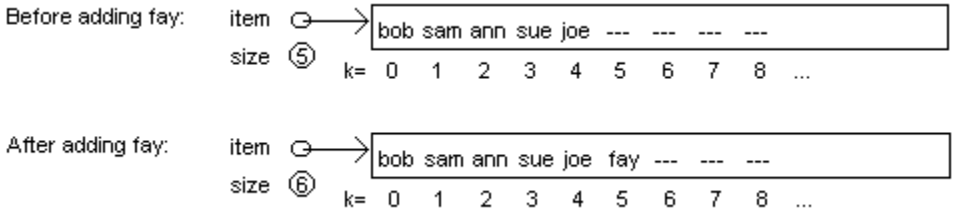


Figure 7.4 A partially-filled array and its size variable

Programming Style Listing 7.7 illustrates the only kind of situation in which you should declare two different local variables of the same method to have the same name (you are never allowed to declare a local variable with the same name as a parameter). The two local variables that control the two for-loops have the same name `k`. Each declaration is only applicable within the for-statement that contains its declaration. As long as neither for-statement contains the other, this is allowed by the compiler.

When you complete a program, you should think it through one more time to make sure nothing could go wrong. You should realize for Listing 7.7 that the company may grow past 5000 workers. In that case, your program silently omits the last few. You need to add a statement somewhere, perhaps the following after the end of the while-loop:

```
if (data.getName() != null)
   JOptionPane.showMessageDialog (null, "Warning: This "
           + "program is skipping some workers.");
```

You will understand loops involving arrays much better if you learn how to describe the loops in ordinary English. When you have a loop with a heading such as `for(k = 0; k < size; k++)` process an array named `item`, you could refer to `item[k]` as "the current item". So the two for-loops in Listing 7.7 could be read as follows:

For each item in the array do...
    Add the current item's `getWeeksPay()` to `totalPay`.

For each item in the array do...
    If the current item's `getWeeksPay()` is larger than `highlyPaid` then...
        Print the String form of the current item.

Generalizing the Worker class

A university may someday come to your company for software that performs many of the same tasks as the Personnel Database Software, but for Students rather than Workers. A doctor may ask for software that performs many of the same tasks, but for Patients rather than Workers. A lawyer may... but you get the idea.

The smart thing to do is to have a superclass from which all of Worker, Student, Patient, Client, etc. can inherit. You can then write much of the software to work with objects of this superclass rather than Workers. Then it will work with Worker objects, since Worker objects inherit all the public methods of the superclass. But it will also work with Patient or Student or Client objects.

You could name the superclass Person. A Person object should have a first name and last name on which the objects are ordered, and probably the year of birth, but not an hourly pay rate or other data specific to employees.

It would take very little extra work to use Person objects in place of Worker objects wherever possible for the Personnel Database Software. You will not of course spend time on Students or Patients or Clients until the need arises. But when another customer asks for similar software, you should be able to reuse over half of your coding. This is called **modular programming** -- you create software for one purpose as a number of modules (methods and classes). Then when you want to use the software for a different purpose, you just pull out a few of the inappropriate modules and slot in others.

Exercise 7.23 Modify Listing 7.7 to print all Workers whose name comes alphabetically before that of the last Worker in the input file.
Exercise 7.24* Modify Listing 7.7 to print all Workers who are older than the average Worker.

7.6 Implementing The WorkerList Class With Arrays

A partially-filled array of Workers is a list of Workers that can be used and modified in many useful ways. The spirit of reusability calls upon us to put various tasks that a list of Workers can perform in separate methods in a suitable class called perhaps WorkerList. This allows us to easily use those methods in several different programs. If we do, the main method in Listing 7.7 could be written in just five statements as follows:

```
public static void main (String[ ] args)
{  JOptionPane.showMessageDialog (null, "highly paid people");
   WorkerList job = new WorkerList (MAX_WORKERS);
   job.addFromFile (new Buffin ("workers.txt"));
   job.printThosePaidOver (1.5 * job.getAveragePay());
   System.exit (0);
}  //======================
```

Implementing the WorkerList class

Listing 7.8 (see next page) shows a start on a WorkerList class. The constructor creates an array of the size specified by the parameter, except that it never makes an array of less than five components. This guards against the error of supplying a negative parameter, which would throw a NegativeArraySizeException. The array is named `itsItem` and the number of useable values in the array is `itsSize`. The logic for the other three methods in Listing 7.8 is virtually the same as in Listing 7.7 (compare them and see).

Figure 7.5 shows a representation of a WorkerList variable as containing an arrow pointing to the WorkerList object. That object's `itsItem` variable is itself an array object reference. So it is also shown as containing an arrow pointing to the object itself.

The figure shows how the WorkerList object looks before and after adding a new Worker to the end of the list. For simplicity, the Worker values are not shown as arrows to yet more objects.

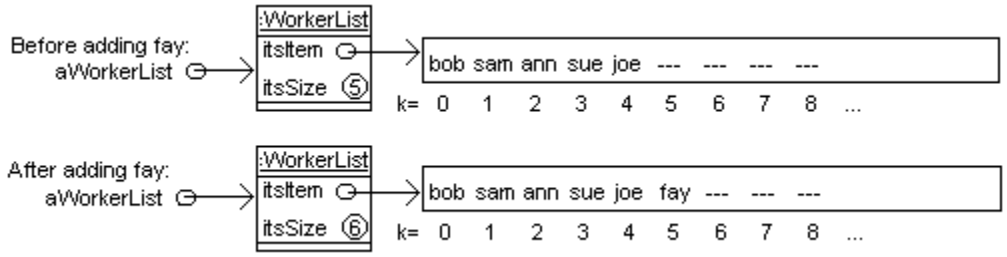


Figure 7.5 A WorkerList object before and after inserting at the end

Listing 7.8 The WorkerList class of objects; some methods added later

```
public class WorkerList extends Object
{
   private Worker[ ] itsItem;
   private int itsSize = 0;


   /** Create an empty list capable of holding max Workers. */

   public WorkerList (int max)
   {  super();
      itsItem = max > 5  ?  new Worker[max]  :  new Worker[5];
   }  //=====================

   /** Add all Worker values in the file to this list, except
    *  no more than there is room for in the list. */

   public void addFromFile (Buffin file)
   {  Worker data = new Worker (file.readLine());
      while (data.getName() != null && itsSize < itsItem.length)
      {  itsItem[itsSize] = data;
         itsSize++;
         data = new Worker (file.readLine());
      }
   }  //=====================

   /** Return the average pay for all workers in the list.
    *  Return zero if the list is empty. */

   public double getAveragePay()
   {  double totalPay = 0;
      for (int k = 0;  k < itsSize;  k++)
         totalPay += itsItem[k].getWeeksPay();
      return itsSize == 0  ?  0.0  :  totalPay / itsSize;
   }  //=====================

   /** Print names of workers making more than the cutoff. */

   public void printThosePaidOver (double cutoff)
   {  for (int k = 0;  k < itsSize;  k++)
         if (itsItem[k].getWeeksPay() > cutoff)
            javax.swing.JOptionPane.showMessageDialog (null,
                        itsItem[k].toString());
   }  //=====================
}
```

Postconditions and internal invariants

The **postcondition** for an action method is a statement of what has changed as a result of calling the method, assuming the precondition for that method has been met. For `printThosePaidOver`, the postcondition is that a list of people with a week's pay higher than the cutoff has appeared on the screen. For `addFromFile`, the postcondition is that each Worker value in the file, up to but not including the first one that would cause the WorkerList to have more than its capacity in Worker values, is stored at the end of the WorkerList in the sequence read. Neither of these two methods has a precondition other than the automatic precondition that the executor is non-null.

An **internal invariant** for a class of objects is a condition on the internal (private) structure of the instances of that class that is (a) true at the time each method is called and (b) true at the time that each method returns. An internal invariant thus describes a precondition of each method as well as a postcondition of each method. When you design a class of objects that stores a number of values, you often have an implicit internal invariant that you rely on in developing your coding. You should write it down so as to make it explicit:

Internal invariant for WorkerLists

1. `itsItem` is an array and `0 <= itsSize <= itsItem.length`.
2. The values in components `itsItem[0]` through `itsItem[itsSize-1]` are the values on the conceptual list of Workers (the list that the user of the class thinks of when he/she uses it) and in the same order, with `itsItem[0]` being the first one.
3. The values stored in `itsItem[itsSize]` and higher do not affect anything.

An internal invariant is called "invariant" because it always remains true no matter what methods are called or in what order. It is called "internal" because the internal logic (the coding of the methods) keeps the precondition for each method true rather than relying on the (external) caller of the method to make the precondition true before the method is called. This particular internal invariant establishes the connection between the abstract concept of a number of values listed in a particular order and the concrete realization of that concept in coding.

The contains method

It is quite useful to have a method that searches the list to see if there is a match for a given Worker. As it goes through the list of Workers one by one, if it ever finds a match, it returns `true`. If the search comes to the end of the list without ever finding a match, the method returns `false`.

You have seen this Some-A-are-B pattern several times in previous chapters, used to find out whether a certain condition is true for at least one of the values in a list or sequence. A reasonable coding is in the upper part of Listing 7.9. Note that the internal invariant justifies the use of the phrase `k = 0; k < itsSize`.

Listing 7.9 Two more WorkerList methods

```
/**  Tell whether par is a value in this WorkerList. */

public boolean contains (Worker par)    // in WorkerList
{  for (int k = 0;  k < itsSize;  k++)
      if (itsItem[k].equals (par))
         return true;
   return false;
}  //=====================

/**  Return a new object, a duplicate of this WorkerList. */

public WorkerList copy()                // in WorkerList
{  WorkerList valueToReturn
             = new WorkerList (this.itsItem.length);
   valueToReturn.itsSize = this.itsSize;
   for (int k = 0;  k < this.itsSize;  k++)
      valueToReturn.itsItem[k] = this.itsItem[k];
   return valueToReturn;
}  //=====================
```

A WorkerList object as a return value

It is perfectly legal to have a method that returns a WorkerList object. For instance, you may at times want to make a second copy of a given WorkerList object. You first make a new WorkerList object with an array of the same size as the executor. Then you fill in the array components with the same values. This logic is in the lower part of Listing 7.9.

It should be clear that you cannot make a true copy of a WorkerList `sam` by declaring `WorkerList sue = sam`. That just makes `sam` and `sue` refer to one and the same WorkerList object. You have two WorkerList variables but only one WorkerList object. It is hopefully just as obvious that you cannot make a true copy by executing `sue.itsItem = sam.itsItem`. You would have two WorkerList objects but only one array of Workers. Then any change you make in `sue's` workers perforce changes `sam's` workers.

Obviously, there are many more methods that should be added to the WorkerList class. But you leave that for the junior programmers on your development team.

Summary for array variables

Java has two categories of objects, array objects and class objects. The only operation you can perform on a class object is to put a dot after it (followed by a method or variable name). The only operation you can perform on an array object is to put brackets after it (with an int value inside the brackets) or find its length.

In general, if `T` is any variable type, then `T[ ]` is also a variable type. A variable of such a type is called an **array variable**. If you declare `x` as a variable of type `T[ ]`, you may define `x = new T[whatever]` using a non-negative int expression inside the brackets. Thereafter, if `k` is an int expression in the range from 0 through `whatever-1`, then `x[k]` is a variable of type `T`.

An array can be a method parameter. For instance, the filled-array-parameter analog of the `printThosePaidOver` method in the earlier Listing 7.8 would be the following class method. Every other example and exercise in this section has a corresponding analog.

```
public static void printThosePaidOver     // independent
           (double cutoff, Worker[] item)
{  for (int k = 0;  k < item.length;  k++)
      if (item[k].getWeeksPay() > cutoff)
         javax.swing.JOptionPane.showMessageDialog (null,
                                  item[k].toString());
}  //======================
```

Exercise 7.25 Write a method `public String alphabeticallyFirst()` in WorkerList: The executor returns the name of the alphabetically first of its Workers. It returns `null` if there are no Workers.
Exercise 7.26 Write a method `public boolean everyonePaidLessThan (double cutoff)` in WorkerList: The executor tells whether all of its Workers have a week's pay that is less than a given double value (the parameter).
Exercise 7.27 Write a method `public boolean inAscendingOrder()` in WorkerList: The executor tells whether all the names of its Workers are in ascending alphabetical order.
Exercise 7.28 Write a method `public void remove (Worker given)` in WorkerList: The executor deletes from its list a single Worker object (if any) equal to a given Worker parameter. Replace it by the last one in the list.
Exercise 7.29 Write a method `public int countSame (WorkerList given)` in WorkerList: The executor tells how many Workers it has in common with the WorkerList parameter (equal Workers in the same position).

Exercise 7.30* Write a method `public Worker oldestWorker()` in WorkerList: The executor returns the Worker object who is oldest (if a tie, return any of the oldest).
Exercise 7.31* Write a method `public boolean equals (WorkerList par)` in WorkerList: The executor tells whether it has the same Workers as the WorkerList parameter (using the `equals` method from the Worker class) in the same order.
Exercise 7.32* Write a method `public WorkerList highOnes()` in WorkerList: The executor returns a new WorkerList object containing only its Worker objects whose pay is above average.
Exercise 7.33* Write a method `public void addAll (WorkerList given)` in WorkerList: The executor adds all the Worker values from a WorkerList parameter to its own list, stopping only if its array becomes full.
Exercise 7.34** Write a method `public void reverse()` in WorkerList: The executor swaps its Worker values around so they are in the opposite order.

7.7 A First Look At Sorting: The Insertion Sort

Consider this problem: You need a program that lists in order of birth year the Workers that are stored in a file. Specifically, you want the Workers to appear on the screen in increasing order of birth year, with ties printed in the same order that they were in the file. A reasonable design for this problem is in the accompanying design block.

Structured Natural Language Design for OrderedByYear
1. Create a WorkerList object.
2. Connect to a file "workers.txt" containing strings that each represent one worker.
3. Repeat until you reach the end of the file or the WorkerList is full...
    3a. Read one Worker value.
    3b. Insert it in the WorkerList before all Workers that have a later birth year.
4. Print all the Workers in the order they occur in the WorkerList.

Once you have this main logic design, you need to see what additional capabilities a WorkerList object must have. A method to find its current size, a method to insert a new Worker in order of birth year, and a method to return a string equivalent of the WorkerList will all be useful. Listing 7.10 contains the fairly obvious coding for the given design.

Listing 7.10 Application program using an array of Workers

```
public class OrderedByYear
{
   public static final int MAX_WORKERS = 5000;


   /** Read a file of up to 5000 Workers and display them
    *  in ascending order of birth years. */

   public static void main (String[ ] args)
   {  System.out.println ("Workers in order of birth year.");
      WorkerList job = new WorkerList (MAX_WORKERS);
      Buffin file = new Buffin ("workers.txt");
      Worker data = new Worker (file.readLine());
      while (data.getName() != null  && job.size() < MAX_WORKERS)
      {  job.insertByYear (data);
         data = new Worker (file.readLine());
      }
      System.out.println (job.toString());
   }  //====================
}
```

Coding the new WorkerList methods

The `size` method simply returns `itsSize`. The `toString` method can combine the `toString()` values of all the Workers together in the order they occur in the WorkerList, with an end-of-line marker between them. This can be useful for when you want to write the values in the WorkerList to a file, to be read in later by another program. These two methods are in the upper part of Listing 7.11.

Listing 7.11 Four more WorkerList methods

```
/** Tell how many Workers are in this WorkerList. */

public int size()                          // in WorkerList
{  return itsSize;
}  //=====================

/** Return a String value representing the entire WorkerList.
 *  It has one Worker's toString() value on each line. */

public String toString()                   // in WorkerList
{  String valueToReturn = "";
   for (int k = 0;  k < itsSize;  k++)
      valueToReturn += itsItem[k].toString() + "\n";
   return valueToReturn;
}  //======================

/** Put data just before all Workers at the end of the
 *  WorkerList that have a larger birth year. */

public void insertByYear (Worker data)   // in WorkerList
{  int year = data.getBirthYear();
   int k = itsSize;
   for ( ; k > 0 && itsItem[k - 1].getBirthYear() > year; k--)
      itsItem[k] = itsItem[k - 1];
   itsItem[k] = data;
   itsSize++;
}  //======================

/** Put all the Workers in ascending order of birth year.  */

public void sort()                         // in WorkerList
{  int save = itsSize;
   itsSize = 1;
   while (itsSize < save)
      insertByYear (itsItem[itsSize]);
}  //======================
```

The `insertByYear` method is more complicated. The obvious place to put a new piece of data is at index `itsSize`; assign `k = itsSize` to indicate that position is available for the new data. However, if the Worker value just below index k has a larger birth year than the new data to be added, that Worker value should be moved up to index k; then subtract 1 from k to indicate that the index one lower is now an available position for putting the new data. Repeat this comparing and subtracting until you see a Worker that does not have a larger birth year or until you make index 0 available. Either way, put the new data value at that now-available index. This logic is in the middle part of Listing 7.11.

Listing 7.11 has one additional method made possible by the existence of the `insertByYear` method. If you have some non-empty WorkerList that is not in increasing order of birth year, and you want to make it so, just call the `sort` method for it. That `sort` method goes through each Worker value in the list, starting from the second one, and inserts it in increasing birth order among all the Worker values that come before it in the WorkerList. This logic is called the **insertion sort logic**. Chapter Thirteen discussed many other kinds of logic that sort a number of values in order.

Exercise 7.35 Write a method `public void insert (Worker given, int n)` in WorkerList: The executor inserts the given Worker at the given index `n`, making room by moving all the values at index `n` and above one component higher, so they remain in the original order. Take no action if `n` is out of range or the array is full.
Exercise 7.36 Write a WorkerList method `public boolean put (Worker par)`: The executor adds `par` at the end of the list, but only if it is not equal to one already there and there is room. It returns true if it actually added the Worker and false if it did not.
Exercise 7.37* Write a method `public boolean containsAll (WorkerList given)` in WorkerList: The executor tells whether every Worker in the given parameter is also in the executor. Call on the `contains` method in Listing 7.9.
Exercise 7.38* Write a method `public void deleteYear (int given)` in WorkerList: The executor deletes every one of its Workers with the specified birth year. The order of those Workers that remain is not specified.
Exercise 7.39* Write a method `public void retainAll (WorkerList given)` in WorkerList: The executor deletes every one of its Workers that is not in the given parameter. The order of those Workers that remain is not specified.
Exercise 7.40** Write a method `public boolean equivalent (WorkerList par)` in WorkerList: The executor tells whether its WorkerList parameter has the same Workers as the WorkerList parameter (using the `equals` method from the Worker class) in some order. Hint: Create a copy of the executor, then write and repeatedly call a private method that tells whether a specified Worker value is in the copy and, if so, replaces it in the copy by `null`.
Exercise 7.41** Write a method that goes through an entire WorkerList, swapping any two adjacent Workers where the first has a larger birth year than the second. Repeat this process as many times as there are Workers in the list. Explain why this is guaranteed to put them in increasing order of birth year. Then explain why it works much more slowly than the sorting logic in Listing 7.11.
Exercise 7.42** Revise your answer to the preceding exercise to stop repeating the swapping process as soon as one pass through the data does not cause any swaps. Discuss whether this improved logic is still much slower than the one in Listing 7.11.

7.8 A First Look At Two-Dimensional Arrays: Implementing The Network Classes

The new language features in this section are not used elsewhere in this book except in Chapter Twelve. They appear at this point only so that you can get a start on the concept of multi-dimensional arrays, which you can then carry further later.

Checkers as an example

A checkerboard for a computer game could be modeled by an 8-by-8 rectangular arrangement of values, some of which represent Checker pieces and some of which represent an empty square. You may declare an array with two indexes to represent the checkerboard, so that `board[0][0]` is the square in the lower-left corner, `board[0][1]` is the next square to its right, etc., up to `board[7][7]` for the upper-right corner. Figure 7.6 specifies the names all of the 64 variables to make this clear.

board[7][0]	board[7][1]	board[7][2]	board[7][3]	board[7][4]	board[7][5]	board[7][6]	board[7][7]
board[6][0]	board[6][1]	board[6][2]	board[6][3]	board[6][4]	board[6][5]	board[6][6]	board[6][7]
board[5][0]	board[5][1]	board[5][2]	board[5][3]	board[5][4]	board[5][5]	board[5][6]	board[5][7]
board[4][0]	board[4][1]	board[4][2]	board[4][3]	board[4][4]	board[4][5]	board[4][6]	board[4][7]
board[3][0]	board[3][1]	board[3][2]	board[3][3]	board[3][4]	board[3][5]	board[3][6]	board[3][7]
board[2][0]	board[2][1]	board[2][2]	board[2][3]	board[2][4]	board[2][5]	board[2][6]	board[2][7]
board[1][0]	board[1][1]	board[1][2]	board[1][3]	board[1][4]	board[1][5]	board[1][6]	board[1][7]
board[0][0]	board[0][1]	board[0][2]	board[0][3]	board[0][4]	board[0][5]	board[0][6]	board[0][7]

Figure 7.6 Checkerboard as an 8-by-8 array named board

If you use int values with named constants RED = 1, BLACK = 2, and EMPTY = 0, then you can declare the `board` variable as follows:

```
int[] [] board;  // variable is declared, array not created
```

Note that it has two pairs of brackets instead of just one, because you want to put two indexes on the `board` variable, not just one. You can initialize it as an array with 64 components in an 8-by-8 arrangements as follows:

```
board = new int [8] [8];  // creates array of 64 components
```

Just as with one-dimensional arrays of numbers, all components are initially zero. You can put four RED pieces on the board, in the bottom row at columns number 1, 3, 5, and 7, as follows:

```
for (int k = 1;  k < 8;  k += 2)
   board[0][k] = RED;
```

You can move a piece from square [3][4] to the northeast as follows:

```
board[4][5] = board[3][4];
board[3][4] = EMPTY;
```

At the end of the game, you can clear off all 64 squares on the board as follows:

```
for (int row = 0;  row < 8;  row++)
   for (int col = 0;  col < 8;  col++)
      board[row][col] = EMPTY;
```

Two-dimensional arrays of objects

You can declare a variable that can refer to a rectangular array of components, each containing a Piece object, and then create the array with all entries `null`, as follows:

```
Piece[] [] item = new Piece [NUM_ROWS] [NUM_COLS];
```

You can then refer to `item[x][y]` in the program as long as `x` has a value of 0 to NUM_ROWS and `y` has a value of 0 to NUM_COLS (including 0 but not including the upper limit). You can assign a value to such a component as follows:

```
item[x][y] = somePieceValue;
```

And you can refer to the value at row number `x` and column number `y` in an expression, for instance:

```
Piece p = item[x][y];
if (item[x][y] == null)...
```

The array `item` is considered a one-dimensional of variables of type `Piece[]`, so `item.length` is the number of rows, which is NUM_ROWS. One of those rows is `item[k]`, and the length of that row `item[k].length` is NUM_COLS.

Implementation of the Network classes

If you did not read most of Sections 5.5-5.7, skip the rest of this section; it discusses an implementation of the Network classes. For this implementation, the Network class stores the connection information in a two-dimensional array `itsNode`. The row of components whose first index is `k` is the set of Nodes that the kth Node connects to; excess components contain `null`. A reference to each Node in the Network is stored in the row of components whose first index is 0, followed by a component containing `null`.

When a Node is created, it is supplied three parameters: its name, whether it is blue, and the row of its connecting Nodes as a parameter. This row is itself a one-dimensional array. When that Node is then asked for a Position object to go through the list of its connections, it creates one with that row of connecting Nodes.

Position objects have two instance variables: one to remember the one-dimensional array that lists the Nodes it is to produce one at a time, and one to remember its current position in the sequence. This should be sufficient information that you could develop the Position and Node classes yourself; it is a major programming project. The Network class in Listing 7.12 (see next page) generates a random number of Nodes with random connections; you could replace that constructor by one that reads connection information from a disk file to handle real-world data.

For the randomizing constructor in Listing 7.12, if the number of Nodes in the Network is e.g. 12, the Network constructor creates a 13-by-13 array for storing the connection information. It creates 12 Nodes to go in the first row. It then computes the probability of one Node connecting to another as 3/11, so that the average Node connects to three others. Finally, it goes through each potential connection of Node `k` to Node `conn` and makes the connection 3/11 of the time.

Language elements
You may declare an array reference using the following, :
Type [ ] [ ] VariableName = new Type [IntExpression] [IntExpression]
You may have more than two pairs of brackets, as long as you have the same number of brackets on the left of such a declaration as you have on the right.

Listing 7.12 The Network class of objects

```
public class Network extends Object
{
   /** Internal invariant:  The sequence of available Nodes is
    *  itsNode[0][k] for 0 <= k < itsNumNodes. The sequence of
    *  Nodes itsNode[0][k] connects to is itsNode[k+1][j] for
    *  0 <= j < n for some n <= itsNumNodes.  The null value
    *  comes after the last value of each such sequence. */

   private Node [][] itsNode;
   private int itsNumNodes;


   public Position nodes()
   {  return new Position (itsNode[0]);
   }  //====================

   /** Create a random Network with 6 to 15 Nodes, about half
    *  blue, and with 3 connections per Node on average.  No Node
    *  connects to itself. This is the only method you need to
    *  replace if you want to obtain Network data from a file. */

   public Network()
   {  java.util.Random randy = new java.util.Random();
      itsNumNodes = 6 + randy.nextInt (10);
      itsNode = new Node [itsNumNodes + 1][itsNumNodes + 1];
      for (int k = 1;  k <= itsNumNodes;  k++)
         itsNode[0][k - 1] = new Node ("Node#" + k,
                                 randy.nextInt(2) == 1, itsNode[k]);

      double probability = 3.0 / (itsNumNodes - 1);
      for (int k = 1;  k <= itsNumNodes;  k++)
      {  // decide which connections the Node #k has
         int j = 0;
         for (int conn = 1;  conn <= itsNumNodes;  conn++)
            if (randy.nextDouble() < probability && conn != k)
            {  itsNode[k][j] = itsNode[0][conn - 1];
               j++;
            }
      }
   }  //====================
}
```

Exercise 7.43 Write statements to have the checker piece at `board[x][y]` jump the piece to its northwest, but only if the square beyond that piece exists and is empty. Do not forget to remove the piece you jump.

Exercise 7.44 Write statements to fill in the top three rows of the `board` with twelve BLACK checker pieces, four per row, alternating as in the initial position of a checkerboard (e.g., one piece goes in `board[7][0]`).

Exercise 7.45 Write an independent method for the checkerboard: `public int numEmptySquares (int[][] board)` returns its number of empty squares.

Exercise 7.46* Write an independent method `public int numNulls (Object[][] par)`: Return the number of null values in the given rectangular array, not counting components with either index being 0.

Exercise 7.47* How would you revise the Network constructor to have each Node k connect to the next three Nodes k+1, k+2, and k+3 (whenever they exist)?

7.9 Implementing A Vic Simulator With Arrays

You have now seen enough of Java that you can understand how the Vic class can be implemented using arrays. It will be quite similar to the simulation in Section 5.5 using Strings (it is very helpful though not necessary to compare the development here with that development). The key difference is that the instance variable `itsSequence` is an array of Strings, each component representing one CD, declared as follows:

```
private String[] itsSequence;
```

That allows you to store the full name of a CD if you wish, rather than faking it with a single letter. The other instance variables remain `itsPos` (a position in the sequence, numbering from 1 up for convenience) and `itsID` (a positive int by which the Vic object is identified). So `getPosition` has the one statement specified in Section 5.5:

```
public String getPosition()            // in Vic
{  return itsPos + "," + itsID;
}  //======================
```

The `backUp` method is the same logic as earlier, except that an attempt to `backUp` when it is illegal calls a `fail` method to explain the problem and then terminate the program:

```
public void backUp()                   // in Vic
{  if (itsPos == 1)
      fail ("backUp from slot ");
   itsPos--;
   trace ("backUp to slot ");
}  //======================

private void fail (String message)     // in Vic
{  System.out.println ("Vic# " + itsID + " crashed on "
          + message + itsPos);
   System.exit (0);
}  //======================
```

Working with an array of Strings

The `seesSlot` method checks that `itsSequence` has a slot at the current position, which requires that `itsPosition` not be greater than the maximum allowed. Since `itsSequence` is an array rather than a String, we must compare with the final `length` variable of the array rather than calling the `length()` method of a String:

```
public boolean seesSlot()              // in Vic
{  return itsPos < itsSequence.length;
}  //======================
```

The `seesCD` method should now be understandable. We still have the class variable `NONE = "0"` with which we can compare the value in component `itsPos`:

```
public boolean seesCD()                // in Vic
{  if ( ! seesSlot())
      fail ("can't see a CD where there is no slot, at ");
   return ! itsSequence [itsPos].equals (NONE);
}  //======================
```

The private `trace` method is a little more complex, because we have to build the String value for the sequence of CDs so we can print the current status. We can start with the first CD `itsSequence[1]` and adding each additional CD string to that, with blanks separating them. The method can be coded as follows:

```
private void trace (String action)     // in Vic
{  String seq = itsSequence [1];
   for (int k = 2;  k < itsSequence.length;  k++)
      seq += " " + itsSequence [k];
   System.out.println ("Vic# " + itsID + ": " + action
             + itsPos + "; sequence= " + seq);
}  //======================
```

Implementing the stack

The stack is also an array of String values, but unlike a sequence, its size is not fixed. That is, `itsStack` is a partially-filled array but `itsSequence` is completely filled. So we need to track the current number of items `theStackSize` in `theStack`. We also need to make `theStack` large enough for any foreseeable demands. The following class variables should suffice, having the stack be empty to start with:

```
private static String[] theStack = new String[1000];
private static int theStackSize = 0;
```

For the `takeCD` method, you first have to make sure that a CD is at `itsPos`. If so, you copy it into `theStack` after all the other CDs currently in `theStack`, thereby increasing the number of items in the stack. Finally, you put `NONE` where the CD was in `itsSequence`:

```
public void takeCD()                   // in Vic
{  if (seesCD())
   {  theStack [theStackSize] = itsSequence [itsPos];
      theStackSize++;
      itsSequence [itsPos] = NONE;
   }
   trace ("takeCD at slot ");
}  //======================
```

The `say` and `reset` methods have exactly the same coding as in Section 5.5, using the same `theTableau` class variable to hold the array of String values that describes the various sequences and `theNumVics` class variable to tell how many Vic objects have been created so far. That leaves only the Vic constructor as a (hard) exercise.

Exercise 7.48 Write Vic's `stackHasCD` method in the context described in this section.
Exercise 7.49 Write Vic's `putCD` method in the context described in this section.
Exercise 7.50* Write Vic's `moveOn` method in the context described in this section.
Exercise 7.51** Write the Vic constructor in the context described in this section, using CD names such as "a1", "b1", "c1" for CDs in the first sequence, "a2", "b2", "c2" for CDs in the second sequence, etc.

7.10 Command-Line Arguments

The `(String[ ] args)` part of the heading of the main method allows it to receive information from the `java` command that started the program. Each word on the command line after `java ProgramX` is assigned to an array component, in the order it appears on the line: `args[0]`, `args[1]`, etc.

For instance, suppose that the statement `Debug.setTrace(true)` is to produce tracing printouts in your program. You could start your main method with this line:

```
Debug.setTrace (args.length > 0 && args[0].equals ("trace"));
```

Now to run the program to produce tracing printouts, you enter `java ProgramX trace`. That will make `args` an array of length 1 with `args[0]` having the value `"trace"`. `trace` is the **command-line argument**. When you do not want tracing printouts, just use the usual `java ProgramX`. That will make `args` an array of length zero.

Using the command line to add numbers

If the command line has four words after the basic command, then `args.length` is 4, `args[0]` is the first word, `args[1]` is the second word, `args[2]` is the third word, and `args[3]` is the fourth word. Words are divided at blanks. Even numerals count as words. For instance, if you have the following Add class, you can enter something like `java Add 4.2 -2.16 13.7 5.1` and have it respond with the correct total of 20.839999999999996 (due to rounding off in base 2):

```
public class Add
{  public static void main (String[ ] args)
   {  double total = Double.parseDouble (args[0]);
      for (int k = 1;  k < args.length;  k++)
         total += Double.parseDouble (args[k]);
      System.out.println ("The total is " + total);
      System.exit (0);
   }  //======================
}
```

You could instead call this `main` method from a different class. For instance, executing the following two statements in another class prints "The total is 13.5":

```
String[] values = {"4.25", "7.5", "1.75"};
Add.main (values);
```

Exercise 7.52* Revise the CountBirthYears program in the earlier Listing 7.4 to let the user supply the file name in the command line. Only use `workers.txt` when the file name is not supplied.

Exercise 7.53* Revise the main method for the Add class to give the answer zero when the user does not put anything after `java Add` (currently the program crashes).

*7.11 Implementing Queue As A Subclass Of ArrayList (*Enrichment)*

The Sun standard library offers a quite useful class of objects named ArrayList (it supercedes the Vector class that was in Java Version 1.0). Each instance of ArrayList represents an ordered collection of Object values, called the **elements** of the ArrayList, and is in many respects similar to an array. The three basic methods available for an **ArrayList** (from the `java.util` package) are the following:

- `someArrayList.size()` tells how many elements are stored in this ArrayList. It is analogous to `item.length` for an array declared as `Object[] item`.
- `someArrayList.get(indexInt)` returns the Object value stored at `indexInt`. This is analogous to using `item[indexInt]` in an expression. ArrayLists are zero-based, i.e., the first value in the array is at index 0.
- `someArrayList.set(indexInt, someObject)` replaces the element at `indexInt` by `someObject`. So this is analogous to the assignment `item[indexInt] = someObject`.

An example of coding using an ArrayList object named `al` is the following. It replaces each element that equals a given `target` value by `null`:

```
for (int k = 0;  k < al.size();  k++)
   if (al.get (k).equals (target))
      al.set (k, null);
```

Changing the length of an ArrayList

If the preceding were coding involving an array, you would replace the three phrases involving `al` by `al.length`, `al[k].equals(target)`, and `al[k] = null`, respectively. So this ArrayList class would not be much use if it were not for the other methods it offers. Those methods have an ArrayList object grow and shrink in length, which of course is beyond the power of an ordinary array. For instance, `al.add(ob)` extends the list of elements to put `ob` at the end, and `al.remove(3)` shrinks the list of elements by deleting the one at index 3 (i.e., the fourth element in the list). Listing 7.13 (see next page) describes the most useful ArrayList methods.

Implementing a Queue with an ArrayList

The RepairShop software in Chapter Six uses a Queue class that includes three operations:

- `aQueue.enqueue (ob)` adds the object value `ob` to `aQueue`. `ob` can be any kind of object.
- `aQueue.dequeue()` returns the next available object on a first-in-first-out basis.
- `aQueue.isEmpty()` tells whether `aQueue` is empty.

The values stored in the Queue are of Object type. Since Object is a superclass of every class in Java, you can put any kind of object you like in a Queue.

A queue is a kind of object that you will see many uses for in later Computer Science courses. Its `dequeue` method removes the element that has been in the queue for the longest period of time. For this reason, a queue is known as a First-In-First-Out data structure (FIFO).

Listing 7.13 Key methods in the ArrayList class, stubbed form

```
public class ArrayList    // stubbed documentation
{
   /** Create an empty list of elements. */
   public ArrayList()                                    { }

   /** Return the number of elements in this list. */
   public int size()                                  { return 0;  }

   /** Return the element at this index.  Throw
    *  IndexOutOfBoundsException unless 0 <= index < size(). */
   public Object get (int index)                   { return null; }

   /** Replace the element at this index by ob.  Throw
    *  IndexOutOfBoundsException unless 0 <= index < size(). */
   public void set (int index, Object ob)                 { }

   /** Put ob as the last element in this list.  Return true. */
   public boolean add (Object ob)                  { return true; }

   /** Delete the element at this index and return it.
    *  Shrink the list by 1 element. Throw
    *  IndexOutOfBoundsException unless 0 <= index < size(). */
   public Object remove (int index)                { return null; }

   /** Insert ob at this index.  Expand the list by 1.  Throw
    *  IndexOutOfBoundsException unless 0 <= index <= size(). */
   public void add (int index, Object ob)                 { }
}
```

The standard specification for a Queue includes a `peekFront` method to allow people to see what they would get if they called the `dequeue` method, yet without modifying the Queue object. This method is not used by the RepairShop software, but it is standard to have it in the Queue class so that other applications that use Queues will have all they need.

Listing 7.14 (see next page) has a straightforward implementation of the Queue class as a subclass of ArrayList. Note that `dequeue` and `peekFront` throw an IndexOutOfBoundsException when the Queue is empty, since that is what the corresponding ArrayList methods do.

Exercise 7.54 Rewrite the methods in Listing 7.14 so that Queue does not extend the ArrayList class. Instead, it has an ArrayList instance variable. This is the use of composition rather than inheritance.

Exercise 7.55 Write an action method named `replaceNullsBy` for a subclass of ArrayList: The executor replaces each `null` value in it by a given Object parameter.

Exercise 7.56 Write a query method named `indexOf` for a subclass of ArrayList: The executor returns the index number of the earliest occurrence of an Object parameter, using the `equals` method to find it. It returns `-1` if the parameter is not found.

Listing 7.14 The Queue class of objects

```
public class Queue extends java.util.ArrayList
{
   public Queue()
   {  super();
   }  //====================

   /** Tell whether the queue has no more elements. */

   public boolean isEmpty()
   {  return size() == 0;
   }  //=====================

   /** Remove the value that has been in the queue the longest
    *  time.  Throw an Exception if the queue is empty. */

   public Object dequeue()
   {  return remove (0);
   }  //====================

   /** Return what dequeue would give, without modifying
    *  the queue.  Throw an Exception if the queue is empty. */

   public Object peekFront()
   {  return get (0);
   }  //====================

   /** Add the given value to the queue. */

   public void enqueue (Object ob)
   {  add (ob);
   }  //====================
}
```

*7.12 More On System, String, And StringBuffer (*Sun Library)*

This section describes features of the System and String standard library classes that can be quite useful, though they are not used elsewhere in this book. It also describes the StringBuffer standard library class.

System.out

`System.out` is a PrintStream object that is already open and prepared to receive output data. It is normally the terminal window. You may use

```
System.out.print (someString)
```

to print a String value without starting a new line immediately after it. For instance, `System.out.print("a")` could execute and `System.out.println("b")` could execute a while later, and the output would be a single line containing `"ab"`. The following statements print all the lowercase letters on one line of output:

```
for (int k = 'a';  k < 'z';  k++)
   System.out.print ((char) k);
System.out.println ('z');
```

The `print` and `println` methods are heavily overloaded to allow a boolean, char, int, double or other numeric value for the parameter, as well as any Object (for which the object's `toString` method is used to decide what appears). So if `sam` is a Worker object, then `System.out.println(sam)` actually prints `sam.toString()`.

Redirecting output to a disk file

You may redirect `System.out`'s output to a disk file. For instance,

```
java Whatever > results.txt
```

sends `System.out`'s output to the file named `results.txt` instead of to the terminal window. You can still send information to the terminal window using `System.err.println(whatever)`, and `System.err.print(whatever)`; `System.err` is the standard error channel for reporting problems.

The System.arraycopy method

The statement

```
System.arraycopy (source, k, target, n, 200);
```

copies 200 values from the array of Objects named `source` to the array of Objects named `target`. `source[k]` is copied into `target[n]`, then `source[k + 1]` is copied into `target[n + 1]`, etc., until the required number of values have been copied. It throws an IndexOutOfBoundsException if data would be accessed outside of an array's bounds, and it throws an **ArrayStoreException** if it is illegal to make the assignments.

Additional String methods

The String class has many more methods than those discussed in Chapter Six (`equals`, `length`, `substring`, `compareTo`, and `charAt`). If `s` and `t` are two String values, then the following methods can be useful:

- `s.concat(t)` is the same as `s + t`.
- `s.equalsIgnoreCase(t)` is the same as `equals` except that a lowercase letter is considered equal to the corresponding capital letter. Thus `"abc".equalsIgnoreCase("AbC")` is `true`.
- `s.compareToIgnoreCase(t)` is the same as `compareTo` except that a lowercase letter is considered equal to the corresponding capital letter. Thus `"BOB".compareToIgnoreCase("abe")` is a positive int.
- `s.indexOf(t)` returns the index where the first copy of `t` begins in `s`; it returns -1 if no substring of `s` is equal to `t`. Thus `"aababa".indexOf("ba")` is 2.
- `s.indexOf(t, n)` is the same as the above except that it only finds those substrings at index `n` or higher. Thus `"aababa".indexOf("ba",3)` is 4.
- `s.endsWith(t)` tells whether `s.equals(someString + t)`. Thus `"apple".endsWith("ple")` is `true`.
- `s.startsWith(t)` tells whether `s.equals(t + someString)`. Thus `"cathy".startsWith("cat")` is `true`.

If `s` is a String value, then:

- `s.trim()` is the result of removing all whitespace from either end.
- `s.toLowerCase()` is the result of replacing each capital letter by the corresponding lowercase letter. Thus `"A1bC".toLowerCase()` is "a1bc".

- `s.toUpperCase()` is the result of replacing each lowercase letter by the corresponding capital letter.
- `s.indexOf(someChar)` returns the first index at which the given character occurs in the string. If it is not in `s`, the method returns -1.
- `s.indexOf(someChar, someInt)` is the same as the above except that it returns -1 if no character at or after the given index equals the given char value.
- `s.replace(someChar, anotherChar)` returns the result of replacing every occurrence of the first parameter by the second parameter. Thus `"apple".replace('p', 'x')` returns "axxle".
- `s.toCharArray()` produces the corresponding array of characters with the same number of components as `s` has characters. If you store it in `char[] chArray` and make some modifications in it, then `new String(chArray)` converts it back to the corresponding String object.

The four versions of `indexOf` have four corresponding versions of `lastIndexOf`, which returns the last index at which the char or substring occurs (or -1 if none).

StringBuffer objects (from java.lang)

You can create a StringBuffer object from a given String value when you want to make substantial changes in the characters of the string. A String object is immutable, so substantial changes require continually creating new String objects. With a StringBuffer object, you make the changes you want and then put the information back into a String object. The essential StringBuffer methods are the following:

- `new StringBuffer(someString)` makes a modifiable copy of the given String value.
- `someStringBuffer.length()` returns the number of characters in it (like `itsSize`).
- `someStringBuffer.charAt(someInt)` returns the character at that index (zero-based as usual).
- `someStringBuffer.setCharAt(someInt, someChar)` puts the given character at the given index in place of whatever char value is currently there.
- `someStringBuffer.toString()` returns a copy of the information as a String object.

For instance, coding to reverse the order of the characters in a StringBuffer named `buf` could be as follows:

```
int len = buf.length();
for (int k = 0;  k < len / 2;  k++)
{  char saved = buf.charAt (k);
   buf.setCharAt (k, buf.charAt (len - 1 - k));
   buf.setCharAt (len - 1 - k, saved);
}
```

Additional methods that change the length of StringBuffers are as follows:

- `someStringBuffer.append(startInt, someString)` puts `someString` at the end of the executor. It also returns the result.
- `someStringBuffer.delete(startInt, endInt)` deletes characters at index `startInt` through `endInt-1` and also returns the result, for `0 <= startInt <= endInt <= someStringBuffer.length()`.
- `someStringBuffer.insert(startInt, someString)` puts `someString` starting at position `startInt` (moving other characters up), and also returns the result, for `0 <= startInt <= someStringBuffer.length()`.

- `someStringBuffer.replace(startInt, endInt, someString)` has the same effect as `someStringBuffer.delete(startInt, endInt).insert (startInt, someString)`.

String buffers are more efficient when you use string concatenation heavily. For instance, the `toString` method for WorkerLists in Listing 7.11 would be better written as follows:

```
public String toString()                    // in WorkerList
{  StringBuffer s = new StringBuffer ("");
   for (int k = 0;  k < itsSize;  k++)
      s.append (itsItem[k].toString()).append ("\n");
   return s.toString();
}  //======================
```

7.13 Review Of Chapter Seven

Listing 7.4, Listing 7.5, and Listing 7.8 illustrate almost all Java language features introduced in this chapter.

About the Java language:

- `Type[ ] t` declares `t` to be the name of an **array variable**, an object variable that can refer to an **array** of values of the specified Type.
- The only operations you can perform on an array variable are to put **brackets** `[ ]` after it or `.length` after it. The only operation you can perform on a non-array object variable is to put a dot after it, followed by a variable or method belonging to its class. If you do any of these things when the object or array variable is `null`, it throws a NullPointerException.
- `someObject.toString()` calls a method in the Object class. It produces the String analog of the Object. Any kind of object can be the executor.
- `t = new Type[38]` creates the array itself, so its 38 **components** can be filled in with values. Any non-negative int value can be used instead of 38 for the size. The array components are numbered from 0 up through 37 if `t`'s length is 38. Any use of `t[0]` or `t[1]` or in general `t[k]` refers to a Type variable. Any non-negative int value less than the length of the array can be used in place of `k` for the **index**.
- The declaration `new someType[n]` where `n` is negative throws a **NegativeArraySizeException**. The use of `someArray[k]` when `k` is negative or not less than the length of the array throws an **ArrayIndexOutOfBoundsException**.
- `t.length` is the number of components the array `t` has available.
- `Type[ ] t = {...}` with a number of values listed inside those braces creates the array and initializes it to have the values listed. This is an **initializer list**.
- `Type[ ][ ] t` declares `t` to be the name of a **two-dimensional array**. Then `t = new Type[n][m]` declares t to have `n * m` components doubly-indexed, where the first index can be `0..(n-1)` and the second can be `0..(m-1)`. `t.length` is `n` and `t[k].length` is `m`.

Other vocabulary to remember:

- **Stubbed documentation** for a class is a list of the method headings with comments describing their functions and not much else. It can be compilable if you add bodies such as `{return 0;}`.
- A program should be **robust**, which means that it should handle unexpected or unusual situations in a reasonable manner without crashing.

- A **test set** is a set of input values used for one run of a program, together with the expected outputs. A **Test Plan** is a large enough number of test sets that you can be confident that almost all of the bugs are out of the program when all tests go right.
- A **partially-filled array** has useable values only in part of the array, conventionally at `0...size-1` (though many programmers use a name different from `size`).
- For **modular programming**, you create software for one purpose as a number of moderately independent modules (methods and classes). Then when you want to use the software for a different purpose, you just pull out a few of the inappropriate modules and slot in others.
- The **postcondition** for an action method is a statement of what has changed as a result of calling the method, assuming the precondition for that method has been met. An **internal invariant** for a class of objects is a description of the state of all objects which the coding in the class maintains as true when each method is called and true when each method is exited. In other words, it is both precondition and postcondition. It describes the connection between the abstract concept for which the class of objects is a concrete realization in software.
- A **command-line argument** is a String value on the command line after `java ClassName`. These values are passed in to the `String[] args` parameter of the main method. `args.length` tells how many values are passed in. `args[0]` is the first String value, `args[1]` is the second String value, etc.

Answers to Selected Exercises

7.1
```
public static void averageDailyPay (Worker karl)
{    JOptionPane.showMessageDialog (null, "The worker's name is " + karl.getName()
                + ", and the average daily pay is " + (karl.getWeeksPay() / 5));
}
```

7.2 Replace those four lines by:
```
CompOp.printLastOne (new Worker (infile.readLine()),
            new Worker (infile.readLine()),new Worker (infile.readLine()));
```

7.3
```
public void printLastOne (Worker first, Worker second, Worker third)
{    Worker last =  first.getBirthYear() >= second.getBirthYear()  ?  first : second;
     if (last.getBirthYear() < third.getBirthYear())
          last = third;
     javax.swing.JOptionPane.showMessageDialog (null, "The youngest is " + last.toString());
}
```

7.4 Replace the last line of printLastOne by:
```
return last;
```
Replace the first part of the heading of printLastOne by:
```
public Comparable findLastOne
```
Replace the call of printLastOne in the main method by:
```
javax.swing.JOptionPane.showMessageDialog (null, "The alphabetically last is "
          + ((Worker) CompOp.findLastOne (first, second, third)).toString());
```

7.7 Replace the greater-than operator > by the less-than operator < .
Also change "first" to "last" everywhere, to have it make sense.

7.8 Insert after "else {": `double total = answerSoFar.getWeeksPay().`
Insert before the if-statement: `total += data.getWeeksPay();`
Insert before the last use of JOptionPane:
```
JOptionPane.showMessageDialog (null, "The total amount paid is " + total);
```

7.9 Insert after "else {":
```
int century = answerSoFar.getBirthYear() % 100;
boolean okay = true;
```
Insert before the if-statement:
```
if (data.getBirthYear() % 100 != century)
     okay = false;
```
Insert before the last use of JOptionPane:
```
if (okay)
     JOptionPane.showMessageDialog (null, "All were born in the same century"));
else
     JOptionPane.showMessageDialog (null, "NOT all were born in the same century"));
```

7.14 Replace the statement count[lastDigit]++; by:

```
if (lastDigit >= 0 && lastDigit < TIME_SPAN)
    count[lastDigit]++;
```

7.15 Rename TIME_SPAN as NUM_LETTERS throughout.
Change the initial/final value of NUM_LETTERS from 10 to 26.
Replace the while-statement and the for-statement after it by:

```
while (data.getName() != null)
{   int index = data.getName().charAt (0) - 'A';
    if (index >= 0 && index < NUM_LETTERS)
        count[index]++;
    data = new Worker (buffer.readLine());
}
for (int k = 0;  k < NUM_LETTERS;  k++)
    JOptionPane.showMessageDialog (null, (char) ('A' + k) + " had " + count[k] + " workers.");
```

7.16 Change the initial YEAR from 1960 to 1920 and the initial TIME_SPAN from 10 to 65.
Replace the statement count[lastDigit]++; by:

```
if (lastDigit >= 0 && lastDigit < TIME_SPAN)
    count[lastDigit]++;
```

If you do this program without arrays, then:
(a) The 3 lines that create the array and initialize it to zero are replaced by 65 initializations to zero.
(b) The count[lastDigit]++ line would have to be replaced by 130 lines such as:

```
else if (lastDigit == 24)
    count24++;
```

(c) And the 2 lines that print the answer would be replaced by 65 lines,
so the net added is 255 lines.

7.20 Say the number in component 0 is 11. Then reversing the direction of the for-loop would copy the 11 into component 1, then copy the 11 into component 2, then copy the 11 into component 3, then again into component 4. You would lose three needed values.

7.21 Replace the for statement and the return statement by the following:

```
for (int k = 0;  k < NUM_DAYS;  k++)
    sum += itsHoursPaid[k] <= 8  ? itsHoursPaid[k]  :  (itsHoursPaid[k] * 1.5 - 4);
return sum * itsHourlyRate;
```

7.23 Replace the statements beginning with "double totalPay" by the following:

```
if (size > 0)  // crash-guard
{   Worker lastWorker = item[size - 1];
    for (int k = 0;  k < size;  k++)
        if (item[k].compareTo (lastWorker) < 0)
            JOptionPane.showMessageDialog (null, item[k].toString());
}
```

7.25

```
public String alphabeticallyFirst()
{   if (itsSize == 0)
        return null;
    Worker early = itsItem[0];
    for (int k = 1;  k < itsSize;  k++)
        if (itsItem[k].compareTo (early) < 0)
            early = itsItem[k];
    return early.getName();
}
```

7.26

```
public boolean everyonePaidLessThan (double cutoff)
{   for (int k = 0;  k < itsSize;  k++)
        if (itsItem[k].getWeeksPay() >= cutoff)
            return false;
    return true;
}
```

7.27

```
public boolean inAscendingOrder()
{   for (int k = 0;  k < itsSize - 1;  k++)  // note itsSize - 1
        if (itsItem[k].compareTo (itsItem[k + 1]) > 0)
            return false;
    return true;
}
```

7.28

```
public void remove (Worker given)
{   int k = 0;
    while (k < itsSize && ! itsItem[k].equals (given))
        k++;
    if (k < itsSize)    // we have found a match
    {   itsItem[k] = itsItem[itsSize - 1];    // replace it by the top one
        itsSize--;
    }
}
```

7.29
```
public int countSame (WorkerList given)
{   int upperLimit = this.itsSize > given.itsSize  ?  given.itsSize  :  this.itsSize;
    int count = 0;
    for (int k = 0;  k < upperLimit;  k++)
        if (this.itsItem[k].equals (given.itsItem[k]))    // we have found a match
            count++;
    return count;
}
```
7.35
```
public void insert (Worker given, int n)
{   if (n >= 0 && n <= itsSize && itsSize < itsItem.length)
    {   for (int k = itsSize;  k > n;  k--)
            itsItem[k] = itsItem[k - 1];  // similar to setHoursWorked
        itsItem[n] = given;
        itsSize++;
    }
}
```
7.36
```
public boolean put (Worker par)
{   if (itsSize == itsItem.length)
        return false;
    for (int k = 0;  k < itsSize;  k++)
        if (itsItem[k].equals (par))
            return false;
    itsItem[itsSize] = par;
    itsSize++;
}
```
7.43
```
if (x > 1 && y < 6 && board[x - 2][y + 2] == EMPTY)
{   board[x - 2][y + 2] = board[x][y];
    board[x][y] = EMPTY;
    board[x - 1][y + 1] = EMPTY;
}
```
7.44
```
for (int y = 0; y < 8; y += 2)
{   board[5][y] = BLACK;
    board[6][y + 1] = BLACK;
    board[7][y] = BLACK;
}
```
7.45
```
public static int numEmptySquares (int[][] board)
{   int count = 0;
    for (int x = 0;  x < 8;  x++)
        for (int y = 0;  y < 8;  y++)
            if (board[x][y] == EMPTY)
                count++;
    return count;
}
```
7.48
```
public boolean stackHasCD()
{   return theStackSize > 0;
}
```
7.49
```
public void putCD()
{   if (! seesCD() && theStackSize > 0)  // or make the second operand stackHasCD()
    {   itsSequence [itsPos] = theStack [theStackSize - 1];
        theStackSize--;
    }
    trace ("putCD at slot ");
}
```
7.54
```
public class Queue extends Object
{   private ArrayList itsList = new ArrayList();
    //  all methods are the same as in Listing 7.13 except put "itsList." in front of each call
    //  of a method from the ArrayList class, e.g., return itsList.remove (0).
}
```
7.55
```
public void replaceNullsBy (Object ob)
{   for (int k = 0;  k < size();  k++)
        if (get (k) == null)
            set (k, ob);
}
```
7.56
```
public int indexOf (Object ob)
{   for (int k = 0;  k < size();  k++)
        if (get (k).equals (ob))
            return k;
    return -1;
}
```

8 Elementary Graphics

Overview

This chapter introduces graphics programming in the context of software for a flag manufacturer. You will see how to create applets using graphics commands, both static drawings and animated drawings. These applets can be part of a web page or part of a stand-alone application using a JFrame.

This chapter contains no new language features. We discuss only using an applet in a web page. The first six sections can be covered after finishing Section 4.7 (arrays are used beginning in Section 8.7). If you wish to use graphics in a stand-alone application, all you need to know about JFrames is in Section 9.1. Using a JFrame is more complicated, which is why we only work with web pages in this chapter.

- Sections 8.1-8.2 explain basic graphics techniques for JApplets using the Graphics2D, Shape, and Color classes.
- Sections 8.3-8.5 develop the logic for drawing a complete animated American flag.
- Sections 8.6-8.7 review basic principles of software development.
- Sections 8.8-8.9 discuss a large graphics program and implement the Turtle class.

8.1 The JApplet, Color, And Graphics2D Classes

You have been hired to create some graphical software by Flag-Maker Inc., a company that makes flags. Specifically, they want all of their web pages and application programs to have a small red-white-and-blue American flag dancing around on the screen the whole time people are using them.

You probably think this is not the smartest idea in the world, but they are offering you $20,000 to design the software, so you keep your mouth shut and do it (this book is about real-life programming techniques, and doing what the client wants is one of the most profitable techniques, as long as it is ethical). You have probably heard of W. A. Mozart, a genius at developing software for the piano and other instruments. He had a financially rewarding job with a large corporation, but the kind of software he had to write was dictated by his district manager, the Archbishop of Salzburg. He quit after ten years so he could write what he wanted; but then he nearly starved.

Imports and the Color class

Java has an enormous number of classes in its standard library for doing graphics; see `http://java.sun.com/docs` for details. One graphics package is `java.awt`; the Color class and Graphics class are in this package. Another package is `javax.swing`, which contains the JFrame and JApplet classes. JFrames are used for applications and JApplets are used for graphical parts of a web page.

The **Color** class contains several class variables named `red`, `blue`, `green`, etc. The complete way to mention e.g. the color `red` from the Color class in your program is `java.awt.Color.red`. It is of course much easier to simply use `Color.red`, which you can do if you have the following line at the top of your compilable file:

```
import java.awt.Color;
```

The thirteen standard color constants available in the Color class are `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `yellow`, `red`, and `white`. Cyan is a greenish-blue. You can also create your own color by supplying red/green/blue (RGB) values in the range from 0 to 255, inclusive:

```
new Color(16, 255, 0) // green with a reddish tinge
```

The JApplet class

You begin developing an applet by defining a subclass of the JApplet class from the `javax.swing` package. Objects of the **JApplet** class represent rectangular panels within a web page or application frame. JApplet is a subclass of the Panel class which in turn is a subclass of the Component class.

The simplest applets contain nothing but a method with the heading `public void paint (Graphics g)`. Listing 8.1 shows an applet that writes two sentences, "stars goes here" in blue and "stripes go here" below it in red. It then draws a line between the two sentences. We call this applet Dancer because a later version will produce the dancing flag. The statements in the `paint` method are explained in the next section. It imports from `java.awt.*` to get Graphics, Graphics2D, and Color.

Listing 8.1 The Dancer applet, version 1

```
import javax.swing.JApplet;
import java.awt.*;
import java.awt.geom.Line2D;

public class Dancer extends JApplet
{
   /** Draw things on the applet's drawing area. */

   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.setColor (Color.blue);
      page.drawString ("stars goes here", 10, 45);
      page.setColor (Color.red);
      page.drawString ("stripes go here", 10, 90);
      page.draw (new Line2D.Double(10, 60, 30, 70));
   }  //=====================
}
```

HTML

To test an applet, you first need to compile it: `javac Dancer.java` for the Dancer applet given in Listing 8.1 produces the compiled form `Dancer.class`. Next, you need to have a web page that refers to it. The following is a very simple web page in HTML that you could use to test this Dancer applet:

```
<HTML>
   <BODY> Your name goes here
      <APPLET CODE="Dancer.class" WIDTH=240 HEIGHT=120>
      </APPLET>
   </BODY>
</HTML>
```

Each HTML tag is written inside angle braces < > as illustrated above. HTML tags usually come in pairs, one to say where a certain kind of section starts and one to say where it ends. Usually the ending tag has the same first word as the beginning tag but with a division sign in front of it. This particular HTML file has a body section but no heading section. The body contains some words that will appear on the web page (put your own name in place of them) and an APPLET pair of tags. The APPLET tag tells the browser where to find the executable file that contains the applet, and it also tells the dimensions of the applet's rectangular area.

Appletviewer and web browsers

If you put that HTML material in a file named `Dancer.html` in your `C:\cs1` folder along with the applet Dancer.class, then enter `file:///C:/cs1/Dancer.html` in the URL locator area of your web browser, you should see the blue and red sentences (if your files are in a different folder from `C:\cs1`, make the obvious substitution).

For testing an applet, you may use the **appletviewer** program that comes with your Java installation: At the command line in the terminal window, in response to the `C:\cs1` prompt, enter `appletviewer Dancer.html` to see just the applet code of the HTML file. Clicking on the word "Applet" at the top brings down the applet menu you can use to reload the applet (to run it again), print the applet, or quit the applet.

A web browser that loads a web page executes the logic of the `paint` method in any applets there. The applet does not need a main method because the web browser has one. This `paint` method makes the drawing. Each time the user moves the web page or returns to it after having linked elsewhere, the web browser calls the `repaint` method for the JApplet (inherited from the Component class). The `repaint` method spreads the applet's background color on the applet's drawing area to erase whatever is there, then it calls the applet's `paint` method.

The left side of Figure 8.1 shows the result of displaying the applet in Listing 8.1. The right side of the figure is the same except that lines have been added ten pixels apart, so you can see the exact placement of the words on the screen. Only the starting point of the words is determined by the second and third values in the `drawString` method call; the size of the font determines how tall and how wide the words are.

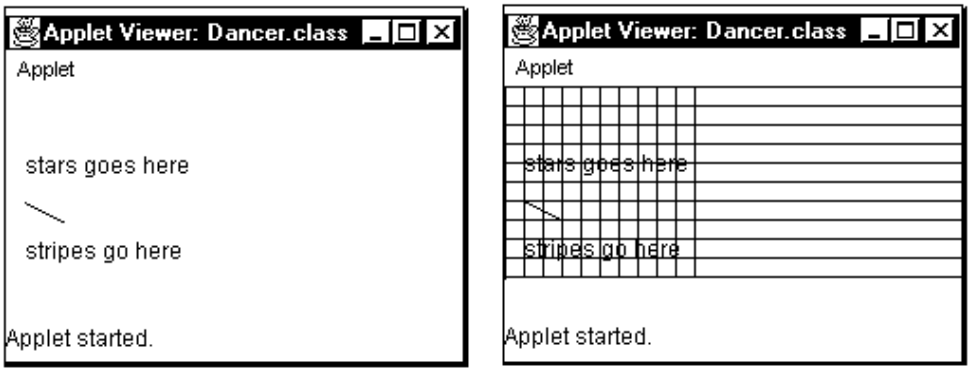


Figure 8.1 Result of the Dancer applet

The Graphics2D class

Graphics2D is a class in the standard `java.awt` package; its objects represent drawing areas. The **drawing area** is measured in units of **pixels**. You specify points on the drawing area with two int values `<x, y>`. The first one tells how many pixels in from the left edge the drawing is to be made. The second one tells how many pixels down from the top edge the drawing is to be made.

Since the HTML specifies that the drawing area for the Dancer class is 240 pixels wide and 120 pixels tall, the center point is at <120,60>, the bottom-left corner is at <0,120>, and the top-right corner is at <240,0>. Anything you draw outside the boundaries of a drawing area does not appear (nor does it crash the program).

Graphics2D is a subclass of the Graphics class, which is also in `java.awt`. Graphics was used in earlier versions of Java. Graphics2D gives more control over geometric objects than does Graphics. To maintain compatibility with older browsers, the `paint` method continues to have its parameter be from Graphics; we then cast the parameter to the Graphics2D class as the first statement inside the `paint` method. As you see in Listing 8.1, we declare a local Graphics2D variable `page` and assign `(Graphics2D)g` to it, where `g` is the formal parameter. We then use `page` for everything.

Graphics2D methods

Each Graphics2D drawing area has a current **drawing Color** in which it draws all characters, lines, etc. As the Dancer applet shows, a call of an applet's `paint` method passes in a value for the Graphics object as the parameter. You can then cast it to a Graphics2D object and send the following messages to it:

- `setColor(Color)` makes the drawing Color the specified value. All drawing will be done in this color thereafter, until the drawing Color is changed. Initially, when the applet begins execution, the drawing Color is `Color.black`.
- `getColor()` returns the current drawing Color.
- `drawString(String, int xCorner, int yCorner)` draws the given string of characters with the bottom-left corner of the first character at the pixel position `<xCorner, yCorner>`.
- `draw(Shape)` draws the Shape object. A Line2D object is a kind of Shape. Its constructor specifies the starting `<x,y>` point in the first two parameters and the ending `<x,y>` point in the last two parameters.
- `fill(Shape)` draws the Shape object plus fills in its inside portion. This does not of course apply to a Line2D object. **Shape** is an interface defined in the `java.awt` package.

You can see that the first sentence that Dancer draws is at <10,45>, a point 10 pixels to the right and 45 pixels below the top-left corner of the drawing area. It is drawn in blue because of the `setColor(Color.blue)` command. The second sentence is drawn directly below it in red. A line is drawn from the point <10,60> to the point <30,70>, which is 20 pixels to the right and 10 pixels down from the starting point <10,60>.

Exercise 8.1 Write a `paint` method for an applet that puts your name in the drawing area in three different colors, all on the same line.
Exercise 8.2 Write a `paint` method for an applet that draws a green rectangle, three times as tall as it is wide. Use `draw(new Line2D...)` for all lines.
Exercise 8.3 Write a `paint` method for the Dancer class that draws a 40x40 square divided into four 20x20 squares. Put the bottom-right corner of the square at the point <50, 70>. Use `draw(new Line2D...)` for all lines.

8.2 Five Shape Classes: Rectangle, Line2D, Rectangle2D, Ellipse2D, and RoundRectangle2D

You need to know how to create various shapes to use in the `draw` and `fill` messages. One basic Shape is a Rectangle, from the `java.awt` package. You can use the following for Rectangle objects:

- `new Rectangle(int x, int y, int w, int h)` constructs a new Rectangle object whose top-left corner is at <x, y>, with width `w` and height `h` measured in pixels. If either `w` or `h` is negative, the Rectangle will not show in the drawing area.
- `grow(int dw, int dh)` adds `dh` pixels above and below the rectangle, and it adds `dw` pixels to the left and right sides of the rectangle. So its width changes by `2*dw`, its height by `2*dh`, and its top-left corner changes by <-dw, -dh>.
- `translate(int dx, int dy)` moves the rectangle `dx` pixels to the right and `dy` pixels down from its current position. The width and height remain unchanged.
- `setLocation(int x, int y)` move the top-left corner of the rectangle to the specified location. The width and height remain unchanged.
- `setSize(int w, int h)` changes the width and height of the rectangle to the specified values. The top-left corner remains unchanged.

Figure 8.2 shows the result of executing the Boxes applet in Listing 8.2. Study carefully the correspondence between the method and the figure.

Listing 8.2 An applet to demonstrate Graphics2D and Rectangle methods

```
import javax.swing.JApplet;
import java.awt.*;

public class Boxes extends JApplet
{
   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      Rectangle box = new Rectangle (30, 60, 50, 40);
      page.draw (box);           // the inner box in the figure
      box.grow (5, 5);
      page.draw (box);           // the outer box in the figure
      box.translate (70, 0);
      page.fill (box);           // the filled box on the right
   }  //======================
}
```

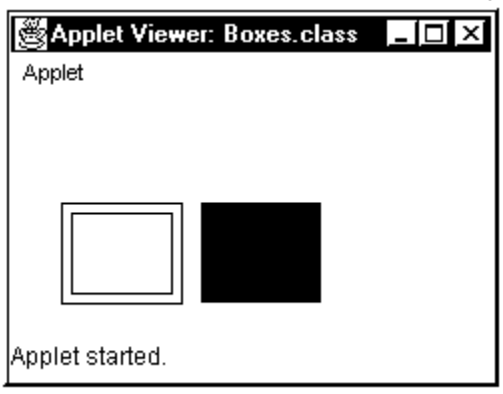


Figure 8.2 Illustration of Graphics2D and Rectangle methods

Line2D, Rectangle2D, Ellipse2D, and RoundRectangle2D

It is often useful to create geometric figures using decimal numbers for the parameters rather than int values. Then you can multiply or divide them by some fixed number without worrying about rounding them off, because the `draw` and `fill` methods use the parameter values rounded to the nearest int.

- `new Line2D.Double(double x, double y, double x2, double y2)` is a line object with one end-point at `<x,y>` and the other at `<x2,y2>`.
- `new Rectangle2D.Double(double x, double y, double w, double h)` is a rectangular object with parameters whose meaning is the same as for Rectangle.
- `new Ellipse2D.Double(double x, double y, double w, double h)` is an elliptical object whose bounding box has upper-left corner `<x,y>` and the given width `w` and height `h`. The **bounding box** is the smallest rectangle with vertical and horizontal sides that contains the ellipse. The ellipse is a circle if `w` and `h` are equal.
- `new RoundRectangle2D.Double(double x, double y, double w, double h, double arcWidth, double arcHeight)` is a rectangular object whose first four parameters have the same meaning as for Rectangle. The last two parameters specify the width and height of the arc on each corner, so you get a rectangular shape with rounded corners.

These four classes are in the `java.awt.geom` package. The reference to such a class has a dot in it. This is because e.g. the Ellipse2D class contains a class named Double nested inside it. If say you want to draw an ellipse that fits exactly within the inner box on the left of Figure 8.2, use this statement:

```
page.draw (new Ellipse2D.Double (30, 60, 50, 40));
```

Caution Check your drawing commands to be sure you do not draw outside the drawing area (e.g., all `x` values are 0 to 240 and all `y` values are 0 to 120). The program will not crash, but nothing will show up. An applet inherits two instance methods `getWidth()` and `getHeight()` from the Component class that you can use to find out its current dimensions.

Exercise 8.4 Write a `paint` method that draws two rectangles that form a Red Cross symbol. Put the word "give" in the left arm of the cross and the word "blood" in the right arm.
Exercise 8.5 Write a `paint` method for an applet that draws three rectangles, each inside the next, all with the same top-left corner. Use the `setSize` method twice appropriately.
Exercise 8.6 Revise the preceding exercise to fill the three rectangles with three different colors: `magenta`, `gray`, and `yellow` (Hint: Fill the outer square first).
Exercise 8.7 Write a `paint` method to draw three filled circles the same size all in a horizontal line, all of radius 10, with the middle one just barely touching each of the two side ones.
Exercise 8.8* Draw the Peace symbol.
Exercise 8.9* (Archery) Start a JApplet class named Archery (to be developed further in later exercises). Have its `paint` method draw a bullseye, a set of five concentric circles filled in with different colors. Write the Archery HTML, then compile and execute your applet. Hint: Fill a larger circle before filling a circle inside of it.
Exercise 8.10** Draw three filled circles with each barely touching each of the other two.

8.3 Analysis And Design Example: The Flag Software

Your client wants a dancing flag. Since a flag is a separate concept, you should have a separate class for Flag objects. Then the class can be reused in other software. The Dancer's `paint` method should pass to a Flag object the information it needs to do its job, which is to draw a flag at a particular point on a particular Graphics drawing area.

For the animation, you will repeatedly create Flags at different points on the screen. Each time the browser refreshes a web page containing an applet, or the user moves to a different part of the web page not containing the applet and then back again, the browser executes a method in the applet with the heading `public void start()`. The animation commands will go in this `start` method.

The body of Dancer's `paint` method should have a single command such as `new Flag (page, 10, 20)` to draw the flag 10 pixels in and 20 pixels down on the drawing area. That gives the revision in Listing 8.3. The numbers 10 and 20 are stored in private variables outside of all methods of the class. That way any method within the class can use the variables, although no method outside of the class can do so. In a later version of Dancer, to make the flag dance, the `start` method will change those two values that the `paint` method uses.

Listing 8.3 The Dancer applet, version 2

```
import javax.swing.JApplet;
import java.awt.*;

public class Dancer extends JApplet
{
   private int itsLeftEdge = 10;
   private int itsTopEdge = 20;


   public void start()
   {  // leave empty for now; animation comes later
      // this method will change itsLeftEdge and itsTopEdge
   }  //======================

   public void paint (Graphics g)
   {  new Flag ((Graphics2D) g, itsLeftEdge, itsTopEdge);
   }  //======================
}
```

Analysis (clarifying what to do) You cannot draw a flag unless you know how it looks. So you do some research on the internet at `www.askjeeves.com` to find a site that gives the correct proportions. You might think you could just estimate it from a picture, but a basic principle of programming is, "Don't make stuff up unless you have to." You will come off looking better if you have the right information.

You find that the flag should be 91 pixels tall and 173 pixels wide, assuming each stripe is 7 pixels tall. The 13 stripes alternate red and white starting with red. The blue field of stars should be 49 pixels tall (corresponding to 7 stripes) and 69 pixels wide. The stars should be in 9 rows staggered 4.9 pixels apart vertically. In each row, each star should be 11.4 pixels to the right of the one before. The rows of stars have 6 and 5 stars alternating starting with the row of 6. That makes 50 stars altogether.

Right away you realize that fractions cannot be used here; only a whole number of pixels can be used in most commands. So you decide to make stars 12 pixels apart with 5 pixels between rows. That leaves 2 pixels extra space above and below the 50 stars. You also need to expand the size of the applet from 240x120 to at least 320x160, to allow room for the flag to move around on the drawing area.

Test Plan (seeing if you did it) You will run the program, check the placement in the upper-left corner of the window, and count the stripes and the stars.

Design (deciding how to do it) First you develop the overall plan for drawing the flag, in deference to a basic principle of programming, "If you don't know where you are going, you are not likely to get there." The accompanying design block is a good start.

STRUCTURED NATURAL LANGUAGE DESIGN for the main Flag logic
1. Draw the 13 red and white stripes.
2. Draw the blue field.
3. Draw the 50 white stars.

The second step of this logic can be done with a simple `fill` command applied to a rectangle; the other two steps are complex enough to require their own methods. The initial design of the Flag class is implemented in Listing 8.4 (see next page), with the `drawStars` and `drawStripes` methods left for later development. The UML class diagram is in Figure 8.3.

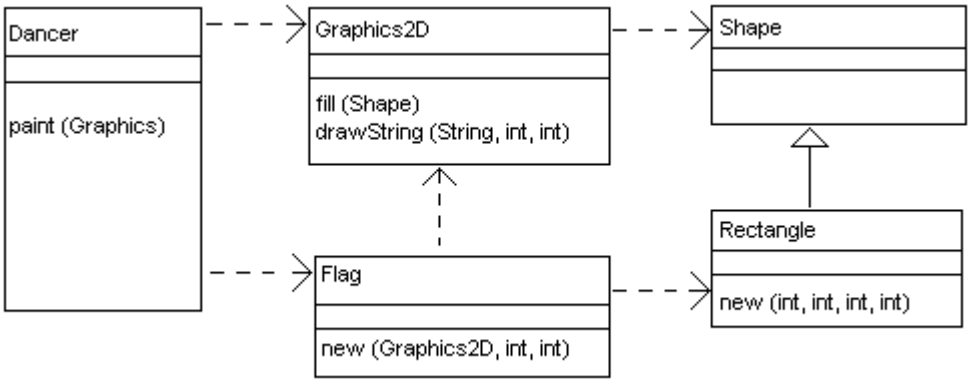


Figure 8.3 UML class diagram for Version 2

The Flag class defines six variables (names all in capital letters: `FLAG_WIDE`, `UNION_TALL`, etc.) that give the dimensions of the flag. These are defined as `final`, which just means that they can never be changed. That makes it safe to have them be public. In the further development of the logic, these constants are used in place of the actual numbers.

Exercise 8.11 Explain why it is not a good plan to do the three steps of the Flag constructor design in the opposite order.
Exercise 8.12 What changes would you make in Listing 8.4 to make the flag twice as big?
Exercise 8.13* Find a picture of the Indiana state flag on the internet. Lay out a structured natural language design for drawing it. Or use some other picture of some complexity.
Exercise 8.14* (Archery) Reorganize your Archery applet to have the `paint` method call a `drawArrow` method to draw an arrow on the left side of the drawing area and a `drawBullseye` method to draw a bullseye on the right side of the drawing area.

Listing 8.4 The Flag class, version 1

```
import java.awt.*;
import java.awt.geom.Line2D;

public class Flag extends Object
{
   public static final int FLAG_WIDE  = 173;
   public static final int FLAG_TALL  = 91;
   public static final int UNION_WIDE = 69;
   public static final int UNION_TALL = 49;
   public static final int PIP = 2;
   public static final int STAR = 5;
   ////////////////////////////////


/** Draw the flag with top-left corner at <x,y>. */

   public Flag (Graphics2D page, int x, int y)
   {  drawStripes (page, x, y);
      page.fill (new Rectangle (x, y, UNION_WIDE, UNION_TALL));
      drawAllStars (page, x, y);
   }  //=====================

   /** Draw 13 stripes in a 173x91 area, top-left at <x,y>. */

   private void drawStripes (Graphics2D page, int x, int y)
   {  page.drawString ("Here be stripes", x + 50, y + 70);
      // to be developed later
   }  //=====================

   /** Draw 50 stars in a 69x49 area, top-left at <x,y>.*/

   private void drawAllStars (Graphics2D page, int x, int y)
   {  page.drawString ("Here be stars", x, y);
      // to be developed later
   }  //=====================
}
```

8.4 Iterative Development Of The Flag Software

Version 1 of Dancer is compilable and runnable. It produces a result that is partial progress towards the desired end result. Now that we have Flag version 1 and Dancer version 2, we can run the applet again. This result is even closer to the desired end result.

Figure 8.4 shows what Dancer v.2 produces. This is the result of version 2 of the iterative development. The result of version 1 was in the earlier Figure 8.1, obtained from Dancer v.1 and no Flag class at all.

In general, once we have a full analysis of a problem for which the software is quite complex, we divide the development of the software into a succession of several versions. Each version adds more functionality to the software, that is, either it does more of what the final version should do or it does better what the previous version already does.

In developing each of these versions, we design the next modifications in the existing software, then implement them in Java, then test them. This is one full cycle in **iterative development** of the software.

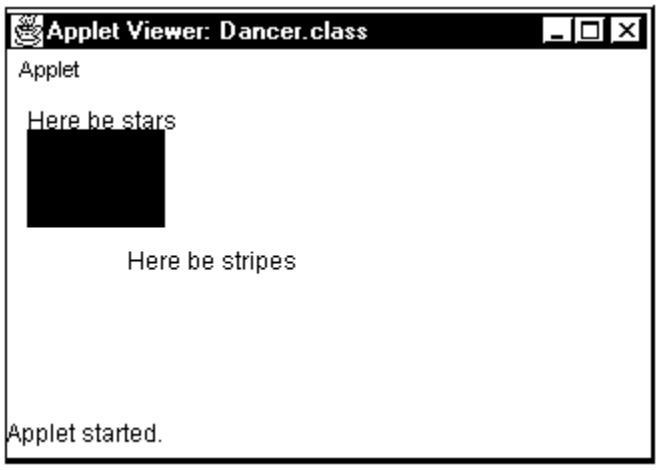


Figure 8.4 After iterative version 2 of the Flag software

In larger software projects, it is typical to have an overall written plan of iterative development where each cycle is estimated to take from two weeks to two months (depending on the type of software). We decide what we want to have at the end of each cycle (the "deliverables").

Next we process one version at a time: First we design what we want to have when we finish the improvement. Then we develop that software in smaller steps of design/implement/test (sort of a sub-iterative development) as illustrated for this Flag software. If the initial analysis was not sufficiently detailed, additional analysis is required for the version before its design stage.

Experience has shown that iterative development of software is effective and efficient. Do not leap into the implementation stage in your programming -- follow this plan. Remember, even Beethoven took lessons in music from more experienced people (although Joe Haydn was so irritated by his unorthodox ideas that he made Beethoven drop the class; but then Beethoven was a genius).

Ease of finding errors

In Figure 8.4, the filled rectangle is black. It was supposed to be blue. This indicates a bug in the software so far. The command to set the drawing Color to `Color.blue` was left out.

This sort of thing happens all the time, in accordance with the basic programming principle, "You're only human, you're supposed to make mistakes." It is easier to find and correct errors such as this when only an incremental change has been made to the software. This is a big advantage of iterative development.

Version 3 of iterative development

The next iterative step is to complete the entire flag, but without animation. The `drawStripes` method is simpler than the `drawStars` method, so we start with the `drawStripes` method. This is in accordance with the basic programming principle, "Always put the hard stuff off until later."

Development of the drawStripes method

You need to draw 13 stripes each 7 pixels tall. They alternate red with white, starting with the red at the top. So each stripe is done by setting the drawing Color to `Color.red` or `Color.white` and then drawing a rectangle. You need to have a field of stars in the top-left corner of the flag, but it is simplest to draw all of the stripes all the way across, since the later drawing of a rectangle puts the blue union over the stripes.

STRUCTURED NATURAL LANGUAGE DESIGN for drawStripes
1. Create a stripe that is 173 pixels wide and 7 pixels tall.
2. Repeat the following 13 times....
    2a. Set the color to red or white, depending on which stripe it is.
    2b. Draw the stripe.
    2c. Move the stripe 7 pixels further down the page.

The top-left corner of the flag is to be at `<x,y>`. These `x` and `y` values are parameters of the `drawStripes` method. So the initial rectangle should have `x` for its first parameter, `y` for its second, 173 (the width) for its third, and 7 (the height) for its fourth.

The problem is to get the color to switch back and forth between red and white. One way is to have an integer variable `k` that counts how many stripes you have made. If `k` is an even number, i.e., `k % 2 == 0`, make the next stripe red, otherwise make it white. The result is in Listing 8.5. The `page.setColor` statement uses either `Color.red` or `Color.white` depending on whether `count` is even.

Listing 8.5 The drawStripes method in the Flag class

```
/** Draw 13 stripes in 173x91 area, top-left at <x,y>. */

private void drawStripes (Graphics2D page, int x, int y)
{  final int TALL = FLAG_TALL / 13;
   Rectangle stripe = new Rectangle (x, y, FLAG_WIDE, TALL);
   for (int k = 0; k < 13; k++)
   {  if (k % 2 == 0)
         page.setColor (Color.red);
      else
         page.setColor (Color.white);
      page.fill (stripe);
      stripe.translate (0, TALL);
   }
}  //====================
```

If you wanted to draw a flag with three colors of stripes that alternate, you could have an int variable that you test to see if it is a multiple of 3 (so draw a red stripe), 1 more than a multiple of 3 (so draw a white stripe), or 2 more than a multiple of 3 (so draw a blue stripe) as follows:

```
if (k % 3 == 0)
   page.setColor (Color.red);
else if (k % 3 == 1)
   page.setColor (Color.white);
else
   page.setColor (Color.blue);
```

Development of the drawStars method

The Flag specifications require you to draw 9 rows of 5 or 6 white stars within a 69x49 field of blue. Each star is to be within a bounding 5x5 square of pixels (but you do not draw the 5x5 square itself). The stars should be 12 pixels apart as you go across. So a row of 6 stars takes up 65 pixels, `(5 * 12 + 5)`, leaving 2 pixels on each side.

Each row starts 5 pixels below the one above it, so 9 rows require 45 pixels. Since you have 49 pixels of height, that again leaves 2 pixels on the top edge and 2 on the bottom edge. Therefore, for a blue union whose corner is at `<x,y>`, the top-left start should be at `<x+2,y+2>`. Once you make a drawing to clarify the positioning of the stars, you have enough information to create the overall plan to draw the 50 stars, as shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for drawStars
1. Set the drawing Color to white.
2. Repeat the following 9 times...
   - 3a. Draw a row of 6 or 5 stars indented 2 or 8 pixels, alternating.
   - 3b. Shift downwards by 5 pixels for the next row.

The actual star will just be five lines. So you make a 5x5 grid of squares and see just what lines have to be drawn to get a star. This requires some work, but eventually you decide on a nice-looking set of five lines that will form a star. That is enough to warrant `drawOneStar` as a separate method. Except for `drawStripes`, the completed Flag class is in Listing 8.6 (see next page), and the flag itself is in Figure 8.5. Reminder: `x += y` is preferable to `x = x + y`, and `x -= y` is preferable to `x = x - y`.

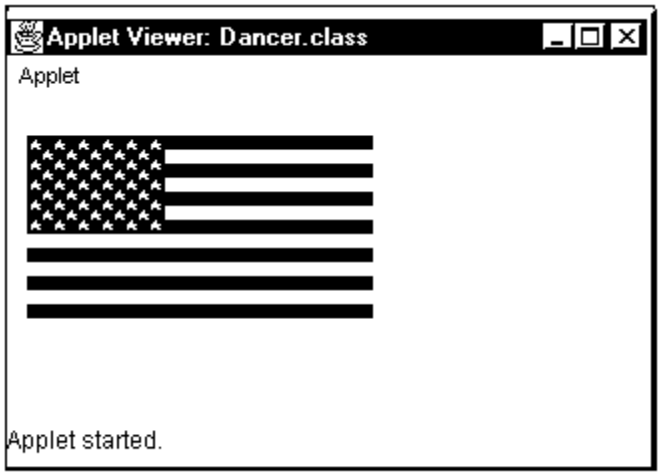


Figure 8.5 After Version 3 of iterative development of the Flag software

Listing 8.6 The Flag class, version 2 (final)

```
import java.awt.*;
import java.awt.geom.Line2D;

public class Flag extends Object
{
   public static final int FLAG_WIDE  = 173;
   public static final int FLAG_TALL  = 91;
   public static final int UNION_WIDE = 69;
   public static final int UNION_TALL = 49;
   public static final int PIP = 2;
   public static final int STAR = 5;
   ////////////////////////////////////


   public Flag (Graphics2D page, int x, int y)
   {  drawStripes (page, x, y);
      page.setColor (Color.blue);
      page.fill (new Rectangle (x, y, UNION_WIDE, UNION_TALL));
      drawAllStars (page, x, y);
   }  //====================

   private void drawAllStars (Graphics2D page, int x, int y)
   {  page.setColor (Color.white);
      for (int row = 0;  row < 9;  row++)
      {  int start = (row % 2 == 0)  ?  x + PIP : x + PIP + 6;
         drawRowOfStars (page, x + UNION_WIDE - STAR, start,
                         y + PIP + row * STAR);
      }
   }  //====================

   private void drawRowOfStars (Graphics2D page, int end,
                                int x, int y)
   {  for (;  x < end;  x += 12)
         drawOneStar (page, x, y);
   }  //====================

   private void drawOneStar (Graphics2D page, int x, int y)
   {  int p2 = PIP * 2;
      page.draw (new Line2D.Double (x + PIP, y, x, y + p2));
      page.draw (new Line2D.Double (x, y + p2, x + p2, y + PIP));
      page.draw (new Line2D.Double (x + p2, y + PIP,
                                    x, y + PIP));
      page.draw (new Line2D.Double (x, y + PIP, x + p2, y + p2));
      page.draw (new Line2D.Double (x + p2, y + p2, x + PIP, y));
   }  //====================
}
```

Exercise 8.15 How would you modify the `drawStripes` method to have 20 stripes each 5 pixels tall, alternating between green and orange?
Exercise 8.16 Write a `drawStripes` method that has 20 stripes cycling among four different colors.
Exercise 8.17 Rewrite the `drawStripes` method to use a boolean variable named `isRed` instead of checking whether the counter is even.
Exercise 8.18 The star-drawing methods use a number that depends on the size of the flag, rather than a constant. If the flag were doubled in size, that number would not change and the flag would not look right. Define a new constant and rewrite those methods accordingly.
Exercise 8.19 Rewrite the `drawAllStars` method without the conditional operator.
Exercise 8.20 Revise the star-drawing methods to draw 6 rows of stars, alternating between 4 and 5 stars in each row, starting with 4. Use the same spacing; do not try to fill up the entire width of the union.
Exercise 8.21* Rewrite the `drawStripes` method to have the body of the for-statement (a) set the color to white, (b) draw that stripe, (c) set the color to red, (d) draw that stripe. You need to draw the first red stripe before the for-statement. This executes faster than Listing 8.5 but is harder to develop.
Exercise 8.22* Does Listing 8.6 use too many constants, thereby obscuring the logic? Or should it have even more constants such as for the number of rows or number of stars?
Exercise 8.23* (Archery) Write the `drawArrow` method. Aim the arrow at the target.
Exercise 8.24* (Archery) Complete your `drawBullseye` method: Put the target on a stand and make the target look three-dimensional.
Exercise 8.25** Rewrite Listing 8.5 to not use the `translate` method. Create a new Rectangle for each stripe.
Exercise 8.26** Create an Applet that places several dozen rectangles of randomly chosen positions, width, and height on the drawing area, using four different colors. Make it look like a Mondrian. Or would ovals look even better?

8.5 Animation In A JApplet

The only thing left is to animate the flag, i.e., have it dance. One common way to do this to have the applet's `start` method repeatedly set a new location for the drawing and then call the applet's `paint` method. The following two JApplet methods are needed:

- `someJApplet.setVisible(true)` makes the drawing area visible. The browser automatically does this before it calls the applet's `paint` method, but you need to do it yourself if you call the `paint` method from the `start` method. Otherwise, whatever the `start` method draws will not show up. Naturally, you can hide the drawing area by using a parameter value of `false` instead of `true`.

- `someJApplet.getGraphics()` returns the Graphics object that the applet uses. You need this object in order to call `paint` from the `start` method.

The browser executes an applet's `start` method when it returns to the web page that contains it after having browsed elsewhere. If your applet provides a method with the heading `public void stop()`, the browser will execute that method when it leaves the applet's web page. This is important for animations that go on for a long time, to keep them from taking up resources when the applet is not displayed. However, you will not need a `stop` method for this Flag applet.

If you only want an action performed when the browser first loads your applet, but not repeated every time the browser returns to the web page, you should put that action in a method with the heading `public void init()`.

Flag analysis and design

Analysis: You do not know just what the client considers a decent dance routine, so you decide to make a reasonable estimation and then see how the client likes it. You decide to have the flag dance to the right, bobbing up and down, then bob back to the left, repeating this cycle five times and then stopping.

Design: To have the Flag dance to the right, you could have `itsLeftEdge` change from 10 to 130 one pixel at a time (since the Flag is 173 pixels wide and you have 320 pixels of space). To get the bobbing effect, you should change its placement below the top of the frame, which is determined by `itsTopEdge`. The first 10 times that `itsLeftEdge` increments, you could have `itsTopEdge` increase by 4s (from 20 up to 60); the next 10 times it decreases by 4s (from 60 back to 20); the next 10 it increases by 4s, the next 10 it decreases by 4s, etc.

Dancing to the left repeats this process except `itsLeftEdge` decrements each time. Much effort is required to get the dancing to the right into Java logic:

```
private void danceRight()  // called by Dancer's start method
{  itsTopEdge = 20;
   for (itsLeftEdge = 10;  itsLeftEdge < 130;  itsLeftEdge++)
   {  paint();
      itsTopEdge += (itsLeftEdge / 10 % 2 == 1) ?  4  :  -4;
   }
}  //=====================
```

Good programming style dictates that `itsLeftEdge` and `itsTopEdge` should be parameters of the call to `paint` instead of instance variables, because of how they are used. But they cannot be, since `paint()` is a method inherited from the JApplet superclass with only one parameter, of type Graphics. So they are instance variables.

When you try out the dancing flag, you see that it is looking good except that the flag moves way too fast. You need a way to slow it down. A handy method in the System class gives the current time, measured in milliseconds since January 1, 1970, as a long value (not int). So you add a `pause` method to the Dancer class to wait a given number of milliseconds, and you insert a `pause(50)` command after each call of the `repaint` method. This completes the final iterative Version 4 of the Flag software, shown in Listing 8.7 (see next page).

The `pause` method is declared as a class method to make it clear that it does not mention any instance variables or instance methods of the Dancer object. It calculates the `timeToQuit` as the specified number of milliseconds after the time at which the `pause` method is called. Then the while-statement loops, doing nothing, until that time comes. This **busywait logic** is inferior to the use of Timers, as discussed in Chapter Nine.

Exercise 8.27 Explain why `itsTopEdge += 4 * (itsLeftEdge / 10 % 2 * 2 - 1)` gives the same result as the assignment statement in `danceRight`. Discuss which of the two expressions is clearer.

Exercise 8.28 (Archery)** Animate the Archery frame (or applet, as the case may be) by having the arrow fly into the target. Repeat the flight 15 times. Every fourth time, have the target jump up in the air just before the arrow hits, then come back down. Print the words "you missed!" each of those times.

Exercise 8.29* (Archery) Try out various pause periods for the Archery frame until you find ones you like. Make the pause between arrow launches far longer than the pauses between changes in position of the arrow as it flies.

Listing 8.7 The Dancer applet, Version 3 (final)

```
import javax.swing.JApplet;
import java.awt.*;

public class Dancer extends JApplet
{
   private int itsLeftEdge;
   private int itsTopEdge;


   public void start()
   {  setVisible (true);
      for (int count = 0;  count < 5;  count++)
      {  danceRight();
         danceLeft();
      }
   }  //=====================

   private void danceRight()
   {  itsTopEdge = 20;
      for (itsLeftEdge = 10;  itsLeftEdge <= 129;  itsLeftEdge++)
      {  paint (getGraphics());
         pause (50);
         itsTopEdge += (itsLeftEdge / 10 % 2 == 1)  ?  4  :  -4;
      }
   }  //=====================

   private void danceLeft()
   {  itsTopEdge = 20;
      for (itsLeftEdge = 129;  itsLeftEdge >= 10;  itsLeftEdge--)
      {  paint (getGraphics());
         pause (50);
         itsTopEdge += (itsLeftEdge / 10 % 2 == 1)  ?  -4  :  4;
      }
   }  //=====================

   private static void pause (int wait)
   {  long timeToQuit = System.currentTimeMillis() + wait;
      while (System.currentTimeMillis() < timeToQuit)
      {  }  // take no action
   }  //=====================

   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.setColor (Color.white);
      page.fill (new Rectangle (0, 0, 320, 160));  // clear area
      new Flag (page, itsLeftEdge, itsTopEdge);
   }  //=====================
}
```

8.6 Review Of The Software Development Paradigm

You have now seen all the basic elements of a workable and efficient process for developing software. This section reviews them and expands on them to be applicable to a very large range of problems. Some software situations will not call for all of the elements, but most complex ones will.

Stage 1: Analysis Study the problem to see exactly what is required of the software. See what parts of the description can be interpreted in more than one way, then find out from the client which is wanted. If you are working under contract with the client, write out the details of these specifications in a document that can be attached to the contract. The primary criteria for the specifications are that (a) they be understandable by the client and a judge (to enforce the contract so you get paid), and (b) they be precise enough that designers and implementers do not need to return to you for further clarification.

Make sure that one part of the specifications does not contradict another part. Make sure that all possible cases have been covered, not just the ones you expect to occur. And make sure that the requirements are **feasible** -- that they can be implemented in software without an unacceptable slowness of response.

Consider which sets of input data you will need to test each aspect of the software. Work out what the output or display will be in each case. Review the specifications to help you decide on good test sets. Use this opportunity to make sure that the specifications are clear and complete.

Stage 2: Logic Design Lay out the overall sequence of operations that the software will perform from beginning to end. Keep it down to a 5- to 10-step description. For event-driven software, as you will see in Chapter Nine, it may be no more than the following two steps:

1. Prepare the frame or applet with buttons, textfields, and whatnot.
2. Wait for someone to click or take other action.

Stage 3: Object Design Decide on the major kinds of objects that will play a role in the software. Decide what operations each will carry out. For each major sub-task that the software has, you should have an object that performs that task, or a set of objects that cooperate in performing that task. Look for objects in your personal library or the standard library that can perform the tasks. If those are not sufficient, see if you can add to the capabilities of existing objects. And if that does not work, invent entirely new objects to perform the tasks.

Stage 4: Version 1 of Iterative Development Choose a subset and/or simplification of the specifications that looks like it can be implemented in a short period of time. Develop the logic for that subset. Structured Natural Language Design is highly useful for complex algorithms. Run whichever of the test sets are appropriate to see if this version of the software accomplishes the goal set for Version 1.

Stages 5, 6,...: Versions 2, 3,... of Iterative Development Choose a larger subset of the specifications each time, adding more functionality, until you eventually reach the end result. Quite often the top level of the software does not change as you progress through these versions; instead, the object classes become more effective. It is normal to discard 10% or more of what you had on the previous version to make it better (even Brahms threw out much of what he wrote; what is left of Brahms' work is of uniformly outstanding beauty -- a word to the wise).

Stage N: Run the tests Make sure that each test set produces the right output. For complex programs, this testing process should be automated. You can do this by e.g. replacing the bodies of the IO methods by logic that reads input from a disk file (rather than the user) and writes output to a disk file. You can then run a program to compare the actual output file with another disk file that contains the expected output. That comparison program should report on any discrepancy between the two files so you know what tests found a bug.

Nota Bene: The technical definition of a **bug** is any disagreement with the specifications. If you do something your way instead of the client's way just because you think it looks better and performs better, it is still a bug. You should either (a) convince the client that

your way is better than the client's way, or (b) do it the client's way, or (c) refuse to take the job. Remember who is paying for it. Remember Mozart. Nota Multum Bene: In the case of a homework assignment for a course, the instructor is your client.

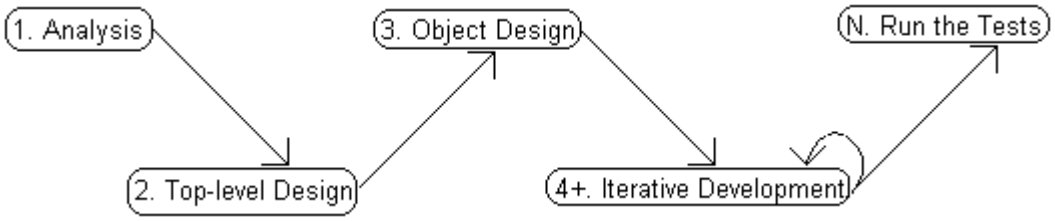


Figure 8.6 Five (or more) stages of software development

8.7 Common Looping Patterns

You have now seen many segments of logic that use loops, whether while-statements, for-statements, or do-while-statements. You have probably noticed that some patterns appear over and over. Probably more than half of all loops you run across fit one of the patterns discussed in this section.

Deciding when the loop stops

Sentinel-controlled loop Each iteration of the loop reads one value from the user or from a disk file. These are data values to be processed as they are read in or to be stored for later processing. When you reach the end of the data values to be processed, the next value obtained is special value that is recognizable as not suitable for processing. This **sentinel** signals the end of the sequence of values.

The while-statements in Listing 7.3 and Listing 7.4 have this logic, where you obtain a value whose name is `null` after you have obtained all of the data values to be processed. So the sentinel value here is a Worker with a `null` name:

```
Worker data = new Worker (file.readLine());
while (data.getName() != null)
{  data.processThisDataValue();
   data = new Worker (file.readLine());
}
```

Count-controlled loop The number of times N you want to carry out a process is known when the loop begins. A counter keeps track of the number of iterations and the loop stops when you have done the process N times, or sooner if you accomplish the purpose of the loop before N times through the loop. An example is the following from Listing 8.6:

```
for (int row = 0;  row < 9;  row++)
{  int start = (row % 2 == 0)  ?  x + PIP : x + PIP + 6;
   drawRowOfStars (page, x + UNION_WIDE - STAR, start,
                   y + PIP + row * STAR);
}
```

Iterator-controlled loop You have a sequence of values to be processed. You keep track of a position in that sequence. Each iteration of the loop advances the position by one step. The loop terminates when you come to the end of the sequence, or sooner if you accomplish the purpose of the loop before the end. If the sequence is stored in an array, the index in the array is the iterator. If it is a sequence of slots for a Vic object, you use `moveOn()` to advance the position indicator, as follows:

```
for (int k = 0;  k < par.length();  k++)
   if (par.charAt (k) == ' ')
      return par.substring (0, k);
return par;

for (Vic sam = new Vic();  sam.seesSlot();  sam.moveOn())
   sam.processThisSlot();
```

User-controlled loop You repeatedly ask for input from the user until he or she provides a value that indicates the loop is to terminate. One example is an attempt to get an acceptable input:

```
int choice = IO.askInt ("Enter a number from 10 to 20: ");
while (choice < 10 || choice > 20)
{  IO.say (choice + " is not in the range from 10 to 20.");
   choice = IO.askInt ("Enter a number from 10 to 20: ");
}
```

Another example is a menu-driven loop, where you ask the user which of several choices he or she wishes to make. The various choices determine what is done inside the loop, except that one of the choices tells you that the user wishes to terminate the entire process. This is actually a variant of the sentinel-controlled loop:

```
String prompt = "Enter A to add, D to delete, P to print, "
                + " or E to exit: ";
char choice = IO.askLine (prompt).charAt (0);

while (choice != 'E')
{  processCurrentChoice (choice);
   choice = IO.askLine (prompt).charAt (0);
}
```

Deciding what to do inside the loop

Those four patterns categorize how you decide when a loop should terminate. The other part of a loop is the action it performs at each of the items it processes. In each of the following common action patterns, the same iterator-controlled looping pattern is used for uniformity, namely, processing the elements of an array. All of these action patterns can occur in sentinel-controlled loops and count-controlled loops as well.

Count-cases looping action The problem is to count how many of the items to be processed have a certain value. To do this, you initialize a counter to zero before the loop, then you add 1 each time you see an item with the required property:

```
int count = 0;
for (int k = 0;  k < a.length;  k++)
   if (someProperty (a[k]))
      count++;
return count;
```

Some-A-are-B looping action The problem is to find out whether at least one of the items has a certain property. You do this by returning `true` when you see an item with the required property as you progress through the loop. If you exit the loop without having returned `true`, then return `false` as the answer:

```
for (int k = 0;  k < a.length;  k++)
   if (someProperty (a[k]))
      return true;
return false;
```

An alternative is to make the loop stop when you see some item with the required property. That puts the `return` statement only at the end of the method. The body of the for-statement has no processing, so we write it as a while-statement. Its only purpose is to find the right value of `k`.

```
int k = 0;
while (k < a.length && ! someProperty (a[k]))
   k++;
return k < a.length;
```

The condition in the for-statement takes advantage of short-circuit evaluation: Evaluating `someProperty(a[k])` when `k` is not less than `a.length` would throw an IndexOutOfBoundsException.

All-A-are-B looping action The problem is to find out whether all of the items have a certain property. This is sort of the opposite of the Some-A-are-B looping action:

```
for (int k = 0;  k < a.length;  k++)
   if ( ! someProperty (a[k]))
      return false;
return true;
```

Process-some-cases looping action The problem is to apply a certain process to those items that have a certain property. An example of this is the while-statement in Listing 7.7, where the process is to store the items away for later use. Another example is the last for-statement in the same listing, where the process is to print certain items in the sequence. Processing could also be modifying the state of object items:

```
for (int k = 0;  k < a.length;  k++)
   if (someProperty (a[k]))
      processCurrentItem (a[k]);
```

Find-max-or-min looping action The problem is to find the largest of the items, or sometimes to find the smallest of the items. The two cases are analogous, so only the finding the largest is discussed here. The largest item you have seen so far is stored in say `largestSoFar`. To process a given item, you compare it with the `largestSoFar`. If the current item is larger, replace the `largestSoFar` by that item, otherwise make no change. But you cannot apply this logic to the first item, because you have no value for `largestSoFar` at that point. So you treat the first item differently:

```
Object largestSoFar = a[0];
for (int k = 1;  k < a.length;  k++)
   if (largestSoFar.compareTo (a[k]) < 0)
      largestSoFar = a[k];
return largestSoFar;
```

In each of these looping action patterns, you need to find out what you are to do if there are zero items to process. The Count-cases action returns 0, the Some-A-are-B action says no one is, the All-A-are-B action says that they all are (which is vacuously true, since none of them are not). But the Find-max-or-min action returns an invalid value: `a[0]` is garbage or it throws an IndexOutOfBoundsException because the array has length zero. So you need to put `if (a.length == 0) return null` or some other crash-guard at the beginning of that logic, whatever is appropriate to the situation.

Exercise 8.30 Rewrite the All-A-are-B example to put the return after the for-statement, as illustrated for the Some-A-are-B action.
Exercise 8.31 Rewrite the Find-max-or-min example to find the minimum rather than the maximum.

Exercise 8.32 Would it work to crash-guard the Find-max-or-min example against a length of zero by replacing the first statement by `Object largestSoFar = (a.length == 0) ? null : a[0];`?
Exercise 8.33* One of the while-statements in Listing 6.1 is a sentinel-controlled loop. Which one is it?
Exercise 8.34* Categorize the looping actions of all loops in Listing 7.8 and Listing 7.9.
Exercise 8.35** Categorize the looping actions of all loops in Chapter Six listings and answers to exercises.

8.8 Case Study In Animation: Colliding Balls

This section will give you additional experience with applets, arrays of objects, inheritance, and especially polymorphism. No important new material is presented in this section.

You want to create an applet with a great deal of action. You decide to have about twenty red balls bouncing around within a rectangle. When one red ball hits another red ball, it "poisons" that other ball. The poisoned ball turns blue and stops moving. It sits in that one spot for a while. Then it disappears and is reborn along the bottom as a moving red ball. This action continues for a minute or two, then it stops.

Design of the main logic

This will not be too hard to design if you work up to it slowly. You will need one JApplet object and many Ball objects. The overall logic is (1) create 21 balls, then (2) start them bouncing, hitting, and dying. The first step can be implemented in just three lines:

```
final static int SIZE = 21;
for (int k = 0;  k < SIZE;  k++)
   item[k] = new Ball();
```

The `start` method of the applet will execute for several thousand cycles, whatever turns out to be around two minutes (you will have to try out several numbers to see which works best). Each cycle will check out each of the 21 balls and take the appropriate action depending on its current status. See the accompanying design block.

DESIGN for the main logic of Colliding Balls
1. For several thousand cycles, do...
    1a. For each one `x` of the 21 Balls, do...
        1aa. If `x` is healthy, then...
            1aaa. Move `x` and see if it hits any other Ball.
            1aab. If so, and if that other Ball is healthy, then...
                Poison that other healthy Ball.
        1ab. Otherwise, if `x` is "dying" then...
            It deteriorates a little more.
        1ac. Otherwise, `x` is "dead" so...
            It reappears as a new healthy Ball.
    1b. Paint the drawing area to show the changed position of the 21 Balls.

Painting the drawing area should begin by clearing out the existing drawing with a filled rectangle. Then it can just draw each Ball's image, perhaps as follows:

```
for (int k = 0;  k < SIZE;  k++)
   item[k].drawImage (page);
```

Both the `start` method and the `paint` method have to refer to the `item` array, so it must be an instance variable of the applet. To be consistent with other nomenclature in this book, we call it `itsItem` rather than `item`.

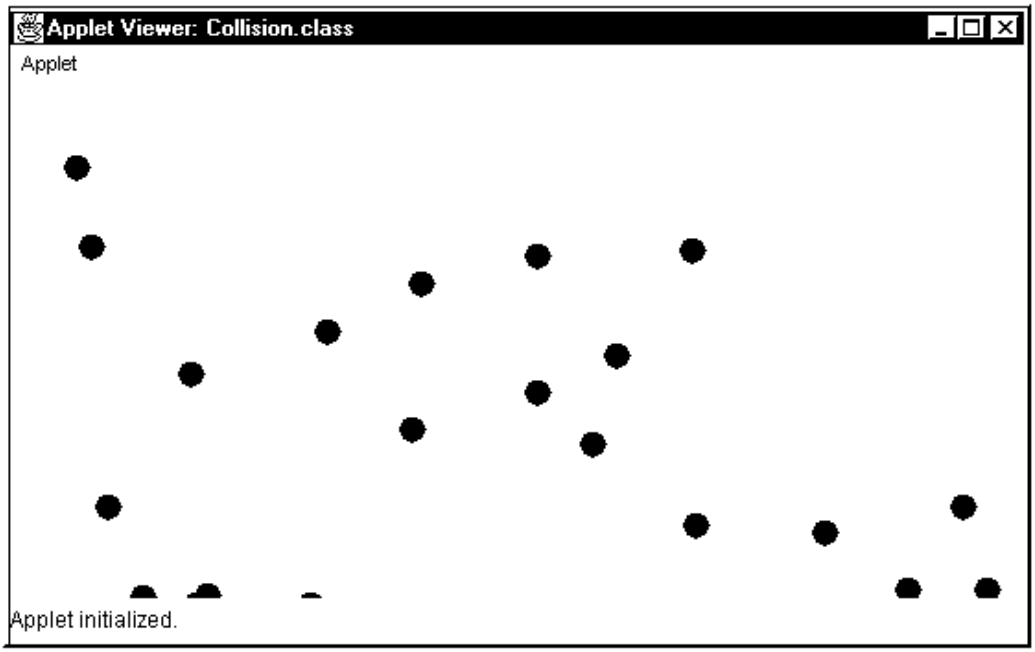


Figure 8.7 The Colliding Balls applet

The logic for telling whether a certain Ball `guy` hits anyone else is to go through the array looking at all the other Balls one at a time until you see one that `guy` does not miss by much. If that one is healthy, then it gets poison.

Listing 8.12 (see next page) expresses this Collision applet logic in Java. It is clear that the Ball class has to be told the dimensions of the rectangle the Balls can move within. This must be done before any Balls can be created, so a special initializing class method is called. The parameters are `getWidth()` and `getHeight()`, two methods that can be used with any applet to find its width and height.

The purpose of the while statement at the end of the `start` method is to pause for a fraction of a second at the end of each cycle to keep the Balls from moving too fast on the screen. This is the same logic as used in the Flag software. The purpose of the while statement in the `moveAndCheckHits` method is to go through all 21 Balls, adding 1 to the index each time, until you see a Ball that you do not miss (by more than 2 or 3 pixels) or you come to the end of the sequence of Balls.

The Ball class

If an object is to move on the screen, it needs to keep track of its current position. These are the instance variables `<itsXspot, itsYspot>` in the Ball class. It also keeps track of the amount to change `x` by, and the amount to change `y` by, each time it moves. These are the instance variables `<itsXmove, itsYmove>` in the Ball class.

Listing 8.12 The Collision JApplet

```
import javax.swing.JApplet;
import java.awt.*;

public class Collision extends JApplet
{
   public static final int NUM_CYCLES = 4000;
   public static final int PAUSE = 90;
   public static final int SIZE = 21;      // number of Balls
   ////////////////////////////////
   private Ball[] itsItem = new Ball [SIZE];  // the 21 Balls


   public void start()
   {  this.setVisible (true);
      Ball.initialize (this.getWidth(), this.getHeight());
      for (int k = 0;  k < SIZE;  k++)
         itsItem[k] = new Ball();
      for (int cycle = 1;  cycle <= NUM_CYCLES;  cycle++)
      {  for (int k = 0;  k < SIZE;  k++)
            if (itsItem[k].isHealthy())
               moveAndCheckHits (itsItem[k]);
            else if (itsItem[k].isDying())
               itsItem[k].deteriorates();
            else
               itsItem[k] = new Ball();
         paint (this.getGraphics());
         long timeToQuit = System.currentTimeMillis() + PAUSE;
         while (System.currentTimeMillis() < timeToQuit)
         {  }  // take no action
      }
   }  //=====================

   private void moveAndCheckHits (Ball guy)
   {  guy.move();
      int c = 0;
      while (c < SIZE && itsItem[c].misses (guy))
         c++;
      if (c < SIZE && itsItem[c].isHealthy())
         itsItem[c].getsPoison();
   }  //=====================

   public void paint (Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.setColor (Color.white);
      page.fill (new Rectangle (0, 0, getWidth(), getHeight()));
      for (int k = 0;  k < SIZE;  k++)
         itsItem[k].drawImage (page);
   }  //=====================
}
```

To get the Ball rolling, you can pick `<itsXspot, itsYspot>` at random along the bottom of the rectangle where the Balls will be moving, though not in the first or last ten percent of the area (so it will look better). Some experimentation came up with a change in the x-direction of plus or minus 2 pixels per cycle, and an initial change in the y-direction of -1 to -3 pixels per cycle. This is because you subtract from `y` each time to have it move up from the bottom of the rectangle.

One way to have the Ball stand still for say 100 cycles before it is reborn is to subtract 1 from a counter on each cycle, starting from 100, and call it dead when the counter reaches zero. You can start a new Ball with the counter (called `itsStatus`) at 100, and keep it there until the Ball is hit. This gives you a way of telling whether a particular Ball is healthy. When it is hit, change `itsStatus` to 99, which signals that the Ball is not healthy. Then deterioration simply amounts to subtracting 1 from `itsStatus` each time. It is "dying" as long as `itsStatus` remains positive but less than 100. Listing 8.13 contains this much of the Ball class.

Listing 8.13 The Ball class except for move, misses, and drawImage

```
import java.util.Random;
import java.awt.geom.Ellipse2D;
import java.awt.*;

public class Ball extends Object
{
   private static final int HEALTHY = 100;  // cycles to languish
   private static Random randy = new Random();
   private static int theRight;    // right side of the screen
   private static int theBottom;   // bottom edge of the screen
   ////////////////////////////////
   private int itsXspot;           // x-value of current location
   private int itsYspot;           // y-value of current location
   private int itsXmove;           // change in x each cycle
   private int itsYmove;           // change in y each cycle
   private int itsStatus;          // tells how healthy it is


   public static void initialize (int rightSide, int bottom)
   {  theRight = rightSide;
      theBottom = bottom;
   }  //=====================

   public Ball()
   {  itsXspot = (int) (theRight *
                        (0.1 + 0.8 * randy.nextDouble()));
      itsYspot = theBottom - randy.nextInt (40);
      itsXmove = randy.nextInt (5) - 2; // range -2 to 2
      itsYmove = randy.nextInt (3) - 3; // range -1 to -3
      itsStatus = HEALTHY;
   }  //=====================

   public boolean isHealthy()
   {  return itsStatus == HEALTHY;
   }  //=====================

   public void getsPoison()
   {  itsStatus = HEALTHY - 1;
   }  //=====================

   public boolean isDying()
   {  return itsStatus < HEALTHY && itsStatus > 0;
   }  //=====================

   public void deteriorates()
   {  itsStatus--;
   }  //=====================
}
```

Movement for one cycle is mainly a matter of adding `itsXmove` to `itsXspot` and adding `itsYmove` to `itsYspot`. However, you have to check whether the Ball has reached one of the boundaries of the rectangle within which it is to move. If it reaches the left or right side, you only need replace `itsXmove` by the negative of `itsXmove`. If it reaches the top or bottom, you replace `itsYmove` by the negative of `itsYmove`. For instance, `itsXmove = -itsXmove` changes 2 to -2 but changes -2 to 2.

The image you draw is just a ball shape, using an Ellipse2D.Double object from the `java.awt.geom` package. You have to choose a diameter; call it UNIT. Since the ball is drawn to the right and below `<itsXspot, itsYspot>`, you want to have the ball "bounce" off the right edge when it is within UNIT pixels of that side, and to bounce off the bottom when it is within UNIT pixels of the bottom. On the left edge, you have it bounce when `itsXspot` is 1, and similarly off the top edge.

A test run showed that this causes a problem. Balls are initially "born" in the area just above the lower boundary. If a Ball is born less than UNIT pixels above the lower boundary and moving upwards, and thus `itsYmove` is negative, it is treated as if it had just arrived there from above, and `itsYmove` is changed to be positive. That makes it move further down, so `itsYmove` is changed to be negative again. The result is that the Ball is stuck in that lower area. Fixing this problem is left as an exercise.

How do you tell when the executor misses the other guy? Of course, you count it as a miss if the guy is the executor itself. Otherwise, you can count it as a miss if the distance between them (looking at `<itsXspot, itsYspot>`) is more than 2. This logic is in Listing 8.14 (see next page), where the standard library `Math.abs` method is used to find the absolute value of the difference of two numbers: `Math.abs(x)` is the absolute value of x.

The `drawImage` method just sets the drawing Color to red or blue depending on whether the ball is healthy and then draws the circle.

Major programming project

Define three subclasses of the Ball class called Scissors, Paper, and Rock. At the beginning of the applet's `start` method, create seven objects from each subclass of Ball instead of creating SIZE ordinary Balls. Have the `drawImage` method in each subclass produce a shape that actually looks like Scissors or Paper or Rock. To make the Rock, look up the Arc2D.Double constructor with PIE type. Use various colors.

The challenging part is what happens when two of these subclass types collide. Whether the Scissors hits the Rock or the Rock hits the Scissors, the Scissors should die and come back as a Rock. Similarly, if Scissors and Paper collide, the Paper should die and come back as Scissors. And if Paper and Rock collide, the Rock should die and come back as Paper. If two objects of the same type collide, they should both die and come back as one each of the other two types. For instance, if two Scissors collide, one comes back as a Rock and the other as a Paper. Use the `instanceof` operator that you have not seen before, e.g., `someBall instanceof Paper` will be `true` if `someBall` is an object from the Paper class but not if it is an object from the Rock or Scissors class.

You will need to have a `hitsOther` method in each of the four classes and replace the call of `getsPoison` in `moveAndCheckHits` by `itsItem[c].hitsOther(guy)`. This polymorphic `hitsOther` method will take the appropriate action in each case, poisoning the right object. And you need to tell each Ball when it is poisoned what kind of thing it is to be reborn as when the 100 cycles are up. You should also display a running count of each kind of object on the right side.

Listing 8.14 The Ball class, the three remaining methods

```
// the Ball class, completed

   public static final int UNIT = 15;       // width of figure


   public void move()
   {  itsXspot += itsXmove;
      itsYspot += itsYmove;
      if (itsXspot <= 1 || itsXspot >= theRight - UNIT)
         itsXmove = - itsXmove;
      if (itsYspot <= 1 || itsYspot >= theBottom - UNIT)
         itsYmove = - itsYmove;
   }  //======================

   public boolean misses (Ball guy)
   {  return this == guy || Math.abs (itsXspot - guy.itsXspot)
                         + Math.abs (itsYspot - guy.itsYspot) > 2;
   }  //======================

   public void drawImage (Graphics2D page)
   {  if (isHealthy())
         page.setColor (Color.red);
      else
         page.setColor (Color.blue);
      page.fill (new Ellipse2D.Double (itsXspot, itsYspot,
                                       UNIT, UNIT));
   }  //======================
```

Exercise 8.36 Fix the `move` method so that the just-born balls do not get stuck at the bottom of the drawing area.
Exercise 8.37* What would be the formula for the `misses` method to see if one object is within a circle of radius 2 of the other? Is the result the same? For what radii is the result different?
Exercise 8.38* Revise the `drawImage` method so that a dying Ball flickers between blue and gray, half-and-half. Hint: use the expression `itsStatus % 10 < 5`.
Exercise 8.39* Revise the Collision class so that the action continue for exactly two minutes rather than 4000 cycles.

8.9 Implementing The Turtle Class

Chapter One describes software that lets you use do graphics programming with statements such as the following, which make a black circle of radius 200 and then a red circle of radius 100:

```
Turtle sue = new Turtle();  // initial drawing Color is black
sue.swingAround (200);   // circle radius 200 centered at sue
sue.switchTo (Turtle.RED);
sue.swingAround (100);   // circle radius 100 centered at sue
```

Listing 8.15 gives half of the Turtle class needed for this software. It should all be completely understandable to you except for the definition of DEGREE.

Listing 8.15 The Turtle class, some methods postponed

```
import java.awt.*;
import javax.swing.JFrame;
import java.awt.geom.*;

public class Turtle extends Object
{  public static final double DEGREE = Math.PI / 180;
   public static final int WIDTH = 760, HEIGHT = 600;
   public static final Color RED = Color.red, BLUE = Color.blue,
                BLACK = Color.black,      GRAY = Color.gray,
                YELLOW = Color.yellow,    PINK = Color.pink,
                ORANGE = Color.orange,    GREEN = Color.green,
                MAGENTA = Color.magenta,  WHITE = Color.white;
   private static Graphics2D page = null;     // for drawing
   ////////////////////////////////
   private double heading = 0;         // heading initially east
   private double xcor = WIDTH / 2;    // centered horizontally
   private double ycor = HEIGHT / 2;   // centered vertically too

   /** Set the drawing Color to the given value.  */

   public void switchTo (Color given)
   {  page.setColor (given);
   }  //======================

   /** Write words without changing the Turtle's state.  */

   public void say (String message)
   {  page.drawString (message, (int) xcor, (int) ycor);
   }  //======================

   /** Pause the animation for wait milliseconds.  */

   public void sleep (int wait)
   {  long timeToQuit = System.currentTimeMillis() + wait;
      while (System.currentTimeMillis() < timeToQuit)
      {  }  // take no action
   }  //======================

   /** Make a circle of the given radius, Turtle at center. */

   public void swingAround (double radius)
   {  if (radius > 0.0)
         page.draw (new Ellipse2D.Double (xcor - radius,
                    ycor - radius, 2 * radius, 2 * radius));
   }  //======================
}
```

Trigonometry

The Turtle's current position is tracked by the two variables <xcor, ycor>. Suppose the Turtle is oriented 30 degrees counterclockwise from a due-east heading, at the far left of Figure 8.8. If told to move say 100 Turtle steps, it has to calculate its new <x,y> coordinates from its original ones. The hypotenuse of the triangle is 100 pixels, so the vertical side of the triangle is 50 pixels (half the hypotenuse) and the base of the triangle is about 86.6 pixels (multiplying the length of the hypotenuse by half of the square root of 3). So the new <x,y> coordinates are 50 pixels above and 86.6 pixels to the right of its original position.

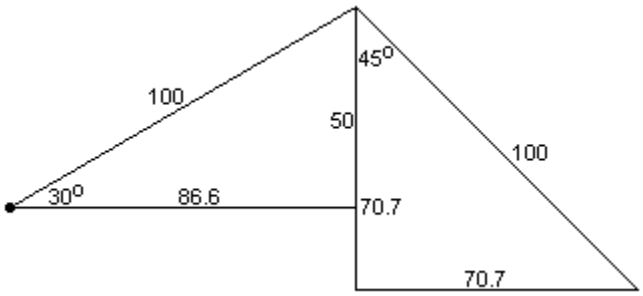


Figure 8.8 Distances and degrees

If the Turtle is then turned to the right 75 degrees, so it is facing southeast, and is then asked to move 100 Turtle steps, its new `<x,y>` coordinates are 70.7 pixels below and 70.7 pixels to the right of the original position (because half the square root of 2 is 0.707). In general, the Turtle needs to add the cosine of its heading to its current x-position `xcor` and the negative of the sine of its heading to its current y-position `ycor`. The reason for subtracting for the y-position is that y-values on a Graphics page are increasing in the downward direction, not the upward direction as is normally done in trigonometry.

Trigonometric methods from the Math class

The Math class in `java.lang` has many useful class methods. In particular, `Math.cos(heading)` is the cosine of the heading and `Math.sin(heading)` is the sine of the heading. So the Turtle needs to add the former to `xcor` and subtract the latter from `ycor`. However, these trigonometric functions are calculated from the number of radians, not the number of degrees. `Math.PI` is defined to be the number of radians in 180 degrees, so one degree is `Math.PI / 180` radians.

You can reduce the amount of calculation the Turtle has to do if you keep track of the heading in terms of radians instead of degrees. So each time the heading changes by the amount `left`, you add `left * DEGREES` to the `heading`. The initial value of the `heading` is zero, which conventionally indicates an orientation due east.

The upper part of Listing 8.16 (see next page) contains the two methods `move` and `paint`, which should now be understandable. The methods are defined to return the Turtle itself rather than being void methods, because it is sometimes convenient to be able to write chained statements such as the following:

```
sam.paint(30, 100).paint(-75, 100).move(-135, 70.7);
```

The Turtle constructor

You may use Turtles to draw on an applet by using `new Turtle (applet.getGraphics(), xstart, ystart)` (the last two parameters give the initial position of the Turtle). This was not mentioned in Chapter One because you had not learned to used applets yet. A subtle point is that a new Graphics2D object is created each time the applet is covered and then uncovered, so the Graphics2D `page` value keeps changing. You should generally create the Turtle in the `paint` method rather than in the `start` or `init` method to establish the new page reference. Otherwise your drawing may not reappear.

This software is designed to be used by rank beginners, before they know how to make an applet or use HTML. In particular, it has to run as a stand-alone application rather than as an applet. So an alternative constructor is provided for JFrames in Listing 8.16. Read Section 9.1 if you want to understand the statements in that constructor.

Listing 8.16 The rest of the Turtle class

```
// the Turtle class, completed

   /** Rotate left by left degrees; MOVE for forward steps. */

   public Turtle move (double left, double forward)
   {  heading += left * DEGREE;
      xcor += forward * Math.cos (heading);
      ycor -= forward * Math.sin (heading);
      return this;
   }  //=====================

   /** Rotate left by left degrees; PAINT for forward steps. */

   public Turtle paint (double left, double forward)
   {  heading += left * DEGREE;
      double x = xcor + forward * Math.cos (heading);
      double y = ycor - forward * Math.sin (heading);
      page.draw (new Line2D.Double (xcor, ycor, x, y));
      xcor = x;
      ycor = y;
      return this;
   }  //=====================

   /** Fill a circle of the given radius, Turtle at center. */

   public void fillCircle (double radius)
   {  page.fill (new Ellipse2D.Double (xcor - radius,
                  ycor - radius, 2 * radius, 2 * radius));
   }  //=====================

   /** Create a turtle on a given page at <xstart, ystart>. */

   public Turtle (Graphics g, double xstart, double ystart)
   {  page = (Graphics2D) g;
      xcor = xstart;
      ycor = ystart;
   }  //=====================

   public Turtle()
   {  if (page == null)
      {  JFrame area = new JFrame ("Turtle drawings");
         area.addWindowListener (new Closer());  // in Chapter 9
         area.setSize (WIDTH, HEIGHT);
         area.setVisible (true);
         page = (Graphics2D) area.getGraphics();
         area.paint (page);
      }
   }  //=====================
```

You may add extra methods to give the Turtle additional capabilities, e.g.:

```
   public void fillBox (double width, double height)
   {  page.fill (new Rectangle2D.Double (xcor - width / 2,
                 ycor - height / 2, width, height));
   }  //=====================
```

Exercise 8.40 What changes would be required in the entire Turtle class if the user prefers to specify degrees of turn to the right in `move` and `paint` and prefers that the Turtle initially be oriented due north?
Exercise 8.41* Turtle software could execute faster if it had three convenience methods, one to turn left a number of degrees without moving, the other to move or paint forward a certain distance without turning. Write the three methods without calling `move` or `paint`.
Exercise 8.42* Revise the FractalTurtle's `drawTree` method (in Listing 1.11) so that each of the two branches off the tree trunk is at a 45-degree angle instead of a 30-degree angle.

*8.10 About Font, Polygon, and Point2D.Double (*Sun Library)*

This section discusses three additional classes that may be useful in working with graphics.

Font objects (from java.awt)

You may control the shape and size of the letters and other characters you write on a Graphics object. The kinds of font allowed depend on your system, but they always include these five: "Serif" (like Times New Roman), "SansSerif" (like Ariel or Helvetica), "Monospaced" (like Courier), "Dialog", and "DialogInput" (both of which are fonts san serif).

- `new Font(someString, styleInt, sizeInt)` creates a new Font object. `someString` is a name such as "Serif". The `styleInt` can be Font.PLAIN, Font.BOLD, Font.ITALIC, or the sum Font.BOLD+Font.ITALIC. The `sizeInt` value should range from 8 (one-ninth of an inch tall) to 36 (one-half of an inch tall).
- `someGraphics.setFont(someFont)` causes all Strings of characters later drawn on the Graphics object to be drawn according to the given Font object.

Polygon objects (from java.awt, implements Shape)

A Polygon is a number of points connected by line segments, with the last point connected to the first. Rectangles and triangles are the simplest polygonal shapes. Since Polygon implements the Shape interface, you can use the `draw` and `fill` commands for Graphics2D objects.

You specify a Polygon object by giving two arrays of int values, the first being the x-coordinates of the points and the second being the y-coordinates of the points. For example, if you want to draw the diamond-shaped figure with corners at <200,50>, <160,150>, <200,250>, and <240,150>, you could have the following four statements (where the third parameter of the constructor is the number of points):

```
int[4] xvals = {200, 160, 200, 240};  // the x-coordinates
int[4] yvals = {50, 150, 250, 150};   // the y-coordinates
Polygon diamond = new Polygon (xvals, yvals, 4);
page.draw (diamond);
```

- `new Polygon(xvals, yvals, sizeInt)` constructs a Polygon object from two arrays declared as `int[]`. For the `sizeInt` points given by `<x[k],y[k]>` for the various values of `k`, each point is connected to the next, except the figure is completed by connecting `<x[sizeInt-1],y[sizeInt-1]>` to `<x[0],y[0]>`.
- `new Polygon()` constructs a Polygon with zero points.

- `somePolygon.addPoint(xInt, yInt)` adds a new point to the end of the Polygon, so that what previously was the last point now connects to this added point and the added point connects to the first point.
- `somePolygon.translate(dxInt, dyInt)` adds `dxInt` to each x-coordinate and `dyInt` to each y-coordinate of the points in the Polygon.

You have to be sure to put the points in the right order; if the third and fourth points of the diamond shape were switched, you would not have a diamond, you would have two triangles that meet at a point.

Point2D.Double objects (from java.awt.geom)

You may want to use the Point2D.Double class to make your coding more compact and more understandable. The Point2D.Double class has several useful methods:

- `new Point2D.Double(x, y)` constructs a Point at double coordinates `x` and `y`.
- `somePoint2D.getX()` returns the current x-value of the point.
- `somePoint2D.getY()` returns the current x-value of the point.
- `somePoint2D.setLocation(x, y)` re-assigns the coordinates as indicated.

The following illustrate how you can use these objects (`p1` and `p2` denote instances of the Point2D.Double class):

- `new Line2D.Double(p1, p2)` is an alternate constructor using points.
- `someShape.contains(p1)` tells whether p1 is inside the Shape object.
- `someLine2D.ptLineDist(p1)` tells how far `p1` is from the line (a double).

8.11 Review Of Chapter Eight

About the java.awt.Graphics2D class (a subclass of Graphics):

- The **drawing area** is measured in **pixels**, counting in from the left and down from the top. The point `<0,0>` is in the top-left corner. At any given time, one particular color is the **drawing Color** for the Graphics object, used for all drawings.
- `someGraphics.getColor()` returns the current drawing Color.
- `someGraphics.setColor(someColor)` makes it the current drawing Color.
- `someGraphics.drawString(someString, x, y)` prints the characters in the String starting at `<x,y>` where `x` and `y` are both int values.
- `someGraphics2D.draw(someShape)` draws the outline of the rectangle, ellipse, polygon, or whatever shape you supply as the parameter.
- `someGraphics2D.fill(someShape)` draws the shape and fills in its interior with the drawing Color.

About the java.awt.Rectangle class (implementing the java.awt.Shape interface):

- `new Rectangle(x, y, width, height)` constructs a new Rectangle object whose top-left corner is at `<x,y>`, with the given `width` and `height` measured in pixels. If either the `width` or the `height` is negative, the Rectangle will not show up. All parameters are int values for this constructor and the following four methods.
- `someRectangle.grow(width, height)` adds `width` pixels to the left and right sides of the rectangle, and adds `height` pixels to top and bottom of the rectangle. So its width changes by `2*width` and its height changes by `2*height`.
- `someRectangle.translate(x, y)` moves the rectangle `x` pixels to the right and `y` pixels down from its current position. The width and height remain unchanged.

- `someRectangle.setLocation(x, y)` moves the top-left corner of the rectangle to the point `<x,y>`. The width and height remain unchanged.
- `someRectangle.setSize(width, height)` changes the width and height of the rectangle to the specified values. The top-left corner remains unmoved.

About some java.awt.geom classes (all implementing java.awt.Shape):

- `new Line2D.Double(x, y, xend, yend)` constructs a new Line object with one end at `<x,y>` and the other at `<xend,yend>`. All parameters are doubles.
- `new Rectangle2D.Double(x, y, width, height)` constructs a new rectangular object with its top-left corner at `<x,y>`, and of the given `width` and `height`. All parameters are doubles.
- `new Ellipse2D.Double(x, y, width, height)` constructs a new elliptical object whose bounding box has its top-left corner at `<x,y>` and has the given `width` and `height`. All parameters are doubles.
- `new RoundRectangle2D.Double(x, y, width, height, arcWidth, arcHeight)` constructs a new rectangular object with rounded corners, with its top-left corner at `<x,y>`, of the given `width` and `height`, where the width and height of the arcs at the corners are given by the last two parameters. All parameters are doubles.

About the javax.swing.JApplet and java.awt.Component class:

- The `JApplet` class is a subclass of Applet, which has these three methods called by a browser: `init()` when the applet is loaded; `start()` when the applet is to start execution; and `stop()` when the applet is to stop execution. Applet is a subclass of the Panel class, which is a subclass of the Container class, which is a subclass of the Component class. The Component class contains the following six methods.
- `someComponent.paint(someGraphics)` paints this Component. The browser calls it whenever the window is uncovered after having been minimized or covered.
- `someComponent.getGraphics()` returns the Component's Graphics object.
- `someComponent.setVisible(aBoolean)` makes the Component visible or not.
- `someComponent.setSize(width, height)` resizes the Component to the given width and height, both of which must be int values.
- `someComponent.getWidth()` returns the int width of the drawing area in pixels.
- `someComponent.getHeight()` returns the int height of the drawing area in pixels.

About other Sun standard library classes:

- The `Color` class (in `java.awt`) defines thirteen final class variables: `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `yellow`, `red`, and `white`. The constructor `new Color(r, g, b)` with int parameters in the range 0 to 255 creates an opaque Color with the corresponding red/green/blue values; the higher the number, the more you have of that color.
- Read through the documentation for the `java.awt` and `javax.swing` classes partially described in this chapter. Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

Basic programming principles mentioned in this chapter:

- If you don't know where you are going, you are not likely to get there.
- Always put the hard stuff off until later.
- You're only human, you're supposed to make mistakes (per Billy Joel).
- Don't tell the client his/her ideas are dumb if it is both profitable and ethical to create the software he/she wants.
- Don't make stuff up unless you have to.
- Don't be afraid to ask for help from someone who knows.

Common looping patterns and looping actions:

- Common looping patterns include sentinel-controlled logic, count-controlled logic, iterator-controlled logic, and user-controlled logic.
- Common looping actions (actions to take inside the loop) include Count-cases, Some-A-are-B, All-A-are-B, Process-some-cases, and Find-min-or-max.

Other vocabulary to remember:

- **Iterative development** of a large program is several cycles of the following: Develop a complete working program that does more of what the final program is to do than what you had at the end of the previous cycle, making sure that most of what you add will not have to be discarded in a later cycle.
- A **busywait** is a segment of coding that keeps the runtime system busy without accomplishing anything useful except to pass the time. It is sometimes used in animation.
- A **bug** in software is any disagreement with the specifications for that software. In particular, it is a bug to have a method do something more than it is supposed to do. For instance, a method that is specified to print two particular lines has a bug if it prints the wrong two lines, or if it prints less than two lines, or even if it prints the right two lines plus another line.

Answers to Selected Exercises

8.1
```
public void paint (Graphics g)
{   Graphics2D page = (Graphics2D) g;
    page.setColor (Color.yellow);
    page.drawString ("YourName", 10, 50);
    page.setColor (Color.cyan);
    page.drawString ("YourName", 70, 50);
    page.setColor (Color.pink);
    page.drawString ("YourName", 130, 50);
}
```

8.2
```
public void paint (Graphics g)
{   Graphics2D page = (Graphics2D) g;
    page.setColor (Color.green);
    page.draw (new Line2D.Double (10, 50, 40, 50));  // top, 30 pixels wide
    page.draw (new Line2D.Double (40, 50, 40, 140));  // right side, 90 pixels tall
    page.draw (new Line2D.Double (40, 140, 10, 140));  // bottom
    page.draw (new Line2D.Double (10, 140, 10, 50));    // left side
}
```

8.3
```
public void paint (Graphics g)
{   Graphics2D page = (Graphics2D) g;
    // THE OUTER SQUARE
    page.draw (new Line2D.Double (10, 30, 50, 30));
    page.draw (new Line2D.Double (50, 30, 50, 70));
    page.draw (new Line2D.Double (50, 70, 10, 70));
    page.draw (new Line2D.Double (10, 70, 10, 30));
    // THE TWO CENTER DIVIDERS
    page.draw (new Line2D.Double (30, 30, 30, 70));
    page.draw (new Line2D.Double (10, 50, 50, 50));
}
```

8.4
```
public void paint (Graphics g)
{   Graphics2D page = (Graphics2D) g;
    page.draw (new Rectangle (10, 30, 60, 20));
    page.draw (new Rectangle (30, 10, 20, 60));
    page.drawString ("give", 11, 45);
    page.drawString ("blood", 51, 45);
}
```

8.5
```
public void paint (Graphics g)
{   Graphics2D page = (Graphics2D) g;
    Rectangle box = new Rectangle (10, 30, 60, 20);
    page.draw (box);
    box.setSize (70, 30);
    page.draw (box);
    box.setSize (80, 40);
    page.draw (box);
}
```
8.6 Replace each "page.draw" by "page.fill" in the preceding answer, and put these three statements just before the page.fill commands, in order:
```
page.setColor (Color.magenta);
page.setColor (Color.gray);
page.setColor (Color.yellow);
```
8.7
```
public void paint (Graphics g)
{   Graphics2D page = (Graphics2D) g;
    page.draw (new Ellipse2D.Double (30, 40, 20, 20));
    page.draw (new Ellipse2D.Double (50, 40, 20, 20));
    page.draw (new Ellipse2D.Double (70, 40, 20, 20));
}
```
8.11 As given, you can draw a completely blue rectangle and then put white stars on it. If you drew the white stars first, how would you fill in the blue part around them?

8.12 Just double the four constants to 346, 182, 138, 98, 4 and 10. The rest of the Flag class should be written so that all other dimensions are doubled as a consequence.

8.15 Replace TALL by 5, 13 by 20, and "Color.red : Color.white" by "Color.green : Color.orange".

8.16 In the drawStripes method, replace 13 by 20 and replace the if-statement by the following:
```
if (k % 4 == 0)
    page.setColor (Color.green);
else if (k % 4 == 1)
    page.setColor (Color.orange);
else if (k % 4 == 2)
    page.setColor (Color.red);
else // (k % 4 == 3)
    page.setColor (Color.white);
```
8.17 Insert this statement before the for-statement:
```
boolean isRed = true;
```
Replace the if-else-statement by the following:
```
if (isRed)
    page.setColor (Color.red);
else
    page.setColor (Color.white);
isRed = ! isRed;
```
8.18 It is not the 9 or the 2; the flag will have 9 rows alternating among 2 possibilities no matter what the size. Define public static final int WIDTH = 12; then replace EDGE + 6 by EDGE + WIDTH / 2 and replace x += 12 by x += WIDTH.

8.19 Replace the body of the for-statement by:
```
if (row % 2 == 0)
    drawOneRowOfStars (page, EDGE, y + EDGE + row * STAR);
else
    drawOneRowOfStars (page, EDGE + 6, y + EDGE + row * STAR);
```
8.20 Replace row < 9 by row < 6 in the drawStars method, and UNION_WIDE - STAR by UNION_WIDE - 2 * STAR in the drawOneRowOfStars method.

8.27 The expression itsLeftEdge / 10 % 2 is tricky enough, being 1 or 0 depending on whether it is an odd or an even multiple of 10. When you double 1 or 0 and subtract 1, you get either 1 or -1, and multiplying that result by 4 means you are either adding or subtracting 4. But that takes far longer for the human reader of your logic to understand. You do that only if there is a strongly compensating advantage to doing so, and there is not; execution speed is not materially affected.

8.30
```
int k = 0;
while (k < a.length && someProperty (a[k]))
    k++;
return k == a.length;
```
8.31 Change the less-than operator < to the greater-than operator >.

8.32 It would. If the length of the array is in fact zero, the for-statement does not execute at all and so null is returned.

8.36 Make the condition for the second if-statement in the moves method the following:
```
if (itsYspot <= 1 || (itsYspot >= theBottom - UNIT && itsYmove > 0))
```
8.40 Change the initial value of heading to double heading = 90 * DEGREES; also, the first statement of both move and paint should be heading -= left * DEGREES.

9 Event-driven Programming

Overview

This chapter introduces graphics programming and event-handling in the context of data entry for a car rental company. You will see how to use many of the new swing components in the Sun GUI library, such as buttons, textfields, menus, and sliders. If you did not read Chapter Eight, you may simply ignore the very few references to JApplet and Graphics2D commands.

- Sections 9.1-9.2 discuss JFrames and how to handle the event of the user clicking on the closer icon.
- Sections 9.3-9.5 introduce objects that listen for a click of a button or a data entry and react to it. They are defined by "inner classes" so they can access the frame's instance variables.
- Section 9.6 completes Version 1 of the CarRental software, which uses the Model/View/Controller pattern.
- Sections 9.7-9.11 describe additional components and listeners, including sliders, timers, combo boxes, radio buttons, and menus. In the process we complete Version 2 of the CarRental software. Section 9.9 is the first section that uses arrays.

9.1 JFrames, Components, And WindowListeners

You have been hired to create some graphical data entry software by the Wysiwyg Car Rental Agency. Their agents take reservations over the phone, enter them into the database while talking, then send the reservations to the appropriate branch at some airport. You have to develop efficient data-entry forms that are pleasant to work with.

JFrames

The `javax.swing` package contains the JFrame and JApplet classes. JFrames are used for applications and JApplets are used for graphical parts of a web page. You begin developing an application program with a graphical user interface (**GUI**) by defining a subclass of the JFrame class from the `javax.swing` package. Objects of the **JFrame** class represent rectangular windows on the display with a title bar and a border.

The JFrame may have a `paint` method just like that of a JApplet, if you want to make drawings and you do not add any components to the JFrame. `paint` is called by the operating system when the user minimizes the window and then brings it back. But since a JFrame does not have a browser to create your graphical object, set its size to a particular width and height, and make it visible, you will need to do that yourself. This initializing process should logically be done in a constructor.

The Painter constructor in the upper part of Listing 9.1 creates a Painter object, where Painter is a subclass of the JFrame class. The `super` call invokes the JFrame constructor, supplying the title you want at the top of the frame. Then it calls the `addWindowListener` method, which is explained shortly. It calls three methods to set the size of the frame to a particular width and height in pixels (`setSize`), to cause the frame to appear on the monitor (`setVisible`), and to make the drawings that the `paint` method does using the Graphics object associated with the frame (`getGraphics` returns this Graphics object). Failure to specify size and visibility can be disconcerting: The default is an invisible frame 0 pixels wide and 0 pixels tall.

Listing 9.1 The Painter class of objects

```
import java.awt.Graphics2D;

public class Painter extends javax.swing.JFrame
{
   public Painter()
   {  super ("Wysiwyg Car Rentals");  // put the title on it
      addWindowListener (new Closer());
      setSize (760, 600);   // 760 pixels wide, 600 pixels tall
      setVisible (true);    // make it visible to the user
      paint (getGraphics());
   }  //====================

   /** Draw a pattern on the frame's drawing area. */

   public void paint (java.awt.Graphics g)
   {  Graphics2D page = (Graphics2D) g;
      page.drawString ("pattern", 10, 40);
      for (int depth = 40; depth < 580; depth += 5)
         page.draw (new java.awt.geom.Line2D.Double (depth,
                    depth, depth + 30, depth));  // horizontal
   }  //====================
}
//#################################################################


public class PainterApp
{
   /** Create and initialize the JFrame. */

   public static void main (String[ ] args)
   {  new Painter();
   }  //====================
}
```

If `jif` is a JFrame object, you may get the title of a JFrame with `jif.getTitle()`, and you may change the title with the method call `jif.setTitle(someString)`. You may find out the current width and height by `jif.getWidth()` and `jif.getHeight()`.

The **Component** class is a superclass of both JFrame and JApplet. The `getGraphics`, `setSize`, `getWidth`, `getHeight`, and `paint` methods are inherited from the Component class, so they are available for both JFrames and JApplets. You should explicitly call the `paint` method initially for JFrames; your browser calls it for JApplets.

The Painter class does not have a main method. That goes in a separate application class, shown in the lower part of Listing 9.1. It does not assign the constructed Painter object to a variable for later use because no other statement uses it later. If you run this program, you can see how a frame-based application works for drawing figures (the logic in this particular `paint` method is interesting but not important; you should totally ignore it if you did not read Chapter Eight). However, we will not use the `paint` method in the car-rental software we are developing.

The WindowListener Interface

If Listing 9.1 did not have the `addWindowListener` method call, you could not stop execution of the program without using Control-C in the terminal window. On some machines, the program locks up the computer and you must restart it. It will not let the user stop the program by clicking on the closer button (the X mark) in the upper-right corner of the frame. The window may in fact disappear, but the program still keeps running, which uses processor time and possibly locks up your computer. You need a window closer to tidy up.

Compile the Closer class of Listing 9.2 to define what a Closer object is. Now the `addWindowListener` method call makes arrangements for the program to stop when the user clicks the usual X in the upper right of the window -- that click sends a message to the Closer object to execute its `windowClosing` method. You should add a Closer object as the WindowListener for each JFrame object your programs have.

Listing 9.2 The Closer class

```
import java.awt.event.WindowListener;
import java.awt.event.WindowEvent;

public class Closer extends Object implements WindowListener
{
   /** Enable the closer icon to terminate the program. */

   public void windowClosing (WindowEvent e)
   {  System.exit (0);
   }  //=====================

   public void windowActivated (WindowEvent e)     { }
   public void windowDeactivated (WindowEvent e)   { }
   public void windowIconified (WindowEvent e)     { }
   public void windowDeiconified (WindowEvent e)   { }
   public void windowOpened (WindowEvent e)        { }
   public void windowClosed (WindowEvent e)        { }
}
```

Interfaces are discussed in detail in Chapter Eleven. For this chapter, all you need to know is that "implements" is very much like "extends" except that (a) the Closer class is called an **implementation** of WindowListener rather than a subclass, and (b) an implementation is required to override all of the methods in its interface.

The **WindowListener** interface and the **WindowEvent** class are in the `java.awt.event` package. The documentation for these two class definitions (depending on your Java version) is in the `jdk1.3\docs\api\java\awt\event` folder. You will see that, to be a WindowListener, a class must implement the seven methods specified in Listing 9.2 (iconifying a window means to minimize it, and activating a window means to give it the focus of keyboard events). However, for simple programs you only need to have the `windowClosing` method to exit the program; the other six can be given the do-nothing implementations shown.

Event-handling

The call of `addWindowListener` in the earlier Listing 9.1 attaches a new Closer object to the JFrame object. When the user clicks on the closer icon at the top-right corner of the JFrame (the big X), the JFrame object sends a `windowClosing` message to whatever WindowListener is attached to it, which would be this Closer object. That `windowClosing` method call executes `System.exit(0)` to terminate the program.

The JFrame object sends additional information to the `windowClosing` method in the form of a parameter of the **WindowEvent** class. This particular `windowClosing` logic does not need that object for anything. But you have to have the parameter anyway. Otherwise you are overloading rather than overriding the method in the WindowListener class. And that would mean that the JFrame object would not call your method.

Polymorphism

Figure 9.1 shows representations of the JFrame object and the Closer object. **Window** and WindowListener are in the standard Sun library, and JFrame is a subclass of Window. In effect, the Window class has an instance variable of type WindowListener named `wilis` (not its real name, but close). Your Closer object is assigned to `wilis`. This is legal because Closer implements WindowListener. Then the user's click executes a `wilis.windowClosing(x)` statement that is in a Window method.

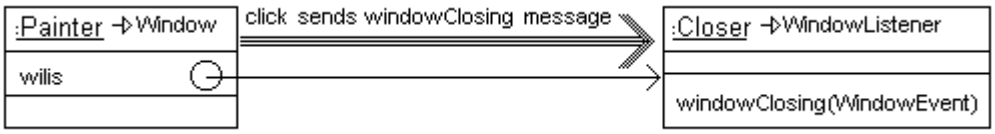


Figure 9.1 The runtime system sends a message to the attached WindowListener

Another program can define e.g. `class Shutter extends WindowListener` and add an object of that type to some JFrame, in which case executing the same `wilis.windowClosing(x)` statement in the Window method could be executing a Shutter method instead of a Closer method. So that statement in the Window method can execute any of several completely different methods (at different times, of course). This is polymorphism.

Exercise 9.1 Rewrite Listing 9.1 to use no import directives.
Exercise 9.2 Rewrite Listing 9.1 to set the lower-right corner of the window only about ten pixels below and to the right of the drawing it makes.
Exercise 9.3* Read the documentation for WindowListener and then explain the difference between `windowClosing` and `windowClosed`.
Exercise 9.4** Read the documentation for WindowAdapter and then explain the advantage of having a class inherit from it instead of WindowListener, as well as the disadvantage.

9.2 JPanels, Containers, And LayoutManagers

After you talk to the clients for the car rental agency and find out what they want, you decide to make the initial design for the software have the following features (Version 1). Each feature is represented by an object placed on the data-entry form. You will add more features in future versions during the iterative development, such as the time the car will be picked up and the time it will be returned:

- DaysRented: A textfield where the agents enter the number of days for the rental.
- BaseCost: A label giving the cost of a compact-car rental, calculated from the number of days rented and with a reduction for periods of a week or more.
- ClearButton: A button that clears all the entries on the form so a person can start over.
- SubmitButton: A button that writes the entries on the form to the database.
- CustomerName: A textfield for the name of the customer making the reservation.
- CreditCard: A textfield for the customer's 16-digit credit card number.

LayoutManagers

Components are added to Containers. Each Container object has an associated LayoutManager object which determines where components are added to the Container. The simplest LayoutManager is **java.awt.FlowLayout**, which just adds each component in book-reading order: The first component added goes at the top-left; each additional component added goes to the right of the one before, unless there is no room, in which case it goes at the far left of the next row. The components are centered in each row.

The content pane

You cannot add components directly to a JFrame or JApplet object; you must add them to its **content pane**. The `getContentPane()` method for JFrames and JApplets returns a **Container** object. That Container object has a layout manager, a BorderLayout which you must replace (until you study Section 9.12). The two key methods you will need from the `java.awt.Container` class are the following:

- `setLayout(LayoutManager)` replaces the layout manager with a new one.
- `add(Component)` adds the given Component as the next item in the list of Components attached to the Container executor. It returns the Component added.

JApplet is a subclass of Component. If you have an applet you want to be part of your frame, your frame has to do what a browser normally does for you: Create the applet, set its size the way you want it, and add it to the content frame. For instance, if Dancer is a JApplet (as it is in Listing 8.1), the following could be in the constructor of a JFrame:

```
Component degas = new Dancer();
degas.setSize (240, 120);
getContentPane().add (degas);
```

If the applet has an `init` method and/or a `start` method that a browser would execute automatically, your JFrame has to call those methods for the applet as well.

JPanels

The standard way to organize many components on a frame is to use panels. A **JPanel** object is an area to which you can attach various buttons, textfields, etc. You then place the JPanel on the content pane of a JFrame or JApplet. JPanel is a subclass of `javax.swing.JComponent`, which is a subclass of `java.awt.Container`, which is a subclass of `java.awt.Component`. A JPanel object has a FlowLayout. This car rental software uses the following two methods from the `javax.swing.JPanel` class, as well as the `add` method available for any Container object:

- `new JPanel()` creates a new JPanel object with a FlowLayout.
- `setBounds(int x, int y, int width, int height)` sets the top-left corner of the executor at `<x, y>`, with the given `width` and `height` measured in pixels. The executor must be a Component whose container has a `null` layout.

We will set the content pane's layout manager to `null` so that we can use the `setBounds` method to say just where we want each JPanel to appear.

The calculator of the base cost will be placed on the top panel on the frame, and the customer name and credit card will be entered on the second panel on the frame. Other components will be in three additional panels. Since it is by far best to keep the size of methods down, we will call five separate methods to obtain the five separate panels, e.g., `subViewOne()` returns a panel with textfields and labels already added.

The coding so far is in Listing 9.3 with all the import directives that will be needed. All parts of that coding have been explained in the preceding few pages. The reason that the name of the class is CarRentalView is that it provides the user's view of this software. The view has five subviews, represented by the five panels on the frame. The five methods that will be developed later are stubbed to make this coding compilable as is.

Listing 9.3 The CarRentalView class, some methods stubbed

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class CarRentalView extends JFrame
{
   public CarRentalView()
   {  super ("Wysiwyg Car Rentals");  // put the title on it
      this.addWindowListener (new Closer());  // enable closing
      this.setSize (760, 600);   // cover most of the screen
      this.init();
      this.setVisible (true);    // make it visible to the user
   }  //====================

   public void init()
   {  Container content = this.getContentPane();
      content.setLayout (null);
      content.add (subViewOne());
      content.add (subViewTwo());
      content.add (subViewThree());
      content.add (subViewFour());
      content.add (subViewFive());
   }  //====================

   private JPanel subViewOne()          { return new JPanel(); }
   private JPanel subViewTwo()          { return new JPanel(); }
   private JPanel subViewThree()        { return new JPanel(); }
   private JPanel subViewFour()         { return new JPanel(); }
   private JPanel subViewFive()         { return new JPanel(); }
}
//#################################################################


public class CarRentalApp
{
   /** Create and initialize the JFrame. */

   public static void main (String[ ] args)
   {  new CarRentalView();
   }  //====================
}
```

Everything that is needed for JFrames but not for JApplets is kept in the CarRentalView constructor, and everything else is done by the `init` method. So if you wanted to make this software into an applet instead, you would only need to change "JFrame" to "JApplet" in the class heading and comment out the CarRentalView constructor. Your browser will automatically create the JApplet, size it, make it visible, and call its `init` method. The same convertibility was maintained in the earlier Listing 9.1: If you change "JFrame" to "JApplet" and comment out the Painter constructor, you get an applet for which your browser will call the `paint` method.

The overall logic of the `subViewOne` method will be as follows. These statements create a panel, attach a few Components, and return the panel to be attached to the JFrame's content pane. That way, whatever you attach to the panel will be inside the frame. The second statement sets the size of the panel with a width that is slightly less than the entire frame, to allow for the border on the frame:

```
// addFirstPanel, statements 1-3
JPanel panel = new JPanel();
panel.setBounds (10, 25, this.getWidth() - 20, 40);
// statements attaching Components, e.g., panel.add(x);
return panel;
```

You are probably starting to become confused about the various standard library classes you have seen, particularly their methods and relationships. Figure 9.2 summarizes all that you have seen so far, except that some unneeded intermediate classes are left out (specifically, JFrame is a subclass of Frame which is a subclass of Window, and JApplet is a subclass of Applet which is a subclass of Panel).

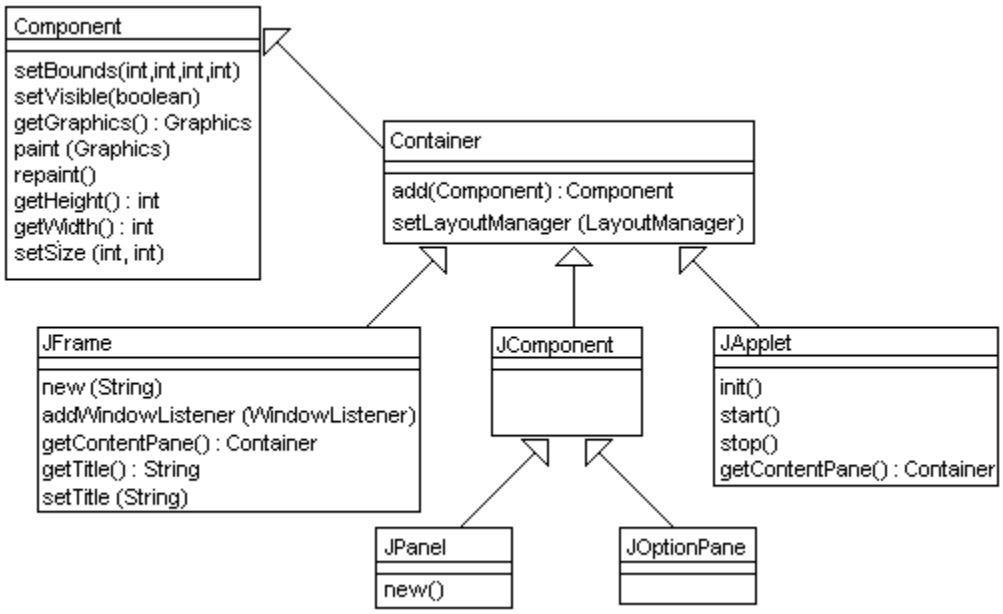


Figure 9.2 Hierarchy for some standard library classes

An optional section at the end of this chapter describes layout managers that are easier to use. For instance, if you replace the second statement of the `init` method in Listing 9.3 by the following, you can omit all statements in CarRentalView that set the bounds of a panel. The layout manager will then re-adjust the height and width of each panel whenever the user changes the shape of the frame:

```
content.setLayout (new BoxLayout (content, BoxLayout.Y_AXIS));
```

You could instead set the layout manager of the content pane as `new FlowLayout()`, which allows you to add many components in book-reading order. However, the `setSize` method for a JPanel you add using FlowLayout may not take effect.

Exercise 9.5* Read the documentation for LayoutManager and then list three kinds of layouts that implement the LayoutManager interface.
Exercise 9.6* Draw the UML class diagram for Listing 9.3.

9.3 JLabels, JTextFields, And ActionListeners

The **JLabel** class provides objects that simply display messages. The user cannot change the message, nor can the user get a reaction from a JLabel by clicking on it. The `javax.swing.JLabel` class is a subclass of JComponent, so you may use a JPanel's `add` method to add it to the panel. JLabel has the following useful methods:

- `new JLabel(String)` creates a JLabel with the given String as its text.
- `setText(String)` changes the text on the JLabel to the given String.
- `getText()` returns the String value that is the current text on that JLabel.

Quite often, a statement such as

```
panel.add (new JLabel ("Customer name:"));
```

does everything that has to be done with that particular label. If you do not expect to refer to the label later, it is superfluous to assign the newly created object to a variable. The label is therefore **anonymous**; it is simply displayed on the panel.

JTextFields

A **JTextField** object appears as a rectangular area that the user can fill in. You have surely seen JTextFields on web pages where you entered your name, address, or password. The `javax.swing.JTextField` class has the following methods:

- `new JTextField(String)` creates a JTextField with the given String as its text.
- `new JTextField(int)` creates a JTextField with the specified number of columns. For variable-width fonts, a column width is the pixel width of the letter 'm'.
- `addActionListener(ActionListener)` attaches a **listener** object to the JTextField. The listener object executes a method named `actionPerformed` when the ENTER key is pressed inside the JTextField.

A JTextField is a subclass of **JTextComponent**, which is a subclass of JComponent. The `javax.swing.text.JTextComponent` class has the following methods:

- `setText(String)` changes the text on the JTextField to the given String.
- `getText()` returns the String value that is the current text on that JTextField.

The subViewOne method

The first panel of the CarRental software is to contain a little calculator that computes the base cost of a car from the number of days it is rented. The formula that the client specifies is the number of days, minus one day for each whole week in the rental period, all multiplied by $30. The actual cost of a rental is the base cost for a compact and some multiple of the base cost for a nicer car. The user will type a number of days into a JTextField instance variable named `itsDaysRented` and press the ENTER key. Then the listener object that reacts to that input will make the corresponding change in the value stored on a JLabel instance variable named `itsBaseCost`.

Figure 9.3 shows how the panel will look on the display (except that the panel itself is invisible, so you will not see the larger rectangular outline on the display). The panel has a JLabel that says "Enter days rented:" and a JLabel that says "The base cost is:". These two labels are not given names in the program, because they are not referred to later in the execution. But the JTextField `itsDaysRented` is given a name when `subViewOne` creates it because the listener object later needs to retrieve its text. And

the JLabel `itsBaseCost` is given a name when `subViewOne` creates it because the listener object later needs to change what it says.

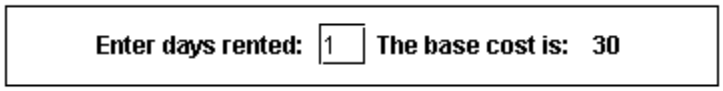


Figure 9.3 Objects on the first panel

Listing 9.4 has the coding required for the `subViewOne` method. The `panel.add` method calls attach the given Component to the panel in the order that they occur. The following statement should be the only puzzler in the `subViewOne` method:

```
itsDaysRented.addActionListener (new DaysRentedAL());
```

Listing 9.4 The subViewOne portion of CarRentalView

```
// CarRentalView class, part 2

   public static final double PRICE_PER_DAY = 30.00;
   ////////////////////////////////
   private JTextField itsDaysRented = new JTextField ("1");
   private JLabel itsBaseCost = new JLabel ("" + PRICE_PER_DAY);


   private JPanel subViewOne()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 25, this.getWidth() - 20, 40);

      panel.add (new JLabel ("Enter days rented:"));
      itsDaysRented.addActionListener (new DaysRentedAL());
      panel.add (this.itsDaysRented);
      panel.add (new JLabel ("The base cost is:"));
      panel.add (this.itsBaseCost);

      return panel;
   }  //=====================

   private class DaysRentedAL implements ActionListener
   {
      public void actionPerformed (ActionEvent e)
      {  int d = Integer.parseInt (itsDaysRented.getText());
         itsBaseCost.setText ("" + (d - d / 7) * PRICE_PER_DAY);
      }
   }  //=====================
```

ActionListeners

The basic listener logic for JTextFields is: `addActionListener` attaches a listener object to the JTextField object. The listener object "listens" for the user to press the ENTER key while the cursor is within the JTextField. When that happens, the listener object executes its `actionPerformed` method, performing whatever actions you have coded as an appropriate response to that keypress. This is precisely analogous to what happens when the user clicks the closer button for a JFrame and a WindowListener object executes its `windowClosing` method.

The parameter of the `addActionListener` method must be an object from a class that implements the **ActionListener interface**. In the lower part of Listing 9.4, it is the DaysRentedAL class ("AL" is for "action listener"; the next section explains the unexpected way in which DaysRentedAL is declared). The ActionListener interface specifies just one method, whose heading is as follows:

```
public void actionPerformed (ActionEvent e)
```

Event-driven programming

For **event-driven programming**, you **register an event-handler** using an `addActionListener` method or the equivalent. The runtime system waits until a user event or other action causes a JTextField or JButton or other Component to send a message to the listener object registered with it. This makes the runtime system execute the corresponding method, then go back to waiting. It will not react to a new event until the previous action is done. It can react only if it is in this "waiting state."

This is why `System.exit(0)` must be executed at the end of a program that uses graphic components. When a graphic component is created by the program, the runtime system goes into a mode where it remains in the waiting state whenever it has nothing to do. So nothing is happening with the program, but it is still running, waiting for listener actions. Even if you close all windows, the program keeps waiting. It takes a `System.exit` command to terminate the program's demand on system resources.

The JTextField object sends additional information through the `actionPerformed` method in the form of an object of the **ActionEvent** class. This particular `actionPerformed` logic does not need that object for anything. But you have to have the parameter anyway, because the interface requires it.

ActionListener is an interface in the `java.awt.event` package, just as is WindowListener. And ActionEvent is a class in the `java.awt.event` package, just as is WindowEvent. This is the reason for the import directive in the earlier Listing 9.2.

JPasswordFields

JPasswordField is a subclass of JTextField. The `javax.swing.JPasswordField` class has two key methods:

- `new JPasswordField(int)` yields a component that differs from JTextField in that, as the user types, an asterisk appears for each character instead of the character typed. This keeps other people from seeing the password being entered. The int parameter gives the width in characters (using an 'm' to measure it).
- `getPassword()` is like `getText()` except it returns a char array instead of a String, which minimizes security problems. Your coding can go through the characters in the array one at a time to check it against the true password and then erase those characters. If you store the password in a String object using e.g. `getText()`, it will not be erased from memory.

Exercise 9.7 What would be the effect of swapping the third and fifth statements in the `subViewOne` method of Listing 9.4?
Exercise 9.8 Revise Listing 9.4 to have another label at the end of the panel with the phrase "for a compact" on it.
Exercise 9.9 Revise the `actionPerformed` method in Listing 9.4 so it interprets the input value as a double temperature in degrees Fahrenheit and displays on the last label the corresponding temperature in degrees Centigrade.
Exercise 9.10* Read the documentation for JTextField, then write one statement that doubles the width of a JTextField object named `sam`.

9.4 Inner Classes And The Secondary Default Executor

You probably noticed the problem with the logic in Listing 9.4: How can a method in the DaysRentedAL class refer to an instance variable in the CarRentalView class? It is bad form to make instance variables public unless they are final variables. But only statements inside of a class can mention private instance variables in that class.

The solution is simple: Put the DaysRentedAL class inside of the CarRentalView class. Now it can mention those variables. Java allows you to define one class inside another, which makes it an **inner class**. The class it is inside of is called its **outer class**. An inner class can have instance variables and methods like the classes you are used to. It cannot contain class methods or class variables, since its heading does not include the `static` modifier. An inner class can mention any method or field variable in its outer class. Also, the outer class can mention any public method or field variable declared within an inner class (though this is not done in this chapter). The `private` modifier in the class heading prevents any outside class from mentioning DaysRentedAL.

The secondary default executor

The `new DaysRentalAL()` command is only allowed inside an instance method or constructor in the outer class. It creates a DaysRentalAL object that "knows" its creator, which is the executor (`this`) of that CarRentalView instance method. Inside a DaysRentalAL method, you can refer to that creator as **CarRentalView.this**. So the two mentions of CarRentalView instance variables inside the `actionPerformed` method could be written as follows:

```
CarRentalView.this.itsDaysRented.getText()
CarRentalView.this.itsBaseCost.setText (whatever)
```

However, the compiler allows you to omit the `CarRentalView.this` specification. As you know, the compiler assumes that any use of an instance variable or method that has no stated executor has `this` for its executor. But if `this` does not have such a variable or method, then the compiler assumes it has `Out.this` as the executor, where Out stands for the outer class. In other words, the implied executor (the default) is the inner class's object if applicable, otherwise it is the outer class's object. Since the DaysRentedAL class does not have field variables with the names `itsDaysRented` and `itsBaseCost`, the compiler correctly interprets the mentions of those variables as mentions of the CarRentalView object's instance variables. In short, `CarRentalView.this` is the secondary default executor.

In general, if X is an inner class of Out, `someOutObject.new X(whatever)` is a call of a constructor of class X. That is, `new` for an inner class requires an executor of class Out. When you use `new` alone inside an instance method or constructor of Out, you implicitly call `this.new X(whatever)`. In particular, the fourth statement of the `subViewOne` method could be written as follows:

```
itsDaysRented.addActionListener (this.new DaysRentedAL());
```

Every JTextField has an instance variable (say it is called `alis`) that refers to an ActionListener kind of object. When the ENTER key is pressed within the JTextField, it creates an event `e` and executes `alis.actionPerformed(e)`.

Figure 9.4 shows representations of some of the various objects involved in this program. A JFrame has in effect a Container instance variable (it might be named `contentPane`) that refers to its content pane. That pane has an instance variable (it might be named

comp) that is an array of components that have been added to the pane. One of those components is the first panel added to the content pane. That first panel also has an array of components that have been added, and the second one of them is the itsDaysRented JTextField (because the first one is an anonymous JLabel).

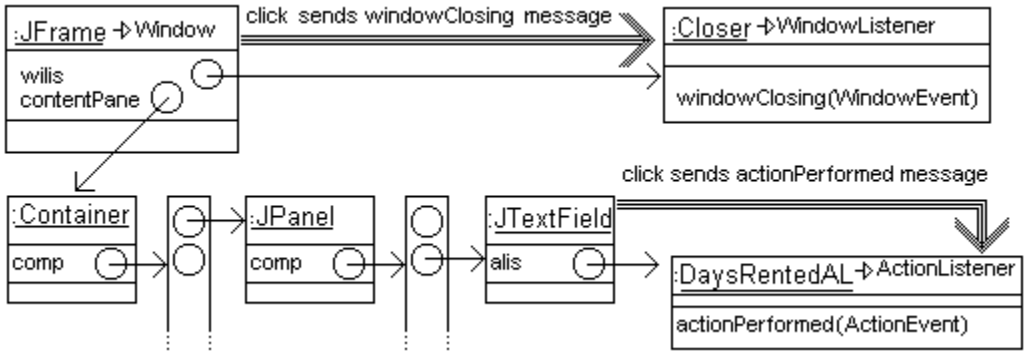


Figure 9.4 The runtime system sends messages to listeners

Language elements
One class can be declared inside of another class with heading: public class ClassName
or: private class ClassName
The inner class cannot have class variables or class methods, but it can have constructors.
Only an instance x of the outer class can call the constructor, e.g.: x.new ClassName (parameters).
Within an instance method of the inner class, Out.this refers to the instance of the outer class Out that called the constructor to create the instance of the inner class.

Exercise 9.11 What changes would be required in Listing 9.4 to have SubViewOne be a private inner class of CarRentalView and also a subclass of JPanel, so that its constructor would be called from the `init` method in Listing 9.3 by the statement `content.add (new SubViewOne())` but still have the same effect?
Exercise 9.12 Explain what changes would be needed to have the DaysRentedAL class outside of the CarRentalView class, not an inner class, and still have it do its job.
Exercise 9.13* Revise Listing 9.4 so that it has another label and textfield saying "Senior discount =" followed by "0". Each time the user clicks the ENTER key in either this new textfield or in `itsDaysRented`, the value in `itsBaseCost` is updated to be X dollars less than it is calculated to be in Listing 9.4, where X is the current dollar value in the new textfield.
Exercise 9.14* Read the documentation for JTextField and find the true name of the variable that stores the ActionListener object (it is not `alis`).

9.5 JButtons And EventObjects

A JButton is an object you can attach to any Container object. You usually attach an ActionListener object to a JButton (otherwise it is functionally equivalent to a JLabel). When the user clicks on the button, that sends a message to whichever ActionListener kind of object is attached to the button. That ActionListener kind of object executes its `actionPerformed` method, thus performing whatever actions you specify as the proper response to the button click.

The following methods are available for use with JButtons. **JButton** is a subclass of the **AbstractButton** class, from which it inherits all of these methods except the constructor. Both of these classes are in the `javax.swing` package:

- `new JButton(String)` constructs the JButton object and writes the given string of characters on the button.
- `getText()` retrieves the string of characters written on the button.
- `setText(String)` changes the string of characters written on the button. For instance, you can change the words on a button named `butn` to be "press me" with the statement `butn.setText("press me")`.
- `setMnemonic(char)` establishes a hotkey. So `butn.setMnemonic('c')` makes it so that a user who presses the letter 'c' while holding down the ALT key gets the same effect as clicking on the button named `butn`. The letter 'c' is underlined in the name on the button.
- `addActionListener(ActionListener)` is the same as for JTextField.

Example of using a button and an ActionEvent

By way of illustration, you could have the following class for the kind of listener object that displays a message each time a button is clicked:

```
public class Ear extends Object implements ActionListener
{
   public void actionPerformed (ActionEvent e)
   {  JOptionPane.showMessageDialog (null, "don't do that!");
   }  //======================
}
```

Now a method in a subclass of JFrame could have the following statements to create a JButton with the word "squawk" on it and attach the button to the frame's content pane. Clicking the button sends the `actionPerformed` message to the Ear that is attached to the JButton and thus displays the message "don't do that!" You could add the same Ear object to each of several buttons as their ActionListener, so that all would produce that same dialog box:

```
JButton butn = new JButton ("squawk");
butn.addActionListener (new Ear());
this.getContentPane().add (butn);
```

ActionEvent is a subclass of **EventObject**, which has a `getSource()` method that returns the Object value that produced the event. By way of illustration, if an Ear object is attached to several different JButtons, you could have it add an exclamation point to the text displayed on whatever button produced the event by putting the following two statements in the `actionPerformed` method of the Ear class. The returned value of `e.getSource()` has to be cast to the JButton subclass, since `getSource` returns a generic Object value:

```
JButton source = (JButton) e.getSource();
source.setText (source.getText() + "!");
```

A TicTacToe example

As one more illustration, a TicTacToe applet might have nine buttons, each initially with the word "open" on it. A player clicks an open button to change it to "X", and the computer opponent then clicks an open button to change it to "O". So the one `actionPerformed` method for all nine buttons could be as follows, assuming that the ActionListener implementation has access to an instance variable named `itsGame` which refers to an object with an appropriate `checkForGameOver` method:

```
public void actionPerformed (ActionEvent e)
{  JButton source = (JButton) e.getSource();
   if ( ! source.getText().equals ("open"))
      JOptionPane.showMessageDialog (null, "Already taken");
   else
   {  source.setText ("X");
      itsGame.checkForGameOver (source);
   }
}  //======================
```

The subViewTwo method

The `subViewTwo` method in the CarRentalView class has the following statements to create a button with the word "Clear" on it and attach an ActionListener instance of the ClearButtonAL class. Clicking this clear button sends the `actionPerformed` message to that listener. The button is added to a JPanel named `panel` which is attached to the content pane of the frame:

```
// statements in subViewTwo
JButton clearButton = new JButton ("Clear");
clearButton.setMnemonic ('c');
clearButton.addActionListener (new ClearButtonAL());
panel.add (clearButton);
```

The action that a ClearButtonAL is to perform when notified is to clear out the values written on the two JTextFields named `itsCustomerName` and `itsCreditCard`. So the following private inner class is put inside the CarRentalView class, to allow the `actionPerformed` method to access the two private instance variables:

```
private class ClearButtonAL implements ActionListener
{
   public void actionPerformed (ActionEvent e)
   {  CarRentalView.this.itsCustomerName.setText("");
      CarRentalView.this.itsCreditCard.setText("");
   }
}  //======================
```

A drawing of the objects and messages involved here would be almost the same as Figure 9.4: The second component of the content pane's array of components is the second panel. The first component of that panel's array of components is the clear button, which has a ClearButtonAL object to which an `actionPerformed` message is sent when the button is clicked.

The second panel also needs a submit button that the user clicks when the data recorded on the screen is to be stored in the customer database. The click will send an `actionPerformed` message to a SubmitButtonAL object.

A summary example

Listing 9.5 (see next page) illustrates an easy way to use a frame for a GUI implementation of a program. Remember that you may have any displayable object in place of the String value for showMessageDialog.

This program implements a queue of characters -- you may add characters to the rear or take them off the front. Specifically, when the user presses ENTER in the text field, that adds characters from the text field to the rear of the data; and when the user clicks the button, that removes the first character from the front of the data. The text area has scrollbars; it displays the current state of the data after each change. The user clicks the OK button to make the frame go away and thus terminate the program.

Listing 9.5 A complete program illustrating basic components

```
import javax.swing.*;
import java.awt.event.*;

public class Gooey extends JPanel
{
   private JTextField itsInput = new JTextField (15);
   private JTextArea itsOutput;
   private String itsData = "";


   public Gooey (int rows, int columns)
   {  this.add (new JLabel ("entry:"));
      itsInput.addActionListener (new TextFieldAL());
      this.add (itsInput);

      JButton button = new JButton ("delete first character");
      button.addActionListener (new ButtonAL());
      this.add (button);

      itsOutput = new JTextArea (rows, columns);
      this.add (new JScrollPane (itsOutput));
   }  //======================

   private class TextFieldAL implements ActionListener
   {  public void actionPerformed (ActionEvent e)
      {  itsData += itsInput.getText();
         itsInput.setText ("");
         itsOutput.append (itsData + "\n");
      }
   }  //======================

   private class ButtonAL implements ActionListener
   {  public void actionPerformed (ActionEvent e)
      {  if ( ! itsData.equals (""))
            itsData = itsData.substring (1);  // delete the first
         itsOutput.append (itsData + "\n");
      }
   }  //======================
}
//#################################################################

public class GooeyApp
{
   public static void main (String[] args)
   {  JOptionPane.showMessageDialog (null, new Gooey (8, 20));
      System.exit (0);
   }  //======================
}
```

Exercise 9.15 Revise the `actionPerformed` method in the original Ear class to change the words on the button that was clicked to "Ouch!".
Exercise 9.16 Revise the `actionPerformed` method for TicTacToe to print two different error messages depending on which of the two illegal choices the user made, a button with "X" on it or a button with "O" on it.
Exercise 9.17* Revise the `actionPerformed` method in the ClearButtonAL class so that it asks the user "Are you sure?" using a JOptionPane method and does not make any change if the user does not confirm it.

9.6 The Model/View/Controller Pattern

The CarRental software is an example of the Model/Controller/View pattern, which is illustrated at the end of Chapter Six for the RepairShop software. Specifically, the CarRentalView subclass of JFrame displays the current values on the monitor for the user to see, i.e., presents the View into the program. The View has five subViews, represented by the five panels added to the content pane of the frame. Another part of the software maintains the customer database but normally does no input or output; it is the data Model for this software. The third part of the software is the Controller; it accepts input and sends messages to the Model and the View to update themselves accordingly. With the two panels added so far, we have three controller objects: DaysRentedAL, ClearButtonAL, and SubmitButtonAL.

The preceding Listing 9.5 provides another example of the Model/View/Controller pattern: `itsData` is the model, the two `actionPerformed` methods are for the controller objects, and the rest of the Gooey class is the view.

We need some kind of object in which to store information about customers. This object is the Model part of the Model/View/Controller pattern. For the initial versions of the software, we will have a prototype CarRentalModel class, barely adequate to let us test the user interface. The following class will suffice for now:

```
public class CarRentalModel extends Object
{  public void add (String given)
   {  System.out.println ("Model got " + given);
   }  //======================
}
```

The completed `subViewTwo` method is in Listing 9.6 (see next page), along with the relevant instance variables and ActionListener implementations (the first illustrates how the secondary default executor is explicitly referenced). Note that it is quite legal to declare instance variables wherever you want in the class. There is a lot to be said for declaring them only immediately before the methods that use them when you have a class of many pages. Figure 9.5 shows what the display looks like during execution of this program so far (Version 1 in the iterative development).

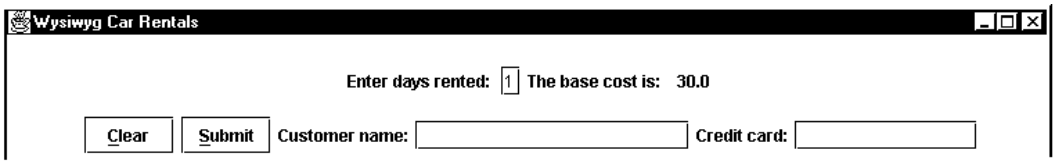


Figure 9.5 Screen shot of execution of CarRentalApp, Version 1

Listing 9.6 The subViewTwo portion of the CarRentalView class

```
// CarRentalView class, part 3

   private CarRentalModel itsModel = new CarRentalModel();
   private JTextField itsCustomerName = new JTextField (18);
   private JTextField itsCreditCard = new JTextField (12);


   private JPanel subViewTwo()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 75, this.getWidth() - 20, 40);

      JButton clearButton = new JButton ("Clear");
      clearButton.setMnemonic ('c');
      clearButton.addActionListener (new ClearButtonAL());
      panel.add (clearButton);

      JButton submitButton = new JButton ("Submit");
      submitButton.setMnemonic ('s');
      submitButton.addActionListener (new SubmitButtonAL());
      panel.add (submitButton);

      panel.add (new JLabel ("Customer name:"));
      panel.add (this.itsCustomerName);
      panel.add (new JLabel ("Credit card:"));
      panel.add (this.itsCreditCard);

      return panel;
   }  //=====================

   private class ClearButtonAL implements ActionListener
   {
      public void actionPerformed (ActionEvent e)
      {  CarRentalView.this.itsCustomerName.setText("");
         CarRentalView.this.itsCreditCard.setText("");
      }
   }  //=====================

   private class SubmitButtonAL implements ActionListener
   {
      public void actionPerformed (ActionEvent e)
      {  itsModel.add (itsCustomerName.getText() + "  "
                     + itsCreditCard.getText());
      }
   }  //=====================
```

The JComponent class

The `setToolTipText(String)` method call sets the text that will appear when the mouse cursor pauses over a component. This method is in the JComponent class from the `javax.swing` package. JButtons, JTextFields, and JLabels are all subclasses of the **JComponent** class, a subclass of the Container class. So you can add a tool-tip to any of those swing components. For instance, you could put the following two statements in the `subViewTwo` method of Listing 9.6, one after each JButton constructor call:

```
   clearButton.setToolTipText ("clear all entries on the form");
   submitButton.setToolTipText ("store all values in the file");
```

Figure 9.6 shows the relationships of the additional subclasses of JComponent discussed in the past few sections, plus the JSlider class to be discussed in the next section. The figure includes all of the methods used in this book for these classes. There are many more swing classes in the standard library; JComponent alone has more direct subclasses than Bach had children (Maria gave him seven and Anna gave him thirteen). The constructors are not mentioned in this figure for the AbstractButton and JTextComponent classes. That is because they are abstract classes (described in Chapter Eleven) and so you cannot create objects of those classes.

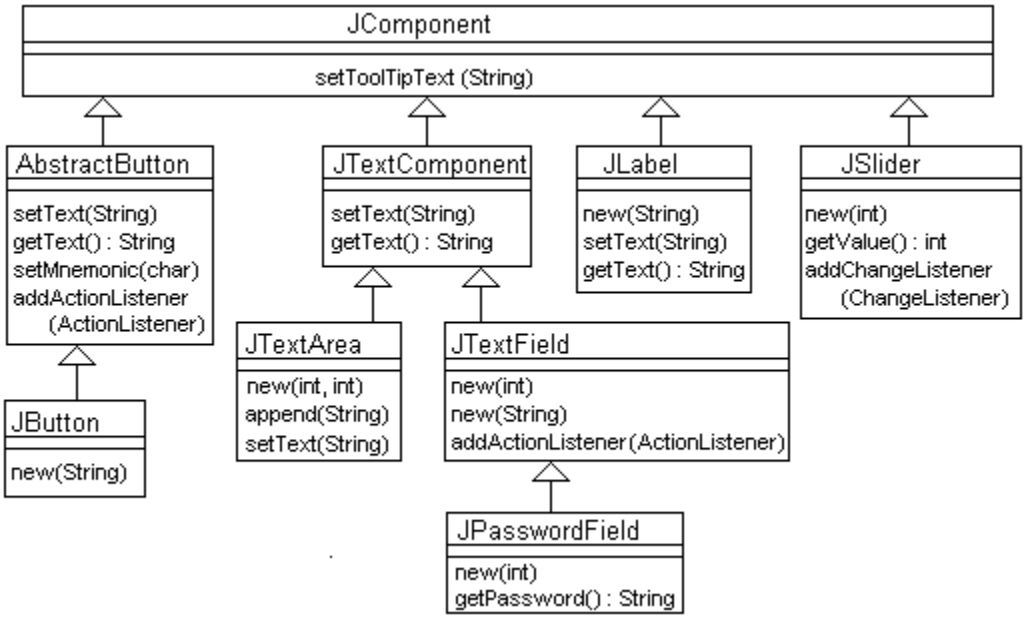


Figure 9.6 UML class diagram of subclasses of JComponent so far

Anonymous classes

Java allows great flexibility in making listener objects. It may seem natural to have each button listen for its own click, but there are times when you want to have a single ActionListener object for several different buttons or other objects.

If you want an ActionListener object to only listen for one Component, you may write the specification for an action listener more compactly as an **anonymous class**. That is a class without a name, for which you only supply the body without constructors. For instance, you may omit the ClearButtonAL class completely from Listing 9.6 if you replace the fifth statement of the `subViewTwo` method by the following:

```
clearButton.addActionListener (new ActionListener()
      {  public void actionPerformed (ActionEvent e)
         {  itsCustomerName.setText ("");
            itsCreditCard.setText ("");
         }
      }
);  // this parenthesis matches the one before "new"
```

All this does is create a single object of the anonymous class, which is an implementor of the ActionListener class and has the `actionPerformed` method specified. You can use this kind of construction, where the body of anonymous class X immediately follows the parentheses of a constructor call `new Whatever()`, wherever you need just one new instance of class X and (a) X is a subclass of class Whatever, or (b) X implements interface Whatever.

An anonymous class is an inner class. In general, it is best to avoid anonymous classes unless they are extremely short, rarely over four or five statements in them. This book does not use anonymous classes elsewhere.

Exercise 9.18 Revise the second `actionPerformed` method in Listing 9.6 so that it prints an error message if either the customer name or the credit card is the empty string.
Exercise 9.19 Where would the textfield for the credit card numbers appear if the panel were not wide enough to have it after the other components?
Exercise 9.20 What changes would be needed in Listing 9.6 to have the two buttons moved to the right of the credit card number?
Exercise 9.21* Revise Listing 9.6 to have two textfields for the customer's name (first name and last name) instead of just one.
Exercise 9.22* Rewrite Listing 9.6 to add either "this" or "CarRentalView.this" wherever possible.
Exercise 9.23* How would you rewrite Listing 9.6 to have SubmitButtonAL be an anonymous class?
Exercise 9.24* Read the documentation for JComponent, then describe three advantages of the new swing components over the original ones (Button, TextField, etc.).
Exercise 9.25** Draw the UML diagram for the entire CarRentalView program so far.

9.7 JSliders And ChangeListeners

When you have several lines of output, a JTextField or JLabel does not work well to display the output. A JTextArea is better, since it is a rectangular area on the drawing surface. The `javax.swing.JTextArea` class is a subclass of JTextComponent, so you can use the `setToolTipText`, `setText`, and `getText` methods with JTextAreas. The methods you saw in Chapter Six are all you really need for JTextAreas:

- `new JTextArea(int, int)` creates a JTextArea with the given number of rows (first parameter) and columns (second parameter) of characters. The width is calculated from the width of the letter 'm'.
- `setText(String)` puts that string value in the JTextArea, replacing whatever was already there. In particular, `answer.setText("")` erases everything in the JTextArea.
- `append(String)` adds some more information to whatever is already in the JTextArea. To start a new line with the information you add, use the command `answers.append('\n' + someString)`, since `'\n'` is the new-line character.

A standard initial sequence with JTextAreas is the following, unless you want to add a JScrollPane as described in Chapter Six. You do not attach a listener object because a JTextArea does not react to any events:

```
JTextArea answers = new JTextArea (50, 10);
someContainer.add (answers);
```

JSliders

A **JSlider** is a bar with a **thumb** icon that moves left and right if it runs horizontally, or up and down if it runs vertically. The bar normally has a numerical scale running from 0 to 100. The `javax.swing.JSlider` class is a subclass of the JComponent class, so you can use the `setToolTipText` methods with JSliders, as well as the following:

- `new JSlider(int)` creates a JSlider object that runs either horizontally or vertically, depending on the value of the parameter. You should use either JSlider.HORIZONTAL or JSlider.VERTICAL for that parameter.
- `getValue()` returns the int value the thumb currently indicates.
- `addChangeListener(ChangeListener)` attaches a listener object to the JSlider object, which will react to any move-and-release action on the thumb.
- `setMinimum(int)` specifies the smallest value that appears.
- `setMaximum(int)` specifies the largest value that appears.
- `setMinorTickSpacing(int)` specifies the distance between tick marks. For instance, if it is 4 and the minimum is set to 10, you will have tick marks on the slider at 10, 14, 18, 22, 26, etc.

You attach a **ChangeListener** object to the JSlider to react to events with the `addChangeListener` method. When the user clicks-and-drags the thumb and then lets go, the JSlider sends a message to that ChangeListener object to execute its `stateChanged` method. ChangeListener is an interface.

This is exactly the same as what happens with a click of a JButton except that the **EventListener** used is a ChangeListener rather than an ActionListener, and the method it executes is named `stateChanged` rather than `actionPerformed`, and the parameter is a **ChangeEvent** instead of an ActionEvent. These are both subclasses of EventObject, which has the `getSource()` method to see which component was activated. Figure 9.7 show the EventListener classes and EventObject classes discussed so far. ChangeListener and ChangeEvent are in the `javax.swing.event` package.

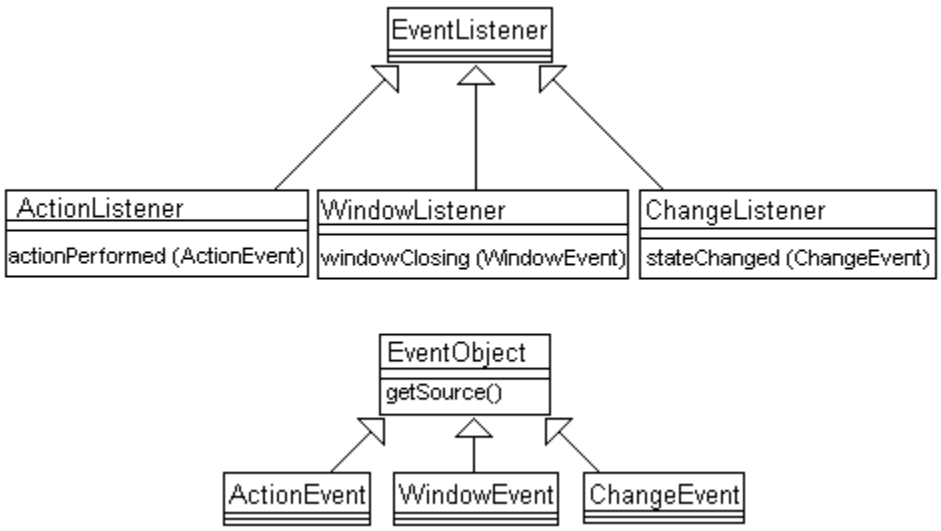


Figure 9.7 UML class diagram of listener interfaces and event classes

The constants `JSlider.HORIZONTAL` and `JSlider.VERTICAL` that indicate which way a JSlider runs are actually defined in the **SwingConstants** interface rather than the JSlider class. But the JSlider class and many others implement SwingConstants, which gives a common set of constants to all of those swing classes. This is another good use of an interface. Remember that an interface is allowed to specify class variables as well as instance method headings.

Listing 9.7 is an applet to illustrate the use of a JSlider and a JTextArea. Each time the user releases the thumb at some point on the slider, the corresponding value is added to a running total and displayed in a textarea.

Listing 9.7 The AddInputs applet

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.Container;
import java.awt.event.*;

public class AddInputs extends JApplet
{
   private JSlider itsInput = new JSlider (JSlider.HORIZONTAL);
   private JTextArea itsOutput = new JTextArea (40, 20);
   private int itsData = 0;


   public void init()
   {  setVisible (true);
      Container content = this.getContentPane();
      content.add (itsOutput);

      itsInput.addChangeListener (new SliderCL());
      itsInput.setToolTipText ("numbers ranging 0 to 100");
      content.add (itsInput);

      JButton show = new JButton ("show total so far");
      show.addActionListener (new ButtonAL());
      content.add (show);
   }  //=====================

   private class ButtonAL implements ActionListener
   {
      public void actionPerformed (ActionEvent e)
      {  itsOutput.append ("\nTotal so far is " + itsData);
      }
   }  //=====================

   private class SliderCL implements ChangeListener
   {
      public void stateChanged (ChangeEvent e)
      {  itsOutput.append ("\nEntry: " + itsInput.getValue());
         itsData += itsInput.getValue();
      }
   }  //=====================
}
```

The `start` and `paint` methods are omitted because the superclass JApplet provides do-nothing implementations of them except that `paint` shows all of the components on the screen. The `init` method is executed automatically by the browser when it loads the applet. If you want to run this within a JFrame application, simply add a new AddInputs object to the JFrame's content pane and call that AddInput object's `init` method.

When you browse a web page containing this applet, it creates an AddInputs object which creates the text area, the slider, and the button and puts them on the screen. It also attaches a listener object to each of the slider and the button. Then the runtime system waits for some action:

- If the user moves the slider to the value 5 (the values range from 0 to 100), the slider sends a `stateChanged` message to whomever is attached to the slider. That message tells the listener to print "Entry: 5" on the text area and add 5 to `itsData` (which is now 5).
- If the user then moves the slider to 7, the slider sends a `stateChanged` message to whomever is attached to the slider, who prints a new line "Entry: 7" on the text area and adds 7 to `itsData` (which is now 12).
- If the user then clicks the button that says "show total so far", the button sends an `actionPerformed` message to whomever is attached to the button (the other listener object). That message tells the listener to append a third line "Total so far is 12" to the text area.

Figure 9.8 shows most of the relationships among the applet, content pane, button, slider, and listener objects created by Listing 9.7.

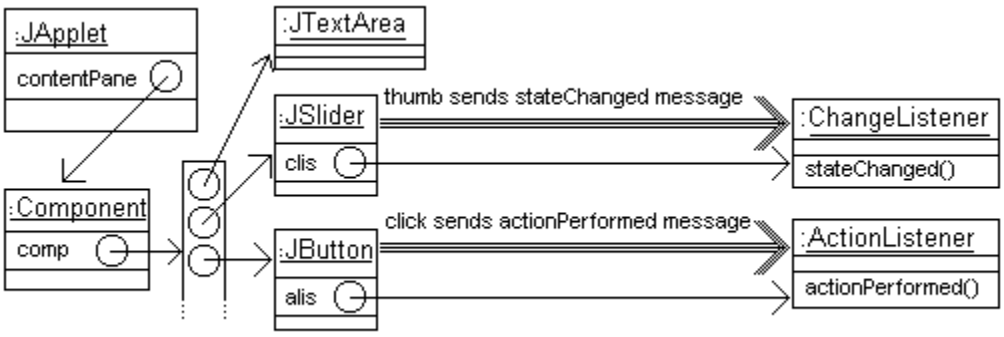


Figure 9.8 Some Component objects for Listing 9.7

A self-listener object

The kind of structure we are using for the CarRental software is not the only good one. An alternative way of organizing Components is to have a "self-listener" object for each Component-plus-listener object you want to add. That way, everything that has to do with the one component is in one location. For instance, you could replace the three statements mentioning `itsInput` in the `init` method of Listing 9.7 by the following one statement, as well as eliminate the declaration of `itsInput` itself:

```
content.add (new SliderCL());
```

All you need instead is the following larger definition of the SliderCL class, in which the JSlider is its own listener. Compare it element-by-element with Listing 9.7:

```
private class SliderCL extends JSlider
                       implements ChangeListener
{  public SliderCL()
   {  super (JSlider.HORIZONTAL);
      this.addChangeListener (this);
      this.setToolTipText ("numbers ranging 0 to 100");
   }

   public void stateChanged (ChangeEvent e)
   {  itsOutput.append ("\nEntry: " + this.getValue());
      itsData += this.getValue();
   }
}  //======================
```

Exercise 9.26 Rewrite Listing 9.7 to not have an instance variable storing the JSlider.
Exercise 9.27 Modify Listing 9.7 to have a second slider. The values from the second one are subtracted from the running total, as if the two sliders indicated deposits and checks in a bank account.
Exercise 9.28* Rewrite Listing 9.7 to use an anonymous class in place of SliderCL.
Exercise 9.29* How would you rewrite Listing 9.7 to make ButtonAL a self-listener?
Exercise 9.30* Rewrite Listing 9.4 to use a self-listener for the JTextField.
Exercise 9.31* Read the documentation for JSliders. Then say what the initial value of the slider is and explain how to specify it at the time the slider is constructed.
Exercise 9.32* Draw the UML diagram for the entire CarRentalView program so far.

9.8 Swing Timers

The CarRental software clients want a timer to click off the seconds that pass while an employee is entering data. The second JPanel object still has some space on it. We will declare a subclass of JLabel named Clock and put the instance variable declaration

```
private Clock itsClock = new Clock (1000);
```

in the CarRentalView class in the earlier Listing 9.6. Then all we need do is add the following statements near the end of the `subViewTwo` method of that class:

```
panel.add (itsClock);
itsClock.start();
```

The Timer class

The **Timer** class is in the `javax.swing` package. It has the following useful methods:

- `new Timer(int, ActionListener)` creates a new Timer object that, when it is going, sends the `actionPerformed` message to the specified ActionListener parameter every `n` milliseconds, where `n` is given by the first parameter.
- `start()` has the Timer object start sending notifications to its ActionListener.
- `stop()` has the Timer object stop sending notifications to its ActionListener.
- `setRepeats(boolean)` if set to `false` has the Timer object only send one notification and then automatically stop; `true` gives multiple notifications.

For the CarRentalApp program, the Clock object should create its own private Timer object with a delay of 1000 milliseconds between notifications (i.e., one second). The text on the JLabel is the number of seconds that have passed, so the Clock object also needs an instance variable `itsCounter` to keep a record of the time that has passed.

The Clock object can be started by the statement `itsClock.start()` in the `subViewOne` method. That starts the Timer with `itsCounter` equal to zero. Each time the Timer notifies its ActionListener object that one time period has passed, the listener increments the clock's `itsCounter` by the number of milliseconds per time period and changes its text to be that number of seconds. The Clock logic is in Listing 9.8 (see next page).

The Clock class is a separately-compiled public class rather than a private inner class because (a) it can be useful in several other programs, and (b) it has no need to access the private members of the CarRentalView class.

Listing 9.8 The Clock class

```
import javax.swing.*;
import java.awt.event.*;

public class Clock extends JLabel
{
   private Timer itsTimer;
   private int itsMillis;  // milliseconds per activation
   private int itsCounter; // milliseconds since it was started


   public Clock (int millis)
   {  super ("0");
      itsMillis = millis;
      itsTimer = new Timer (millis, new TimerAL());
   }  //======================

   public void start()
   {  itsCounter = 0;
      itsTimer.start();
   }  //======================

   public void stop()
   {  itsTimer.stop();
   }  //======================

   private class TimerAL implements ActionListener
   {
      public void actionPerformed (ActionEvent e)
      {  Clock.this.itsCounter += itsMillis;
         Clock.this.setText("" + (Clock.this.itsCounter / 1000));
      }
   }  //======================
}
```

Restarting the clock

This design is unsatisfactory. The clock keeps ticking away, eventually recording tens of thousands of seconds as the day wears on. The clients say that the purpose of the clock is to track how long it takes the employee to make a single data entry. So the timer should restart at zero each time the employee switches to a new customer, i.e., when the clear button is clicked. And the clock should stop when the data is stored in the file, i.e., when the submit button is clicked.

Therefore, add the following statement to the `actionPerformed` method in the ClearButtonAL class of the earlier Listing 9.6:

```
itsClock.start();
```

Also add the following statement to the `actionPerformed` method in the SubmitButtonAL class in Listing 9.6:

```
itsClock.stop();
```

Timers versus busywaits

The dancing flag software in Listing 8.7 has a busywait logic, in which the CPU is kept uselessly busy doing nothing until 50 milliseconds have passed. It could be done much more nicely with a Timer. To illustrate the technique, we will apply it in a simpler case, inserting 0.1 second pauses after each horizontal line drawn by the `paint` method in Listing 9.1. We leave Listing 8.7 as a (hard) exercise.

Listing 9.9 breaks up the execution of the loop in Painter's `paint` method into multiple executions of the `actionPerformed` method of a Timer's listener, once each 100 milliseconds. Each tick of the Timer execute one iteration of the loop, and an instance variable `itsDepth` keeps track of which iteration it is on (in place of the local variable `depth`). We have to be sure to stop the Timer after the last iteration. Compare the logic in Listing 9.9 closely with the original Painter class in Listing 9.1.

Listing 9.9 Time-delayed Painter class

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Painter extends JFrame
{
   private Timer itsTimer = new Timer (100, new TimerAL());


   public Painter()
   {  super ("Wysiwyg Car Rentals");
      addWindowListener (new Closer());
      setSize (760, 600);   // 760 pixels wide, 600 pixels tall
      setVisible (true);
      paint (getGraphics());
   }  //====================

   public void paint (Graphics page)
   {  page.drawString ("pattern", 10, 40);
      itsTimer.start();
   }  //======================

   private class TimerAL implements ActionListener
   {
      private int depth = 40;

      public void actionPerformed (ActionEvent e)
      {  Graphics page = getGraphics();
         int wide = (600 - depth) * (600 - depth) / 600;
         page.drawLine (depth, depth, depth + wide, depth);
         depth += 5;
         if (depth >= 580)
            itsTimer.stop();
      }
   }  //======================
}
```

Exercise 9.33 Revise Listing 9.8 so that it shows the elapsed time in tenths of a second instead of in whole seconds.
Exercise 9.34 Rewrite Listing 9.8 to not use an instance variable to keep track of the number of seconds, but still do what it originally did.

Exercise 9.35* Revise Listing 9.8 so it displays minutes and seconds, in the form illustrated by 11:46.
Exercise 9.36** Revise Listing 8.7 to use a Timer rather than a busywait. You will need `itsCount` as an instance variable, initialized to zero. Stop the clock when `itsCount` becomes 5. Hint: Use a boolean variable `itsDancingRight`, initialized to `true`, that keeps track of whether you are dancing to the right or to the left.

9.9 JComboBoxes Using Arrays Of Objects

The CarRental clients want to have three combo boxes where the employee chooses the month, the day, and the hour when the car will be picked up. You need three more similar boxes for returning the car. A combo box shows all the possible choices that the employee has in a pull-down menu with the one currently selected highlighted. If the pull-down menu is not down, the user sees only the one choice that is currently selected.

JComboBox is a subclass of JComponent. The `javax.swing.JComboBox` has the following methods (and many more besides):

- `new JComboBox(Object[])` creates a new JComboBox with the values in the array as its choices. If the array components are Strings, the words will appear as the choices, in the same order as they are stored in the array, indexed from zero on up.
- `addActionListener(ActionListener)` says what listener object to notify if a selection is made.
- `getSelectedIndex()` returns the int value that is the index of the currently selected Object.
- `setSelectedIndex(int)` selects the Object with the specified index.
- `getSelectedItem()` returns the Object that is currently selected.
- `setMaximumRowCount(int)` says how many items will be visible at a time. A scroll bar appears if this is less than the total number of items.
- `addItem(Object)` adds this one more Object to the end of the list of choices.
- `removeItemAt(int)` deletes the Object at that index from the list of choices.

The CarRental software has to have available an array of 12 Strings for the 12 months of the year. It also needs an array of Strings for the 31 days "01" through "31" and an array of Strings for the hours in the day at which people can rent a car. The client says that the permissible times are from 7 A.M. through 5 P.M., which makes 11 array entries. These class variables are declared in Listing 9.10 with array initializer lists. Figure 9.9 shows how the two new panels should look.

Listing 9.10 Arrays for ComboBoxes in the CarRentalView class

```
// CarRentalView class, part 4

   private static String[] months = {"January", "February",
          "March", "April", "May", "June", "July", "August",
          "September", "October", "November", "December"};
   private static String[] days = {"01", "02", "03", "04", "05",
          "06", "07", "08", "09", "10", "11", "12", "13", "14",
          "15", "16", "17", "18", "19", "20", "21", "22", "23",
          "24", "25", "26", "27", "28", "29", "30", "31"};
   private static String[] hours = {"7am", "8am", "9am", "10am",
          "11am", "noon", "1pm", "2pm", "3pm", "4pm", "5pm"};
```

Figure 9.9 Third and fourth panels of the CarRentalView frame

You can have instance variables named `itsStartMonth` and `itsEndMonth` to refer to the JComboBoxes that list the twelve months, declared as follows:

```
private JComboBox itsStartMonth = new JComboBox (months);
private JComboBox itsEndMonth = new JComboBox (months);
```

It is convenient for the employee if, after choosing `itsStartMonth`, the value of `itsEndMonth` automatically changes to the same month. The same should be done for the hours `itsStartHour` and `itsEndHour`. More often than not, the ending month and hour are the same as the starting month and hour, though the day is usually different, so there is no point in having the ending day change with the starting day.

This means that you need to attach an ActionListener object to `itsStartMonth` so that any change to its selected index changes the selected index of `itsEndMonth` to match. The `actionPerformed` method would therefore be coded as follows:

```
public void actionPerformed (ActionEvent e)
{  itsEndMonth.setSelectedIndex
             (itsStartMonth.getSelectedIndex());
}
```

Similarly, you need to attach an ActionListener object to `itsStartHour` so it can update `itsEndHour` when needed. Listing 9.11 (see next page) shows the resulting additions to the CarRentalView class. Read through it to be sure you understand it all.

Drawing on a JComponent

You should not draw directly on a JApplet or JFrame if you are planning to add any components whatsoever to it. You instead draw on a JPanel that is attached to its content pane or attached to some other component. The following statement calls the standard method to refresh a drawing on a given panel object:

```
panel.paintComponent (panel.getGraphics());
```

Each JComponent has a `getGraphics` and a `paintComponent` method for this use. The `paintComponent` method is called instead of the `paint` method of a JApplet, but it does much the same thing. There is one caveat however: Be sure to have its first statement call the `paintComponent` method of the superclass. Otherwise you will not like what you see. So a sample `paintComponent` method is the following:

```
public void paintComponent (Graphics g)
{  super.paintComponent (g);
   Graphics2D page = (Graphics2D) g;
   page.draw (new Line2D.Double (20, 30, 100, 200));
   // additional drawing commands
}  //======================
```

Listing 9.11 subViewThree and subViewFour for the CarRentalView class

```
// CarRentalView class, part 5

   private JComboBox itsStartMonth = new JComboBox (months);
   private JComboBox itsStartDay   = new JComboBox (days);
   private JComboBox itsStartHour  = new JComboBox (hours);
   private JComboBox itsEndMonth   = new JComboBox (months);
   private JComboBox itsEndDay     = new JComboBox (days);
   private JComboBox itsEndHour    = new JComboBox (hours);


   public JPanel subViewThree()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 125, this.getWidth() - 20, 40);
      panel.add (new Label ("starting month/day/hour:"));
      itsStartMonth.addActionListener (new StartMonthAL());
      panel.add (itsStartMonth);
      panel.add (itsStartDay);
      itsStartHour.addActionListener (new StartHourAL());
      panel.add (itsStartHour);
      return panel;
   }  //====================

   public JPanel subViewFour()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 175, this.getWidth() - 20, 40);
      panel.add (new Label ("ending month/day/hour:"));
      panel.add (itsEndMonth);
      panel.add (itsEndDay);
      panel.add (itsEndHour);
      return panel;
   }  //====================

   private class StartMonthAL implements ActionListener
   {  public void actionPerformed (ActionEvent e)
      {  itsEndMonth.setSelectedIndex
                   (itsStartMonth.getSelectedIndex());
      }
   }  //====================

   private class StartHourAL implements ActionListener
   {  public void actionPerformed (ActionEvent e)
      {  itsEndHour.setSelectedIndex
                  (itsStartHour.getSelectedIndex());
      }
   }  //====================
```

Exercise 9.37 Write a private method that could go in the CarRentalView class and be called by `panel.add (boxOfNumbers (60))` to add a JComboBox that lists the 60 different values for the minutes part of the time ("0" through "59"). Create the array using a for-statement, not an initializer list.

9.10 JCheckBoxes, JRadioButtons, ButtonGroups, And ItemListeners

Another piece of information that the CarRental software must record is the category of vehicle that the customer wants to rent. The four choices available are compact, full-size, luxury, and sport utility. The `subViewFive` panel is to be used for this choice. Moreover, the customer is to indicate whether he/she wants manual shift, air conditioning, and/or a CD player.

JCheckBoxes

The **JCheckBox** class is a subclass of **JToggleButton**, which describes buttons with two states, "selected" and "unselected". JToggleButton is a subclass of AbstractButton, which is also the superclass of JButton. The following methods are quite useful for `javax.swing.JCheckBox`, in addition to the `getText`, `setText`, `setMnemonic`, and `addActionListener` methods inherited from the AbstractButton class:

- `new JCheckBox(String, boolean)` creates a new button with the specified text on it. The button is selected if the second parameter is `true`, unselected if `false`.
- `isSelected()` returns `true` if and only if the button is currently selected.
- `setSelected(boolean)` makes the button selected if the parameter is `true` and unselected otherwise.

The CarRental software is to have three JCheckBox buttons on the fifth panel, one for each of the options of "manual", "air cond", "CD player". So you could add the following three instance variables to the CarRentalView class, initialized to the most commonly chosen values:

```
JCheckBox itsManual = new JCheckBox ("manual", false);
JCheckBox itsAir = new JCheckBox ("air cond", true);
JCheckBox itsCD = new JCheckBox ("CD player", true);
```

You also need statements to attach these buttons to the fifth panel. This software does not need an `actionPerformed` method for the buttons, because no particular action is required at the time one of these buttons is selected. But you do need to keep track of the button itself so that, when the submit button is clicked to store information in the data file, its `actionPerformed` method can test `itsManual.isSelected()`, etc.

JRadioButtons

The choice among compact, full-size, luxury, and SUV could be indicated by a JComboBox, but the clients have seen JRadioButtons on web pages and they want those. Specifically, you must have a group of four buttons, one for each vehicle type, of which only one can be selected at any given time.

The **JRadioButton** class in the `javax.swing` package is a subclass of JToggleButton, so you have the `isSelected`, `setSelected`, and `addActionListener` methods available as described previously, plus the following constructors:

- `new JRadioButton(String)` creates a new button with the specified text on it.

ButtonGroups

You must specify that the four particular radio buttons you have act as a group, to enforce the rule that only one can be selected at a time. Another application might have two groups of two radio buttons, which would allow one of the first two to be selected as

well as one of the second two. To distinguish one way of grouping from another way, you must add the four radio buttons to a **ButtonGroup** object. The three most useful methods in the `javax.swing.ButtonGroup` class are as follows:

- `new ButtonGroup()` creates a ButtonGroup object.
- `add(AbstractButton)` attaches the given button to the ButtonGroup executor. This method does not return a value.
- `remove(AbstractButton)` removes the given button from the executor. This method does not return a value.

You now have enough information to add the radio buttons to the software. Declare in the CarRentalView class an additional instance variable `itsVehicle` to keep track of the kind of vehicle chosen, add to the panel a label saying to choose one of the following four buttons, and then add the four buttons to the panel. All four buttons should also be added to the same button group and all should be given the same action listener. Listing 9.12 gives the completed coding for subViewFive, thereby completing Version 2 of the CarRental software. Figure 9.10 shows what the display looks like. Remember that the number at the far right is the clock ticking away.

Listing 9.12 subViewFive for the CarRentalView class

```
// CarRentalView class, part 6 (completed)

   private JCheckBox itsManual = new JCheckBox ("manual", false);
   private JCheckBox itsAir = new JCheckBox ("air cond", true);
   private JCheckBox itsCD = new JCheckBox ("CD player", true);
   private String itsVehicle = "compact";


   public JPanel subViewFive()
   {  JPanel panel = new JPanel();
      panel.setBounds (10, 225, this.getWidth() - 20, 40);

      panel.add (itsManual);
      panel.add (itsAir);
      panel.add (itsCD);
      panel.add (new JLabel ("Choose one:"));

      ButtonGroup group = new ButtonGroup();
      ActionListener alis = new ButtonGroupAL();
      String[] text = {"compact", "full-size", "luxury", "SUV"};
      for (int k = 0;  k < text.length;  k++)
      {  JRadioButton car = new JRadioButton (text[k]);
         car.addActionListener (alis);
         group.add (car);
         panel.add (car);
      }

      return panel;
   }  //=====================

   private class ButtonGroupAL implements ActionListener
   {
      public void actionPerformed (ActionEvent e)
      {  itsVehicle = ((JRadioButton) e.getSource()).getText();
      }
   }  //=====================
```

The ItemListener interface

The convention is to attach an ItemListener object to a JCheckBox instead of an ActionListener object when you want the selection to notify a listener object. The AbstractButton class has an `addItemListener` method with an ItemListener parameter for this purpose. The **ItemListener** interface requires this one method heading in any class that is to implement it:

```
public void itemStateChanged (ItemEvent e);
```

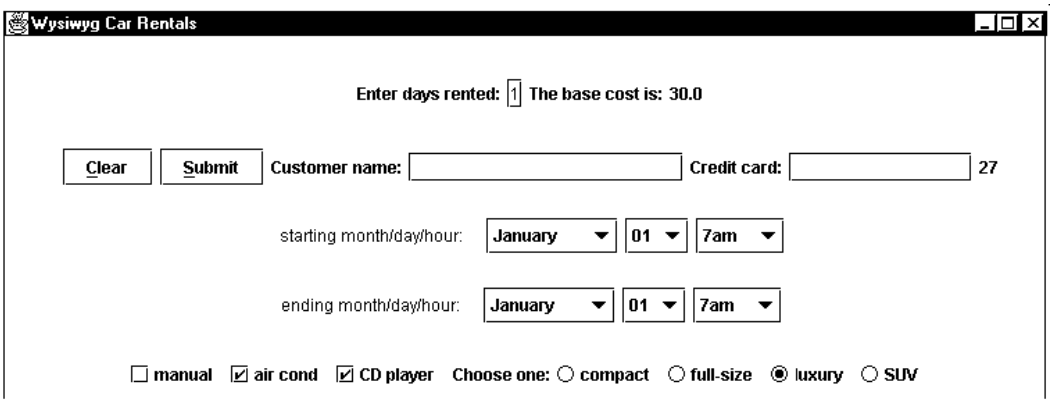


Figure 9.10 Display for the completed CarRental software

If you wanted, for instance, the choice of a manual transmission to trigger an extra 10% discount (because people who drive a stick shift are really special people deserving of extra consideration), you could add an anonymous class as follows:

```
itsManual.addItemListener (new ItemListener()
   {  public void itemStateChanged (ItemEvent e)
      {  if (e.getStateChange() == ItemEvent.SELECTED)
            itsDailyRate *= 0.90;
         else
            itsDailyRate /= 0.90;
      }
   }     );
```

You can see that an **ItemEvent** carries more information than an ActionEvent: You can query whether the event was the selecting of the button (as opposed to unselecting). If you used an ActionListener, you would have to test instead this condition:

```
((JCheckBox) e.getSource()).isSelected().
```

Action commands

Each AbstractButton object has a private String instance variable called its "action command". You may set the value using `someButton.setActionCommand (someString)`. The value of this is that, when clicking a button dispatches an ActionEvent, that ActionEvent takes the action command of the button with it, which you can retrieve using `someEvent.getActionCommand()`.

Suppose that in Listing 9.12 you had statements such as the following:

```
JRadioButton button = new JRadioButton ("compact");
button.setActionCommand ("Pontiac Sunfire");
```

Then you can determine the make of vehicle chosen within the `actionPerformed` method by referring to `e.getActionCommand()`. The alternative would be to have a four-way multiway selection statement based on `((JRadioButton) e.getSource()).getText()`, which would be clumsy.

Exercise 9.38 What changes would Listing 9.12 require to use for a button group of seven days of the week, recording in `itsDay` the text that is on the one chosen?
Exercise 9.39 Modify Listing 9.12 to have it select air conditioning any time the vehicle type is selected as luxury or SUV.
Exercise 9.40 Explain why you need the class cast in Listing 9.12. What class cast could you use instead of the one there?
Exercise 9.41* Revise Listing 9.12 to set a particular make and model of car as the action command for each car. Use a second array of String values so you can make the assignment within the loop.
Exercise 9.42* Revise the `actionPerformed` method of SubmitButtonAL to "add" to `itsModel` all the additional data on the third, fourth, and fifth panels.
Exercise 9.43* Read the documentation for AbstractButton and describe how to put a picture on a button.

9.11 Of Mice And Menus

If you use a modern word processor or spreadsheet, the top of the screen has a bar that says "File", "Edit", "View", etc. That is a **menu bar**. Each JFrame and JApplet has a menu bar associated with it, at the top of the rectangular area. You can access it or replace it using the following:

```
bar = someJFrame.getJMenuBar();
someJFrame.setJMenuBar (someJMenuBar);
```

If you wanted a JMenuBar object put some place else on a JFrame or JApplet, you could create one using the constructor call `new JMenuBar()`. Once you have access to the JMenuBar, you can add several JMenu objects to it as follows:

```
bar.add (menuOne);
bar.add (menuTwo);
```

You have to first create those menus, perhaps as follows:

```
JMenu menuOne = new JMenu ("animals");
JMenu menuTwo = new JMenu ("fruits");
```

This means that the words "animals" and "fruits" will appear as the first two menus to choose from on the JMenuBar, just as the first two choices on the JMenuBar for the Word program are "File" and "Edit". The `add` method in the JMenuBar class returns the JMenu object that is being added, so you could have written the preceding two pairs of statements using the `add` method, as follows:

```
JMenu menuOne = bar.add (new JMenu ("animals"));
JMenu menuTwo = bar.add (new JMenu ("fruits"));
```

Of course, a menu is no good unless it has several menu items to choose from. The following two statements put two kinds of animals on the "animals" menu:

```
menuOne.add (new JMenuItem ("dog"));
menuTwo.add (new JMenuItem ("cat"));
```

Oops! There is no point in doing this, because if the user clicks on one of these kinds of animals, nothing happens. That is because you have no ActionListener to handle the event of clicking. You need to attach an ActionListener object to each item on the menu. The `add` method from the JMenu class returns the JMenuItem object that was added, so a compact way of adding the same ActionListener object to all MenuItems is as follows:

```
ActionListener alis = new MenuItemAL();
menuOne.add (new JMenuItem ("dog")).addActionListener (alis);
menuOne.add (new JMenuItem ("cat")).addActionListener (alis);
menuOne.add (new JMenuItem ("fish")).addActionListener (alis);
```

An alternative `add` method for JMenuItems takes a String value as the parameter, creates a JMenuItem with that text on it, and returns the JMenuItem created, so you can add two menu items to the "fruits" menu as follows:

```
menuTwo.add ("apple").addActionListener (alis);
menuTwo.add ("lemon").addActionListener (alis);
```

The JMenu class is a subclass of JMenuItem, which means that you can have a JMenu as one of the items on another menu. A menu is basically a button you click on to get a popup menu, and a menu item is basically a button you click on to get an effect (from `alis` in the example), so JMenuItem is a subclass of AbstractButton.

Reacting to a menu item choice

Now the user can select either the animals menu or the fruits menu, then click on one of the three or two choices the menu has. That sends the `actionPerformed` message to `alis`, who will need to know which MenuItem was clicked. Fortunately, `alis` can ask the ActionEvent parameter for its source, which will be the JMenuItem clicked. Then `alis` can ask that JMenuItem for its label, so it will know what action to take.

For a whimsical example, suppose the content pane contains a JLabel named `itsSound` that is to have written on it the sound that the animal makes when the user clicks an animal, or the color of the fruit when the user clicks a fruit. The `actionPerformed` method could then be as shown in Listing 9.13.

Listing 9.13 The MenuItemAL inner class of objects

```
private class MenuItemAL implements ActionListener
{
   public void actionPerformed (ActionEvent e)
   {  String choice = ((JMenuItem) e.getSource()).getText();
      if (choice.equals ("dog"))
         itsSound.setText ("woof");
      else if (choice.equals ("cat"))
         itsSound.setText ("meow");
      else if (choice.equals ("fish"))
         itsSound.setText ("burble");
      else if (choice.equals ("apple"))
         itsSound.setText ("red");
      else if (choice.equals ("lemon"))
         itsSound.setText ("yellow");
      else // unrecognized
         itsSound.setText ("what?");
   }
}  //======================
```

MouseListener objects (from java.awt.event)

On an unrelated note, placed here only to justify the title on this section, you can construct a listener object that reacts to user actions with the mouse. It must implement five methods of the MouseListener interface, to respond to each of the five possible mouse actions. The Component class (from which JComponent inherits) has a method for adding a MouseListener object:

```
someComponent.addMouseListener (someMouseListener);
```

A MouseListener object executes `mouseEntered` when the mouse cursor enters the Component and `mouseExited` when the mouse cursor leaves the Component. It executes `mousePressed` when the user presses the mouse button and `mouseReleased` when the user releases it (within the bounds of the Component). If the press and release are done without moving the mouse cursor, the MouseListener object then executes `mouseClicked` as well.

The five methods are fully described in Listing 9.14. Note that a special MouseEvent object is passed as a parameter. It allows you to retrieve the x-coordinate and y-coordinate of the point where the mouse action took place. In the example, this information is written on some label named `lab`.

Listing 9.14 The ClickML inner class of objects

```
private class ClickML implements MouseListener
{
   public void mouseEntered (MouseEvent e)
   {  lab.setText ("enter " + e.getX() + "," + e.getY());
   }

   public void mouseExited (MouseEvent e)
   {  lab.setText ("exit " + e.getX() + "," + e.getY());
   }

   public void mousePressed (MouseEvent e)
   {  lab.setText ("press " + e.getX() + "," + e.getY());
   }

   public void mouseReleased (MouseEvent e)
   {  lab.setText ("release " + e.getX() + "," + e.getY());
   }

   public void mouseClicked (MouseEvent e)
   {  lab.setText ("click " + e.getX() + "," + e.getY());
   }
}  //======================
```

Adapters

You may replace `implements MouseListener` by `extends MouseAdapter` in the ClickML class heading. **java.awt.event.MouseAdapter** is a convenience class in Java that implements all five methods with do-nothing coding, so a subclass can just override the ones it wants. This is nice when you only want to react to one or two of the five possible events. There is also a **java.awt.event.WindowAdapter** class that has do-nothing coding for all seven WindowListener methods (Listing 9.2) for the same purpose.

Exercise 9.44 Add a third menu named "cars" to the `bar` JMenuBar. Then add three brands of cars to that menu, each with the same ActionListener as all other items.
Exercise 9.45 Add logic to Listing 9.13 to have a click on one of the three kinds of cars put on `itsSound` an alliterative adjective describing the car of the preceding exercise.
Exercise 9.46 Rewrite Listing 9.2 to use a WindowAdapter instead.
Exercise 9.47* Read the documentation for JMenuBar and describe how you would find out the current number of menus on the `bar` JMenuBar.
Exercise 9.48** Read the documentation for JMenu and describe how you would remove the "cat" from the "animals" menu and leave the other two.
Exercise 9.49** Revise Listing 9.13 to use `getActionCommand` appropriately. Also make the other changes needed to store the action commands in the five menu items.

*9.12 More On LayoutManagers And JLists (*Sun Library)*

The only way you have seen to control the layout of Components is FlowLayout (from the `java.awt` package), which puts things in book-reading order one row at a time, and `setBounds` to specify precisely the location and size of a Component when the LayoutManager is `null`. It is easier to use one of the several LayoutManager objects available when it gives you what you want. The following methods are useful with any of the LayoutManagers described here:

- `someContainer.setLayout(someLayoutManager)` specifies the LayoutManager to be used by that Container.
- `someContainer.getLayout()` returns the LayoutManager currently in use.
- `someLayoutManager.layoutContainer(someContainer)` has the layout's container recalculate the layouts when an addition or deletion is made. This method is in the **LayoutManager** interface that all layout managers implement.
- `Box.createGlue()` returns a Component that expands to fill any available excess space. This lets you avoid having all excess space at the end of a layout; it is a class method in the `javax.swing.Box` class.

BorderLayout objects (from java.awt)

The content pane of a JFrame or JApplet is initially given a BorderLayout. You may add up to five Components to a Container that has a BorderLayout, one in each of the NORTH, SOUTH, EAST, WEST, or CENTER of the area. A Component in the NORTH or SOUTH part extends horizontally for the entire width of the area. A Component in the EAST or WEST usually takes up about a third of the width. A Component in the CENTER expands to take up whatever area is left.

- `new BorderLayout(x, y)` constructs a layout manager with `x` pixels between Components horizontally and `y` pixels between them vertically.
- `someContainer.add(someComponent, BorderLayout.NORTH)` puts the Component in the NORTH position (and similarly for the other four positions) when using BorderLayout. Leaving out the second parameter puts the Component in the CENTER. If you try to put two Components in one part, it could foul things up.

BoxLayout objects (from javax.swing)

A BoxLayout puts all its components in one row, either horizontally or vertically as you choose. It makes them all the same height (respectively, width) to the extent possible. Different Containers cannot share the same BoxLayout object.

- `new BoxLayout(someContainer, BoxLayout.X_AXIS)` constructs a manager with a horizontal row of Components.
- `new BoxLayout(someContainer, BoxLayout.Y_AXIS)` constructs a manager with a vertical column of Components.

GridLayout objects (from java.awt)

A GridLayout object divides the area into a rectangular grid of subareas, all of the same size. Components are added left-to-right in the first row, then in the next row, etc. You may specify the number of rows to be zero, which means that the layout will allow as many rows as are needed for the Components you add (and similarly for columns, but you cannot have both values zero). Note that specifying 1 for the rows or columns gives roughly the equivalent of a BoxLayout.

- `new GridLayout(numRows, numCols)` constructs a manager with `numRows` rows and `numCols` columns.
- `new GridLayout(numRows, numCols, x, y)` constructs a manager with `numRows` rows and `numCols` columns, and with `x` pixels between Components horizontally and `y` pixels between Components vertically.

CardLayout objects (from java.awt)

The Components added in a CardLayout do not all appear in its Container; only one appears at a time, initially the first one added to the Container. The CardLayout class has several methods for switching in another Component to display. The ordering of the Components is the order in which the `add` method was called for them.

- `new CardLayout()` constructs a manager with only one Component showing.
- `someCardLayout.first(someContainer)` displays the first Component.
- `someCardLayout.last(someContainer)` displays the last Component.
- `someCardLayout.next(someContainer)` displays the next Component after the one currently displayed.
- `someCardLayout.previous(someContainer)` displays the Component before the one currently displayed.

JList objects (from javax.swing)

JList is a subclass of JComponent that lists a number of Objects. You specify the Objects as an `Object[]` parameter to the JList constructor. The order of the Objects displayed is the order they occur in the array (e.g., component 3 of the array is at index 3 in the JList and will be the fourth Object displayed).

- `new JList(someArray)` constructs a JList with as many items on it as the array has. `someArray` must be an array of Objects (or a subclass of Object).
- `someJList.setVisibleRowCount(someInt)` specifies how many values will appear at a time. However, you do not get a scroll bar with it, so you generally want to also say `someContainer.add(new JScrollPane(someJList))`.
- `someJList.getSelectedIndex()` returns the index number of the Object currently selected. It returns -1 if no value is currently selected.
- `someJList.getSelectedValue()` returns the Object currently selected, or null if nonesuch.
- `someJList.setSelectedIndex(someInt)` makes that Object the one currently selected.
- `someJList.addListSelectionListener(someListSelectionListener)` adds a listener object that reacts whenever a selection is made.

A typical kind of listener class for a JList is the following:

```
private class MyListEar implements ListSelectionListener
{  public void valueChanged (ListSelectionEvent e)
   {  // whatever action is appropriate
   }
}  //======================
```

You may have a JList that allows selection of several items from the list at one time. But the default (when the JList object is constructed) is to allow just one selection. Read your Java documentation if you want to find out how to handle multiple selections.

*9.13 More On JComponent, ImageIcon, And AudioClip (*Sun Library)*

You may set the Font object that controls how letters and other characters look for any JComponent object such as a JTextField or JLabel (Fonts were discussed at the end of Chapter Eight). You only need the following kind of statement:

```
someJComponent.setFont (someFont);
```

You may want to put a border on a JComponent, particularly if it is a JPanel. The simplest way to do this is with the following statement, which places a simple green line around a component. Look in the Java documentation for more kinds of borders:

```
someJComponent.setBorder
       (BorderFactory.createLineBorder (Color.green));
```

ImageIcon objects (from javax.swing)

Suppose you have a small picture in a disk file named "oakTree.gif" in the plants folder. You can get it into your program as follows:

```
ImageIcon tree = new ImageIcon ("plants/oakTree.gif");
```

Now you can display the picture in any of several ways. One way is to just put it at the `<x,y>` coordinates where you want it on a particular component:

```
tree.paintIcon (someComponent, someGraphics, x, y);
```

Another way is to put it on a JLabel or other JComponent, along with whatever other text might be there (see your JLabel documentation for how to control which goes where):

```
someJComponent.setIcon (tree);
```

AudioClip objects (interface from java.applet)

An Applet (and thus a JApplet) has two methods that help you play sounds. `getDocumentBase` returns the URL of the place where the HTML file was loaded from, and `getAudioClip` returns an AudioClip object found at a particular URL. So you only need a statement of the following form to get some music into your Applet:

```
AudioClip music = getAudioClip (getDocumentBase(), "pop.au");
```

The AudioClip interface prescribes three methods to let you control the sound of `music`:

- `music.play()` plays the sound one time through.
- `music.loop()` plays the sound over and over again.
- `music.stop()` stops playing the sound.

9.14 Review Of Chapter Nine

Listing 9.5 illustrates an inner class, the only new language feature introduced in this chapter other than anonymous classes, which should be used quite sparingly anyway.

About the Java language:

- If class X **implements** Y, then X must override every method prescribed by Y. Your coding is not allowed to attempt to execute the body of a method in Y itself.
- You can declare a class `private class X` inside of an outer class Out. You then construct an object of this **inner class** X only with an Out object as the executor, as in `out.new X()`. Of course, if the construction takes place inside an instance method or constructor of the Out class, the Out object defaults to the executor.
- Inside class X, you may refer to the Out object that constructed the X executor of a method as `Out.this`. A reference to an instance variable or method inside X that only makes sense with an Out object as executor defaults to `Out.this`.
- To make an **anonymous class**, put the body of the class directly after the parentheses of a constructor call `new Whatever()`, with no heading on the class. The class is considered to be a subclass of Whatever, or an implementation if Whatever is an interface. The anonymous class is considered an inner class; it cannot contain constructors or class methods or class variables. It is bad form to have more than half-a-dozen statements total inside an anonymous class.
- **Event-driven programming** is attaching listener objects to components (**registering** the event-handler) and thereafter only taking action in response to an event that causes one of those components to send an action message to its listener object.
- Read through the documentation for the `java.awt` and `javax.swing` classes partially described in this chapter. Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

About the java.awt.Component class (has no constructors):

- `someComponent.paint(someGraphics)` paints this Component.
- `someComponent.repaint()` colors over everything in the background color and then calls paint.
- `someComponent.getGraphics()` returns the Component's Graphics object.
- `someComponent.setSize(widthInt, heightInt)` resizes it to the given width and height.
- `someComponent.setVisible(someBoolean)` makes it visible or invisible with its existing Components. Components added to it later may not appear, so `setVisible(true)` should be the last thing you do when constructing a JFrame, except perhaps call `paint`.
- `someComponent.setBounds(xInt, yInt, widthInt, heightInt)` puts its upper-left corner at `<xInt,yInt>` and makes its size `<widthInt,heightInt>`. This only takes effect if the LayoutManager of its Container is set to `null`.
- `someComponent.getWidth()` returns the int width of the Component in pixels.
- `someComponent.getHeight()` returns the int height of the Component in pixels.

About the java.awt.Container class (a subclass of Component):

- `someContainer.add(someComponent)` adds the parameter to the end of a list of Components attached to this Container and returns the Component that was added.
- `someContainer.setLayout(someLayoutManager)` specifies the way the Components will be laid out in this Container. The LayoutManager must be set to `null` if its Components are to use `setBounds`.
-

About the javax.swing.JFrame class (a subclass of Container):

- `new JFrame(titleString)` creates an initially invisible window on the monitor screen with the specified title.
- `someFrame.getTitle()` returns its title (a String at the top of the frame).
- `someFrame.setTitle(titleString)` replaces a JFrame object's title.
- `someFrame.addWindowListener(someWindowListener)` attaches a listener.
- `someFrame.getContentPane()` returns the Container object to which all components of a JFrame are to be attached. Also available for a JApplet.
- `someFrame.getJMenuBar()` returns its menu bar. Also available for a JApplet.
- `someFrame.setJMenuBar(JMenuBar)` replaces its menu bar. Also available for a JApplet.

About the javax.swing.JComponent class (a subclass of Container):

- JComponent is designed for a pluggable look-and-feel.
- `someJComponent.setToolTipText(flyoverString)` specifies the text that pops out on a flyover.

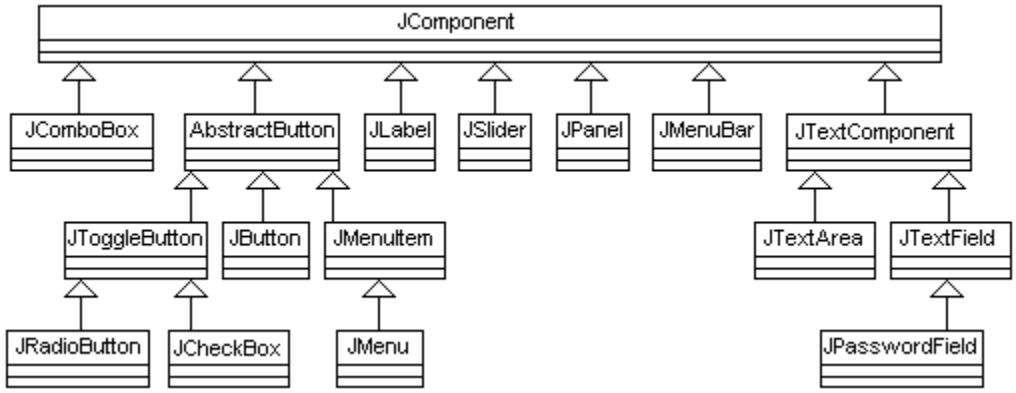


Figure 9.11 Subclasses of JComponent discussed in this chapter

Constructors for various JComponent subclasses (all in javax.swing):

- `new JPanel()`. JPanel is a subclass of JComponent.
- `new JLabel(textString)`. JLabel is a subclass of JComponent.
- `new JTextField(textString)`. JTextField a subclass of JTextComponent. A JTextField allows just one line of input text and reacts to the ENTER key.
- `new JTextField(widthInt)` specifies the width as the number of 'm' characters.
- `new JPasswordField(widthInt)` specifies the width in 'm' characters.
- `new JButton(textString)`. JButton is a subclass of AbstractButton.
- `new JTextArea(rowInt, columnInt)`. JTextArea is a subclass of JTextComponent. It is described in some detail in Chapter Six.
- `new JSlider(directionInt)`. JSlider is a subclass of JComponent. The int parameter can be `JSlider.HORIZONTAL` or `JSlider.VERTICAL`.

- `new JComboBox(Object[])`. JComboBox is a subclass of JComponent. The given list of Objects tells the choices available, in the order listed.
- `new JCheckBox(textString, someBoolean)`. JCheckBox is a subclass of AbstractButton. The optional second parameter makes its selected status `true` or `false`.
- `new JRadioButton(textString, someBoolean)`. JRadioButton is a subclass of AbstractButton. The optional second parameter makes its selected status `true` or `false`. In a ButtonGroup of buttons, only one button can be selected.
- `new JMenuBar()`. JMenuBar is a subclass of JComponent.
- `new JMenu(textString)`. JMenu is a subclass of JMenuItem.
- `new JMenuItem(textString)`. JMenuItem is a subclass of AbstractButton.

Methods that add listener objects to subclasses of JComponents:

- `comp.addActionListener(someActionListener)` is in the AbstractButton, JTextField, and JComboBox classes, and thus is also available for JButton, JRadioButton, JCheckBox, and JMenuItem objects.
- `someJSlider.addChangeListener(someChangeListener)` is in the JSlider class.
- `someAbstractButton.addItemListener(someItemListener)` is in the AbstractButton class, but we generally only use it for a JCheckBox object.
- `someComponent.addMouseListener(someMouseListener)` is in the Component class and therefore available for all kinds of JComponents.

Other quite useful methods for subclasses of JComponent:

- `comp.setText(someString)` changes the text on the component. It is in JLabel, AbstractButton, and JTextComponent, and thus available for JButton, JRadioButton, JCheckBox, JMenuItem, JTextField, and JTextArea objects.
- `comp.getText()` returns the current text on the component. It has the same availability as the preceding `setText`.
- `someJPasswordField.getPassword()` returns a char array.
- `someAbstractButton.setMnemonic(someChar)` makes that character the hotkey for that AbstractButton, so that the ALT key followed by this character effectively clicks or selects that button.
- `someJTextArea.append(someString)` adds text to the end of the existing text in the JTextArea.
- `someJSlider.getValue()` returns the int value where the user released the thumb of a JSlider.
- `someJSlider.setMinimum(someInt)` specifies the smallest value that appears on a JSlider.
- `someJSlider.setMaximum(someInt)` specifies the largest value that appears on a JSlider.
- `someJSlider.setMinorTickSpacing(someInt)` specifies the distance between ticks on a JSlider.
- `someJComboBox.getSelectedIndex()` returns the int value that is the index of the currently selected Object in a JComboBox.
- `someJComboBox.setSelectedIndex(indexInt)` selects the Object with the specified index in a JComboBox.
- `someJComboBox.getSelectedItem()` returns the Object currently selected in a JComboBox.
- `someJComboBox.setMaximumRowCount(someInt)` gives the number of items displayed by a JComboBox, with a scroll bar if the actual number of items is greater.
- `someJComboBox.addItem(someObject)` adds this Object to the end of the list for a JComboBox.
- `someJComboBox.removeItemAt(indexInt)` deletes the Object at that index from the JComboBox.

- `comp.isSelected()` tells whether the JRadioButton or JCheckBox is currently selected.
- `comp.setSelected(someBoolean)` makes the JRadioButton or JCheckBox selected if the parameter is `true` and makes it unselected otherwise.
- `comp.setActionCommand(actionString)` stores a String value in the JComboBox, JTextfield, or AbstractButton (so it can be retrieved from an ActionEvent object for JButton, JRadioButton, JCheckBox, JMenu, or JMenuItem objects).
- `someJMenuBar.add(someJMenu)` adds the next menu to the MenuBar and returns that JMenu object.
- `someJMenu.add(someJMenuItem)` adds the next menu item to the JMenu and returns the JMenuItem that was added.
- `someJMenu.add(someString)` adds to the JMenu a new JMenuItem that it constructs from the String, and returns the JMenuItem that was added.

About the javax.swing.Timer class (a direct subclass of Object):

- `new Timer(waitInt, someActionListener)` creates a Timer object that, when running, sends an `actionPerformed` message to the ActionListener parameter every `waitInt` milliseconds.
- `someTimer.start()` has the Timer object start sending notifications to its listener.
- `someTimer.stop()` has the Timer object stop sending notifications to its listener.
- `someTimer.setRepeats(someBoolean)` if set to `false` has the Timer object only send one notification and then stop; `true` gives multiple notifications.

About the javax.swing.ButtonGroup class (a direct subclass of Object):

- `new ButtonGroup()` creates a ButtonGroup object to which JRadioButtons can be attached. When a button in that group is selected, the ButtonGroup object will deselect all other buttons in that group. A ButtonGroup is not a Component.
- `someButtonGroup.add(someAbstractButton)` attaches the given button to the ButtonGroup executor. This method does not return a value.
- `someButtonGroup.remove(someAbstractButton)` removes the given button from the ButtonGroup executor. This method does not return a value.

About EventObject and its subclasses:

- `EventObject` (in `java.util`) has the method `getSource()` that returns the Object that generated the event. It has the following five among its subclasses:
- `WindowEvent` (in `java.awt.event`) is used for JFrames.
- `ActionEvent` (in `java.awt.event`) has the method `getActionCommand()` for retrieving the action command String value attached to the object that generated the event (if any).
- `ChangeEvent` (in `javax.swing.event`) is used for JSliders.
- `ItemEvent` (in `java.awt.event`) has the `getStateChange()` method that either returns `ItemEvent.SELECTED` (to indicate that the object was selected) or `ItemEvent.DESELECTED` (so it was not selected).
- `MouseEvent` (in `java.awt.event`) has the `getX()` and `getY()` methods for finding the `<x,y>` coordinates where the mouse action occurred.

About listener interfaces (all are sub-interfaces of EventListener):

- The `WindowListener` interface (in `java.awt.event`) has seven methods, one of which is `windowClosing(WindowEvent)`, called when the user clicks on the window's closer icon.
- The `ActionListener` interface (in `java.awt.event`) has one method `actionPerformed(ActionEvent)` that is called when a Component experiences an event that its ActionListener object can respond to.

- The `ChangeListener` interface (in `javax.swing.event`) has one method `stateChanged(ChangeEvent)` that is called when a Component experiences an event that its ChangeListener object can respond to.
- The `ItemListener` interface (in `java.awt.event`) has one method `itemStateChanged(ItemEvent)` that is called when an item changes.
- The `MouseListener` interface (in `java.awt.event`) has five methods (see Listing 9.14) that are called when various mouse actions occur.

Answers to Selected Exercises

9.1 Delete imports and replace "Graphics" by "java.awt.Graphics" in one place, "Graphics2D" by "java.awt.Graphics2D" in two places, and "Line2D" by "java.awt.geom.Line2D" in one place.

9.2 The lower-right corner of the drawing, for depth=575, is <575+1, 575>, so setSize (586, 585).

9.7 The text field that initially contains the number 1 would appear to the left of the phrase "Enter days rented:" on the display.

9.8 Add the following statement just before the return statement:

```
panel.add (new JLabel ("for a compact"));
```

9.9

```
public void actionPerformed (ActionEvent e)
{   double d = Double.parseDouble (itsDaysRented.getText());
    itsBaseCost.setText ("" + (d - 32.0) * 5 / 9);
}
```

9.11 Replace the following: "private JPanel subViewOne() { JPanel panel = new JPanel();" by this: "private class SubViewOne extends JPanel { public SubViewOne() { super();" Replace "this" by "CarRentalView.this" in two places. Replace "panel" by "this" throughout the method, except replace "return panel;" by "}".

9.12 A bad answer for this problem is to make itsDaysRented and itsBaseCost public variables. So add to CarRentalView two methods getDaysRentedText() and setBaseCostText(String) with the obvious one statement of coding each and have the following statement in subViewOne: itsDaysRented.addActionListener (new DaysRentedAL (this)); Then code the DaysRentedAL class as follows:

```
public class DaysRentedAL implements ActionListener
{   private CarRentalView itsView;
    public DaysRentedAL (CarRentalView givenView)
    {   super();
        itsView = givenView;
    }
    public void actionPerformed (ActionEvent e)
    {   int d = Integer.parseInt (itsView.getDaysRentedText());
        itsView.setBaseCostText ("" + (d - d / 7) * CarRentalView.PRICE_PER_DAY);
    }
}
```

9.15 Add the following statement:

```
    ((JButton) e.getSource()).setText ("Ouch!");
```

9.16 Replace the two lines beginning with "if" by the following:

```
if (source.getText().equals ("X"))
    JOptionPane.showMessageDialog (null, "You chose that one before");
else if (source.getText().equals ("O"))
    JOptionPane.showMessageDialog (null, "That one belongs to me already");
```

9.18

```
public void actionPerformed (ActionEvent e)
{   if (itsCustomerName.getText().length() == 0 || itsCreditCard.getText().length() == 0)
        System.out.println ("You cannot have a blank entry");
    else
        itsModel.add (itsCustomerName.getText() + "  " + itsCreditCard.getText());
}
```

9.19 It would appear centered below the row that contains the other components.

9.20 Move the two statements that add the buttons to be at the end of the subViewTwo method, just before the return statement.

9.26 Move the declaration of itsInput to right after content.add (itsOutput), omitting "private": JSlider itsInput = new JSlider (JSlider.HORIZONTAL); In the stateChanged method, insert this as the first statement:

```
JSlider itsInput = (JSlider) e.getSource();
```

9.27 Define another instance variable:

```
private JSlider itsNeg = new JSlider (JSlider.HORIZONTAL);
```

Put the following two statements in the AddInputs init method:

```
content.add (itsNeg);
itsNeg.addChangeListener (this);
```

Replace the stateChanged method by the following:

```
public void stateChanged (ChangeEvent e)
{   int amount = event.getSource() == itsInput
                    ?  itsInput.getValue()  :  - itsMinus.getValue();
    itsOutput.append ("\nEntry: " + amount);
    itsData += amount;
}
```

9.33 Change the String parameter in the last statement of the actionPerformed method to the following:

```
(Clock.this.itsCounter / 1000) + "." + (Clock.this.itsCounter / 100 % 10).
```

9.34 Replace the first statement of the start method by setText ("0");

Replace the first statement of the actionPerformed method by the following:

```
Clock.this.setText ("" + (1 + Integer.parseInt (Clock.this.getText())));
```

9.37

```
private JComboBox boxOfNumbers (int howMany)
{   String[ ] values = new String [howMany];
    for (int k = 0;  k < howMany;  k++)
        values[k] = "" + k;
    return new JComboBox (values);
}
```

9.38 You only need to rename itsVehicle as itsDay and write the declaration of text as follows:

```
String[ ] text = {"Sun", "Mon", "Tue", "Wed", "Thur", "Fri", "Sat"};
```

9.39 Add this statement as the last statement of ButtonGroupAL's actionPerformed method:

```
if (itsVehicle.equals ("luxury") || itsVehicle.equals ("SUV"))
    itsAir.setSelected (true);
```

9.40 e.getSource() returns an Object value, not a JRadioButton value.
So e.getSource() does not have a getText method. So a cast is needed.
Since the getText method is actually in the AbstractButton class, you could cast it
to AbstractButton or even JToggleButton and get the same effect.

9.44

```
JMenu menuTres = bar.add (new JMenu ("cars"));
menuTres.add ("Ford").addActionListener (alis);
menuTres.add ("Chevrolet").addActionListener (alis);
menuTres.add ("Lexus").addActionListener (alis);
```

9.45 Insert the following just before the last "else" of the method in Listing 9.13:

```
else if (choice.equals ("Ford"))
    itsSound.setText ("fantastic Ford");
else if (choice.equals ("Chevrolet"))
    itsSound.setText ("shiny Chevrolet");
else if (choice.equals ("Lexus"))
    itsSound.setText ("lovely Lexus");
```

9.46

```
public class Closer extends WindowAdapter
    {   public void windowClosing (WindowEvent e)
        {   System.exit (0);
        }
    }
```

10 Exception-Handling

Overview

The situation described here illustrates the full process for developing software of significant complexity. It also illustrates the need for technical knowledge of a subject matter area in order to develop software that solves a problem in that area. However, it is not so difficult that you cannot follow it with moderate effort. The structure of this chapter is as follows:

- Section 10.1 describes the problem to be solved by the Investor software.
- Sections 10.2-10.4 present the throwing and catching of Exceptions and apply those new language features to parsing numbers and using text files. These are the only new language features needed elsewhere in this book.
- Section 10.5 gives the Analysis, Test Plan, and Design of the Investor software.
- Sections 10.6-10.7 use arrays to develop the Portfolio class.
- Section 10.8 uses statistics and Math methods for the Asset class.
- Section 10.9 is an independent section that tell about some kinds of Java statements not otherwise used in this book.
-

10.1 Problem Description For The Investor Software

A group of investors comes to your consulting firm to ask for help in choosing their investments. Each of them has a substantial amount of money in a tax-sheltered 401(k) plan at work, ranging from $100,000 to $300,000. The investors want to know what investment strategy will give them the best growth in value until retirement. Your job is to develop software that will help them.

The investors are in their early to mid forties. They have children who are beginning college, so they will not be able to add to their retirement plans for the next 20 years or so (even after the progeny graduate, the loans have to be paid off). On the other hand, federal law prevents them from withdrawing from the 401(k) plan until they are around age 60, except with a hefty penalty. The government makes this restriction because of the great tax benefits a 401(k), 403(b), and IRA offer: The investments grow free of taxes for as long as they remain in the retirement plan. For these reasons, none of the investors plans to add to or subtract from the retirement plans for the next 15 to 20 years.

The money in this particular 401(k) plan is invested in five different mutual funds, each with a diverse holding in a single class of financial assets. The five asset classes available are money market, short-term bonds, domestic large-cap stocks, domestic small-cap stocks, and foreign stocks. The money market fund, for instance, purchases U.S. Treasury bonds and other high-quality bonds with 3-month maturities. A bond purchased for $9,880 will repay $10,000 at the end of 3 months, which is $120 interest, which is 1.21% for the 3 months, or 4.95% annualized. Since the mutual fund holds bonds of various maturities coming due every week at least, and since repayment of the bonds is never in doubt, the value of the investment never drops in value. This is what investors mean when they talk about "cash."

The company that these investors work for only allows them to change the allocation of money among the mutual funds quarterly. That is, each three months an investor can move part or all of the money from one or more mutual funds in to other mutual funds.

Each investor currently has a certain amount of the 401(k) money in each asset class. The investors want to know how to adjust the allocation of money to the various mutual funds each quarter so they have the best results at the end of the 15 to 20 years.

The background required

Your consulting firm is able to take on this assignment only because:

(a) It has an expert on call who knows very well how 401(k) plans, mutual funds, and the financial markets work (the expert's name is Bill), and
(b) It has software designers who have a moderate understanding of those things.

The investors had previously gone to another software development firm, but the people there, though quite good at developing software, knew almost nothing about investing. Your firm knows how important it is that software designers be reasonably comfortable with the subject matter area that the software is to model. That is why your firm hired job applicants with courses in a wide range of technical and scientific fields, even if they had only moderately good grades in computer science courses. The firm ignored applicants who excelled at their computer science courses but had little breadth of knowledge.

After all, no one pays good money for just any old program, no matter how efficient and well-designed. The program has to solve a problem that people need to have solved, and that generally requires knowledge of a subject matter area such as biology, physics, chemistry, or economics. And more often than not, it requires a moderately sophisticated understanding of mathematics.

The development for this problem will become a little technical at times. That is unavoidable in much of software development. You need to see how technical knowledge is applied to produce a good software design. And this particular technical knowledge is something you should have anyway. If you do not understand how long-term financial investments work, you need to learn about them. After all, what good will it do to make a lot of money as a computer scientist if you do not know anything about how to invest it for long-term growth? You do not want to have to live on peanut butter and baked beans during your retirement.

10.2 Handling RuntimeExceptions; The try/catch Statement

The investors will enter decimal numbers in response to requests from the investor program. From time to time the user may mis-type a numeral, accidentally hitting a letter or two decimal points or some such. That makes the string of characters "ill-formed". Some coding may have the following statements:

```
String s = JOptionPane.showInputDialog (prompt);
return Double.parseDouble (s);
```

The method call `Double.parseDouble(s)` can produce the exceptional situation. If the string of characters is ill-formed, this `parseDouble` method call executes the following statement, which throws a NumberFormatException at the `askDouble` method:

```
throw new NumberFormatException ("some error message");
```

An Exception is an object that signals a problem that should be handled. As written, the `askDouble` method cannot handle the problem, so it crashes the program. The `askDouble` method should be modified to ask again for a well-formed numeric value. As another example of a throw statement, the following statement might be appropriate in a method whose precondition is that a parameter `par` is to be non-negative:

```
if (par < 0)
   throw new IllegalArgumentException ("negative parameter");
```

Thrown Exceptions

Certain methods or operations throw an Exception when given information that does not fit their requirements. Exceptions come in two groups: RuntimeExceptions and other Exceptions. The following lists the kinds of **RuntimeExceptions** that you are most likely to see with your programs:

- **ArithmeticException** is thrown when e.g. you evaluate `x / y` and `y` is zero. This only applies to integers; with doubles, you get infinity as the answer.
- **IndexOutOfBoundsException** is thrown when e.g. you refer to `s.charAt(k)` but `k >= s.length()`. The two subtypes are ArrayIndexOutOfBoundsException for an array and StringIndexOutOfBoundsException for a String value.
- **NegativeArraySizeException** is thrown when e.g. you execute `new int[n]` to create an array and the value of `n` is a negative number.
- **NullPointerException** is thrown when e.g. you refer to `b[k]` and `b` has the `null` value, or when you put a dot after an object variable that takes on the `null` value. But the latter only applies when the object variable is followed by an instance variable or instance method; it does not apply for a class variable or class method.
- **ClassCastException** is thrown when e.g. you use `(Worker) x` for an object variable `x` but `x` does not refer to a Worker object.
- **NumberFormatException** is thrown by e.g. `parseDouble` or `parseInt` for an ill-formed numeral.

The RuntimeException class is a subclass of Exception. The kinds of Exceptions listed above are all subclasses of RuntimeException. All of them are in the `java.lang` package that is automatically imported into every class.

Crash-guarding an expression

When you write a method that executes a statement that can throw a RuntimeException, you should normally handle the situation in your method. The best way to handle it is to make sure that the Exception can never be thrown: Test a value before using it in a way that could throw an Exception object. For instance:

- The phrase `if (x != 0) y = 3 / x` cannot throw an ArithmeticException.
- The phrase `for (k = 0; k < b.length && b[k] < given; k++)` cannot throw an IndexOutOfBoundsException when `b[k]` is evaluated.
- The phrase `if (p != null && p.getAge() > 5)` cannot throw a NullPointerException.
- Nor can `if (p == null || p.getAge() <= 5)`.

These phrases use crash-guard conditions, so-called because the conditions guard against crashing the program due to an uncaught Exception. The last three listed depend on the short-circuit evaluation of boolean operators.

The try/catch statement

You can avoid all RuntimeExceptions by proper programming. But the NumberFormatException and a few others cannot be avoided easily. In those cases, it is generally easier to handle the Exception by writing a **try/catch statement** as follows:

```
try
{  // normal action assuming the Exception does not arise
}catch (Exception e)  // specify the kind of Exception
{  // special action to take if and when the Exception arises
}
```

The try/catch statement requires the beginning and ending braces for both blocks even if you have only one statement inside a block. If an Exception is thrown by any statement within the **try-block**, that block terminates abruptly and the statements within the **catch-block** are executed. In Figure 10.1, for instance, when the `result` method is called with the argument 0, no value is assigned to `quotient` and no value is returned by the method call. Instead, `y = 7` is the very next operation carried out. Note: This book puts the right brace on the same line with the `catch` keyword to remind you that `catch` is not the first word of a statement, as it does the `while` in a do-while statement.

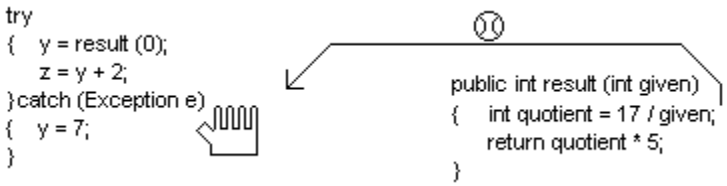


Figure 10.1 Method throwing an exception caught by a catch-block

Insisting on valid numeric input

In many cases when the user enters an ill-formed numeral, the software should repeatedly try for good input. This is done by the following logic. It has an **infinite loop**, which is normally indicated in Java by `for(;;)`. The runtime system executes the logic in the try-block. If no Exception arises, the correct value is returned. In general, the runtime system totally ignores a catch-block unless some action in a try-block throws an Exception:

```
public double getUserInput (String prompt)
{  for (;;)
      try
      {  String s = JOptionPane.showInputDialog (prompt);
         return Double.parseDouble (s);
      }catch (NumberFormatException e)
      {  prompt = "That was a badly-typed number. Try again.";
      }
}
```

The Exception is thrown if the String parameter `s` that is to be converted to a decimal number contains letters or other defects. In that case, the runtime system goes immediately to the catch-block and executes its logic. This prints a message and goes through another iteration of the loop. The `for(;;)` idiom is special in that the compiler does not insist you have a `return` statement in both the try and the catch blocks.

If the user clicks the CANCEL button in response to the coding above, `s` is `null` and so `parseDouble` throws a NullPointerException. Since no catch-block is available for a NullPointerException, the Exception will be thrown to the method that called the coding.

Multiple catch-blocks

If you know there are several kinds of Exceptions that a logic sequence could produce, and you want to handle each one differently, you may have more than one catch-block, as shown in Listing 10.1. The runtime system checks each of the Exception types in order until it finds the first one that fits (the same class or a superclass of the kind of Exception that was thrown). So the assignment `x = s.charAt(5)` is not made (in the third catch-block) if 5 is outside the range of possible indexes (i.e., `b.length() <= 5`). The last catch-block in the statement catches all remaining kinds of Exception that might occur.

Listing 10.1 A try/catch statement with multiple catch-blocks

```
try
{  x = s.charAt (5) / Integer.parseInt (s);
}catch (NumberFormatException e)
{  return 0;
}catch (IndexOutOfBoundsException e)
{  System.out.println (e.getMessage() + " for index 5");
}catch (ArithmeticException e)
{  x = s.charAt (5);
}catch (Exception e)
{  System.exit (0);
}
```

Note that each catch-block has what is basically a parameter of Exception type, which makes it different from any other kind of Java statement. A catch-block is "called" by the runtime system when any statement within the try-block throws its kind of Exception. All Exception objects have a method named `getMessage` that returns an informative String value, as used in the second catch-block.

The `getUserInput` method given earlier in this section could have the following second catch-block added at the end of its try-catch statement. That would cause the segment to continue asking for numeric input:

```
catch (NullPointerException e)
{  say ("I really must insist you enter a number.");
}
```

Exception-handling in the IO class

Listing 10.2 (see next page) shows a more sophisticated version of `askDouble`, `askInt`, and `askLine` than the one in Chapter Six. It uses exception-handling. This will be much better for the investor software. The `askLine` method substitutes an empty String for the `null` value you get when the user cancels the input window. The `askInt` and `askDouble` methods rely on this to avoid a NullPointerException. They both allow the user to press the ENTER key to indicate an entry of zero. They use the standard `trim()` method from the String class to discard any leading or trailing blanks.

Perhaps the only time you should not handle a potential RuntimeException within a method you write is when you do not know which of several possible ways to handle it, because it depends on what the calling method wants to do. For instance, a method that is to compare two objects to see which comes first, using `x.compareTo(y)`, throws a ClassCastException if the two objects are not the same type (such as one Person object and one Time object). The method cannot know what the appropriate action is. So it lets the calling method handle the Exception. The calling method should either avoid the possibility of a ClassCastException with a crash-guard or other protection, or it should have a try-catch statement taking whatever action is appropriate.

You may choose to add the phrase `finally {...}` at the end of the try-catch structure. This block is guaranteed to be executed after the try-block is executed or, if it throws an Exception, after the appropriate catch-block is executed. It is sometimes used to release system resources that were earlier allocated. You will almost surely have no need for this feature in your first few computer science courses.

Listing 10.2 Alternative to the IO class in Listing 6.2

```
import javax.swing.JOptionPane;

public class IO
{
   /** Display a message to the user of Jo's Repair Shop. */

   public static void say (Object message)
   {  JOptionPane.showMessageDialog (null, message,
                "Jo's Repair Shop", JOptionPane.PLAIN_MESSAGE);
   }  //=====================

   /** Display prompt to the user; wait for the user to enter:
    *    (a) for askLine, a string of characters;
    *    (b) for askInt, a whole number;
    *    (c) for askDouble, a decimal number;
    *  Return that value; but return "" or zero if the input is
    *  null.  Prompt again if a number is ill-formed. */

   public static String askLine (String prompt)
   {  String s = JOptionPane.showInputDialog (prompt);
      return s == null  ?  ""  :  s;
   }  //=====================

   public static double askDouble (String prompt)
   {  for (;;)
         try
         {  String s = askLine (prompt).trim();
            return s.length() == 0 ? 0 : Double.parseDouble (s);
         }
         catch (NumberFormatException e)
         {  prompt = "Ill-formed number: " + prompt;
         }
   }  //=====================

   public static int askInt (String prompt)
   {  for (;;)
         try
         {  String s = askLine (prompt).trim();
            return s.length() == 0  ?  0 : Integer.parseInt (s);
         }
         catch (NumberFormatException e)
         {  prompt = "Ill-formed integer: " + prompt;
         }
   }  //=====================
}
```

Language elements
A Statement can be: throw ObjectReferenceOfSubclassOfException
A Statement can be: try {StatementGroup} catch (ClassName VariableName) {StatementGroup}
The ClassName must be a subclass of Exception. You may have more than one catch phrase.

Exercise 10.1 Write an example of an expression using a crash-guard if-statement for ArrayIndexOutOfBoundsException and one for StringIndexOutOfBoundsException.
Exercise 10.2 Write an example of an expression using a crash-guard for NegativeArraySizeException.

Exercise 10.3 Listing 6.1 could throw a RuntimeException in some cases. Revise it with an appropriate try/catch statement.
Exercise 10.4 The part of CarRentalView that is in Listing 9.4 could throw an Exception. Handle the Exception in a reasonable way.
Exercise 10.5 If the user makes two bad entries for `askInt("Entry?")` from Listing 10.2, what are the three prompts?
Exercise 10.6* Write a test program that contains `for(;;)` and a statement after the body of that for-statement. What does the compiler say when you try to compile it? What about `while(true)`? What about `while(2+2==4)`?
Exercise 10.7* Write a version of `askInt` that has two additional int parameters named `min` and `max`. The method is to require the user to repeatedly enter a whole number `n` until it satisfies `min <= n <= max`.

10.3 Basic Text File Input; Handling Checked Exceptions

Several programs developed in this book get their input from a file, not from the user at the keyboard. First you need a Reader object. You can get one by either of these two constructor calls, which create an object of a subclass of the Reader class:

- `new FileReader ("name.txt")` connects to a disk file named "name.txt".
- `new InputStreamReader (System.in)` connects to the keyboard.

Next you need a method to read one line at a time from the input source. The BufferedReader class in the standard library provides a `readLine` method that does this; it returns null when it reaches the end of the file. The BufferedReader constructor takes a Reader object as the parameter. For instance, the following is an example of coding that counts the number of lines in a file with the name `filename`:

```
BufferedReader fi = new BufferedReader
                        (new FileReader (filename));
int lines = 0;
while (fi.readLine() != null)
   lines++;
```

The BufferedReader and InputStreamReader classes are subclasses of the Reader class, and FileReader is a subclass of InputStreamReader. All of these classes are in the `java.io` package, so you must have `import java.io.*;` above a class definition that uses these Reader subclasses.

The FileReader constructor can throw an **IOException** if the file is not there or if the file is protected against the program reading from it. And the `readLine` method can throw an IOException if the file is corrupted. This is an Exception that you have to explicitly handle it in your coding. We next develop the Buffin class, designed to isolate the exception-handling and to give an easy-to-remember way of constructing file objects. For instance, you can replace the first statement in the code segment just shown by the following, to get the same result without needing explicit exception-handling:

```
Buffin fi = new Buffin (filename);
```

The logic of the Buffin constructor

The Buffin constructor is in Listing 10.3, together with the private field variables it needs. When you call the Buffin constructor with a String value, it tries to connect to the physical hard-disk file of the given name. If that try fails (which it will if the String value is the empty string), the FileReader constructor throws an IOException (specifically, a

FileNotFoundException). But the try/catch statement in the private `openFile` method deftly catches the throw and connects to the keyboard instead. Either way, the `openFile` method returns to the Buffin constructor an object from a subclass of Reader.

Listing 10.3 The Buffin class of objects

```
import java.io.*;

public class Buffin extends BufferedReader
{
   private static boolean temp;
   ////////////////////////////////
   private boolean isKeyboard;


   /** Connect to the disk file with the given name.  If this
    *  cannot be done, connect to the keyboard instead. */

   public Buffin (String filename)
   {  super (openFile (filename));
      isKeyboard = temp;
   }  //=====================

   private static Reader openFile (String filename)
   {  try
      {  temp = false;
         return new FileReader (filename);  // IOException here
      }
      catch (IOException e)
      {  temp = true;
         return new InputStreamReader (System.in);
      }
   }  //=====================

   /** Read one line from the file and return it.
    *  Return null if at the end of the file. */

   public String readLine()
   {  if (isKeyboard)
      {  System.out.print (" input> ");
         System.out.flush();  // flush the output buffer
      }
      try
      {  return super.readLine(); // in BufferedReader
      }
      catch (IOException e)
      {  System.out.println ("Cannot read from the file");
         return null;
      }
   }  //=====================
}
```

The call of `super` then passes that Reader object to the BufferedReader constructor that accepts a Reader object; so the Buffin constructor in effect calls `new BufferedReader (someReader)`. The `isKeyboard` attribute of the Buffin object records whether the Reader object is the keyboard or a hard disk file.

The shenanigans with `isKeyboard` and `temp` in the Buffin class are probably puzzling. Why not assign to `isKeyboard` directly? The problem is, the `openFile` method cannot do so. The `this` object being constructed does not exist until after the return from the `super` call, therefore its instance variable `this.isKeyboard` does not exist either. So instead we store the value in `temp` until such time as `this.isKeyboard` exists.

You could legitimately wonder why a separate `openFile` method is needed. Why not have the body of the constructor be `try {super (new FileReader (filename));}...`? The reason is that the very first statement in a constructor has to be the `super` call; it cannot be a try/catch statement.

The logic of the readLine method in Buffin

The Buffin `readLine` method overrides the BufferedReader `readLine` method. It first checks whether input is coming from the keyboard. If so, it prompts with the > symbol, so the user knows the system is waiting for something to be typed. The `flush` method call is used to coordinate between input and output; it makes sure that the prompt appears on the screen before the user enters the input.

The Buffin `readLine` method then calls BufferedReader's `readLine` method to read one line of characters from the stream and returns it. If the input is `null`, that means that the end of the file has been reached. If that throws an IOException, particularly if you are at the end of the file, the try/catch statement in the `readLine` method deftly catches the throw, prints an error message on the screen, and returns `null`.

Technical Note If you want to write information to a disk file, the simplest way is to use `System.out.println` messages and re-direct the output. For instance, the output from a program named X will be re-directed to a file named `answer.txt` if you start the program running with the following command in the terminal window:

```
java X > answer.txt
```

Handling checked Exceptions

The Exception class is the superclass of all the various Exception types. The standard Sun library has the following definition:

```
public class Exception extends Throwable
{  public Exception()
   {  super();
   }  //=====================
   public Exception (String s)
   {  super(s);
   }  //=====================
}
```

The string of characters in the second constructor is a message that describes the kind of error that occurred. The **Throwable** class, the superclass of the Exception class, has a `getMessage` method that returns that String value. So if `e` is the Exception parameter of a catch-block, you can use `e.getMessage()` within a catch-block to access the message.

The subclasses of the Exception class that you are most likely to encounter during execution of a program are RuntimeException, InterruptedException, and IOException. An **IOException** can occur when you attempt to open a file of a given name and no such file exists, or when you try to write a value to a file that is locked against changes, or when you try to read something more from a file after you have already read everything in

it. An **InterruptedException** can occur when a thread is put to sleep (discussed later in this chapter).

Checked Exceptions

All subclasses of Exception that are not RuntimeExceptions are called **checked Exceptions**. That includes the IOException class. The general rule is that the compiler requires a method to have logic that does something about an Exception if:

- That method calls a method that can throw a checked Exception, or
- That method contains a statement that can throw a checked Exception.

That method must either have a try/catch statement that handles the Exception that might be thrown or have the **throws clause** `throws SomeException` at the end of its heading. The SomeException specified in the heading must be the kind of checked Exception you are to handle or else a superclass of it.

When you write a method with a throws clause, you are passing the buck, i.e., throwing the checked Exception at whatever method called your method. This is often called "propagating the Exception." Then that calling method has to have logic that does something about it, again one of the two options just mentioned. If your method is the main method, however, the program crashes after printing the error message.

An example of throwing an Exception further is in the following independent class method, assuming that a call of `printOneItem` can throw a PrinterAbortException (when the user cancels the print job prematurely):

```
public static void printStuff (Object[] toPrint)
            throws java.awt.print.PrinterAbortException
{  for (int k = 0;  k < toPrint.length;  k++)
      printOneItem (toPrint[k]);
}  //=====================
```

You may put a `throws` clause in the heading of a method that throws a RuntimeException if you think it is clearer. But a method that calls it is still not required to handle it, since it is not a checked Exception.

Language elements
A MethodHeading can have this phrase at the end: throws ClassName
The ClassName must be a subclass of Exception.
You may list several ClassNames in the throws clause, separated by commas.

Exercise 10.8 If you call the Buffin constructor to open a file that cannot be opened, you get the keyboard instead. You cannot easily tell whether this happened. Write a Buffin method `public boolean checkError()` that returns `true` if that happened and `false` if it does not (this is the way the PrintWriter class handles exceptional cases).
Exercise 10.9 Revise the `printStuff` method to catch a PrinterAbortException, print the message it contains, and immediately terminate the method.
Exercise 10.10 Revise the `printStuff` method to catch a PrinterAbortException, print the message it contains, and continue printing the rest of the values.
Exercise 10.11* The BufferedReader class has a constructor with a Reader parameter which is used by the Buffin constructor. How would the Buffin methods have to change if BufferedReader did not have such a constructor, but instead had two constructors, one with a FileReader parameter and the other with an InputStreamReader parameter?
Exercise 10.12* Revise Listing 10.3 so that the instance variables are `isKeyboard` and a BufferedReader object, and Buffin does not extend anything but the Object class.
Exercise 10.13* Search the java documentation to find two new kinds of checked Exceptions. Describe the circumstances in which they are thrown.

10.4 Throwing Your Own Exceptions

Suppose you are to develop a method that is to process its input, including parameters and the state of its executor, in a certain way. However, certain input values can make proper processing impossible. This method is called from several different places, and the way the exceptional situation should be handled depends on the place from which your method is called.

This is often the situation when you are writing a method as part of a library class that will be used by several programs that others have written or will write in the future. In this kind of situation, you should usually throw an Exception. That definitely solves your problem of how to handle the special situation, because throwing an Exception terminates abruptly all processing of the method that throws it. So all further processing within your method can be done with the knowledge that the exceptional case has disappeared.

Your first consideration is whether the troublesome situation could have been easily guarded against by the method that called your method. If so, you should normally throw a RuntimeException; if not, you should normally throw a checked Exception. The difference is this: If you throw a checked Exception, any method that calls your method must have a try/catch statement to handle it (or pass it higher), even if the calling method guarded against the exceptional case and so could not possibly cause an Exception.

For example, Java has a statement that pauses the current thread of execution for a given number of milliseconds (Threads are discussed in detail at the end of this chapter):

```
Thread.sleep (300)
```

This causes a pause of three-tenths of a second. If you are running several programs at once on your computer, this `sleep` command frees up the computer chip for those other threads of execution. By contrast, a busywait using a looping statement that executes while doing nothing ties up the computer chip. Two examples of busywait are the following loops:

```
while (System.currentTimeMillis() < longTime)
{  }
for (int k = 0;  k < 5000000; k++)
{  }
```

The `Thread.sleep` command can throw a checked Exception, namely, an **InterruptedException**. This happens when some other thread of execution within your program "wakes up" your thread before the specified time is up. You are forced to have a try/catch statement for this Exception even when your program does not have any other threads of execution:

```
try
{  Thread.sleep (someMilliseconds);
}
catch (InterruptedException e)
{  // no need for any statements here
}
```

It would be much pleasanter to be able to write just one line here instead of six. But you cannot because InterruptedException is not a RuntimeException. It was set up this way because, if you have a program with several threads of execution, and you put one of them to sleep, there is usually no way for you to guard against one of those other threads interrupting your sleeping thread. By contrast, you can easily guard against a NullPointerException with a reasonable amount of care.

```
java.lang.Exception
     java.lang.InterruptedException
     java.io.IOException
          java.io.EOFException
          java.io.FileNotFoundException
     java.io.print.PrinterException
          java.io.print.PrinterAbortException
     java.lang.RuntimeException
```

Figure 10.2 Partial listing of the Exception hierarchy

How to throw an Exception

The simplest way to throw an Exception and avoid a troublesome situation is to use one of the following two statements (the first when you want a checked Exception):

```
throw new Exception (someExplanatoryMessage);
throw new RuntimeException (someExplanatoryMessage);
```

The explanatory message is a String value. Almost all subclasses of Exception have two constructors, one with a String parameter and one not. You use the one without the parameter if you feel that the name of the Exception is enough information. In a catch-block, you may call `e.getMessage()` to retrieve the message that `e` has, assuming `e` is the name you choose for the Exception parameter of the catch-block.

For example, the Worker constructor in Listing 7.6 returns a worker with a `null` name if the input String `s` is `null`. It could instead throw a checked Exception as follows:

```
if (s == null)
   throw new Exception ("The string does not exist");
```

The constructor would therefore have to have the phrase `throws Exception` at the end of its heading as follows. Then any method that calls it must handle the Exception:

```
public Worker (String s) throws Exception
```

The main methods in Listing 7.4 and Listing 7.7 call that constructor, so they would also have to have revised headings, unless you rewrite the logic with a try/catch statement:

```
public static void main (String[] args) throws Exception
```

If you wanted a WorkerList to have a method that returns the last Worker in the list, it could be written to throw an unchecked Exception for an empty list as follows:

```
public Worker getLastWorker() throws RuntimeException
{  if (itsSize > 0)
      return itsItem [itsSize - 1];
   else
      throw new RuntimeException ("The list is empty!");
}  //=====================
```

Note that this logic does not execute a return statement in both branches. You are only required to execute either a `return` or a `throw` statement in all branches of an if-statement at the end of a method that is to return a value. The phrase `throws RuntimeException` is optional in the heading of this revised `getLastWorker` method, because it is an unchecked Exception.

Creating a subclass of Exception

You could create your own class of Exceptions if you wish. You should subclass either Exception or RuntimeException, depending on whether you want it checked. For instance, the last statement of the `getLastWorker` method just given could be either `throw new BadPersonException("The list is empty!")` or `throw new BadPersonException()` if you have the following class definition:

```
public class BadPersonException extends RuntimeException
{  public BadPersonException()
   {  super();
   }  //=====================
   public BadPersonException (String message)
   {  super (message);
   }  //=====================
}
```

You are allowed to extend other subclasses of Exception when you define your own Exception class. But it is simplest to make all the Exception classes you define just like the BadPersonException class just shown, with two possible differences: (a) replace "BadPerson" by whatever you want in three places, and (b) either keep or omit "Runtime" in the heading.

Exercise 10.14 Define a BadStringException class that is a checked Exception.
Exercise 10.15 Revise Listing 6.4 to have the `trimFront` and `firstWord` methods throw a BadStringException when the String value has zero characters.
Exercise 10.16 Revise the `yearsToDouble` class method in Listing 6.1 to throw an ArithmeticException when the interest rate is not positive. Modify the method heading as needed.
Exercise 10.17* Revise the `parseDouble` method of Listing 6.5 to throw a BadStringException when the first non-whitespace character other than '-' or '-.' or '.' is not a digit.

10.5 Analysis, Test Plan, And Design For The Investor Software

Further discussion with the investors reveals that they have IRA accounts in addition to their 401(k) plans. They have more flexibility with these IRA accounts; they want to know whether it is profitable to trade frequently (perhaps daily) in them. Since they have day jobs, they would only be able to make buy or sell decisions once a day, in the evening. And since they do not have enough in the accounts to make it worthwhile to buy individual stock holdings, the IRAs are also invested only in mutual funds.

The investors may start regular monthly additions to the 401(k) accounts and IRA accounts in perhaps 15 years, to build up their retirement funds in the five years or so before they retire. And once they retire, they expect to have regular monthly withdrawals for living expenses. So your software needs to allow for daily price changes of assets and a fixed-amount addition to or subtraction from a portfolio of mutual funds each month. According to the clients, each deposit is to be distributed among the mutual funds proportionally to the current holdings. Similarly, each withdrawal is to be taken from the mutual funds proportionally.

The way a mutual fund works, money market or otherwise, is as follows: The mutual fund buys or sells some securities every day as investors move money into or out of the fund. For instance, if the fund has $500 million at the end of one day, and an investor has $50,000 invested in that fund, then that investor owns 0.01% of the assets of that fund as of the close of business that day. If the fund's assets have a value of $505 million at the end of the next day, the fund managers take out say $10,000 for their expenses and salaries and profits, leaving $504,990,000 net value in the mutual fund. The $4,990,000 of profit belongs to the individual investors in proportion to what each owned. In particular, the person who had $50,000 in the fund the day before sees the value of the investment go up by $499.

Mutual funds change in price once each trading day, conventionally at 4 p.m. when the normal markets close, and a year has roughly 252 trading days, allowing for Wall Street holidays. You will need an algorithm for simulating the change in value for the day, which will give a different result for each of the mutual funds. A simulation is needed because no one knows what the markets will do, in terms of prices of investments rising and falling. "Simulation" in this case means educated guesswork.

The clients have no clear idea of how the change in price each day should be modeled, except that random outcomes are involved. Bill the expert works it out with them that he will do some research on historical trends, develop good formulas for this calculation with random numbers, then check back with the clients with some proposals they can accept or modify. He will give you a very rough approximation that you can use until he finishes working this out with the clients. So until the correct algorithms are determined, you are essentially developing a prototype of the program.

Average returns

If an investment for a 3-year period has e.g. annual returns of 7%, 31%, and -13%, respectively, then the total value at the end of the three years is calculated by first converting to the annual multipliers of 1.07, 1.31, and 0.87, then multiplying them together to get 1.22; therefore, the total 3-year return is 22%. In general, we use these annual multipliers in calculations rather than the vernacular percentages (note that adding the percentages and getting the total of 25% does not give us any meaningful value).

An **average return** is not found by dividing the 22% (let alone the 25%) by 3. Instead, you figure out what multiplier you could have used every year to get the same result. Since 106.85% to the third power is 122%, the average return is 6.85%. This way of calculating an average is called the **geometric average**.

Tax-free growth

An IRA or 401(k) plan offers investment growth of your after-tax investment essentially free of income taxes (except for a non-deductible IRA). They require that the money invested come from earned income (rather than e.g. interest earnings) and that you do not withdraw it until you are close to retirement age. There are two basic kinds, Roth IRA and deductible IRA, depending on whether the taxes on the money invested are paid when you deposit the money or when you withdraw it.

Suppose you can spare $700 to invest in after-tax money, and you invest it in stocks using an IRA when your combined federal and state income tax rate is 30%. If you let the money grow for 25 years before you spend it, it could reasonably be expected to increase by a factor of perhaps 10. The two main kinds of tax shelters are:

- You can put the $700 in a Roth IRA. It grows to about $7,000, all of which you get to spend tax-free.
- You can put $1000 in a deductible IRA or a 401(k) or a 403(b) and get $300 back immediately on your income taxes. It grows to about $10,000, of which $3,000 is due to the $300 deferred taxes and $7,000 is due to your after-tax cost of $700. When you cash it in and pay taxes at the 30% combined tax rate, you simply give the government its $3,000 and you keep the $7,000 that your $700 became.

Either way, the out-of-pocket cost to you is the same, namely, $700. And the end result is the same, namely, $7,000. Your $700 out-of-pocket cost grows tax free either way. The only difference is when you give the government its due.

There are other considerations that indicate what blend of Roth and deductible tax-shelters you should have. A crude approximation is that money on which you expect to pay tax at a substantially lower rate when you retire should be in a deductible tax-shelter, and all other money should be in a Roth IRA. The reasons are (a) if the tax rate is lower, you get to keep part of the government's $3,000 (e.g., if your combined rate in retirement is only 20%, your taxes are $2,000, so you get an extra $1,000); (b) the Roth IRA has better features than a deductible IRA in most other respects.

Planning the tests

Bill the expert tells you that a particular mutual fund will have two multipliers to describe its long-term behavior. One is its daily average and the other is its daily volatility. The daily volatility measures how much fluctuation there is in the day-to-day returns. These two values for the money market could be 1.0002 (averaging somewhat more than 5% annually) and 1.0001 (very low volatility).

The simplest test of the software is to put $10,000 in just one mutual fund and let it ride for 20 years. Calculate by hand what the **expected result** would be, which is the daily average multiplier to the power 252*20 ("by hand" means with a spreadsheet or calculator, of course). Run the test 10 or 20 times, then check that you get values distributed somewhat evenly around the expected result. Do this for at least three of the available mutual funds.

A second test is to split $10,000 evenly among the five mutual funds, $2,000 each. At the end of each month (which is 21 trading days), **rebalance** whatever money you have as 20% in each of the five mutual funds. Repeat this for 20 years. Run this test 10 or 20 times, then check that the values are distributed somewhat evenly around the expected result. The **expected result** is found as follows:

1. Calculate each of the five daily average multipliers to the power 21 to get the five monthly averages;
2. Find the arithmetic average of those five monthly averages;
3. Raise it to the power 12*20 (the total number of months).

Each of those tests can be redone with the regular addition of say $200 per month to the account. In the case of the rebalancing test, the $200 would be split evenly among the five mutual funds when the deposit is made. Then the tests can be done again with the regular subtraction of say $50 per month.

Design of the Investor software

The overall logic of the Investor program is fairly simple:

1. Create a portfolio of five mutual funds with the initial allocations that the user specifies.
2. Allow many days to pass, updating as needed for market changes and the user's decisions.

You need to break this logic down into separate steps. When the program begins operation, it should print out a summary of what each of the five asset classes is. Then it should find out how much the user has in the account and how it is distributed. It is probably simplest to first add the amount to the money market account and then ask the user to reallocate it among the five asset classes. The program should also ask how much the user is adding to the account each month.

The program could ask the user each day for the desired allocation among the five asset classes, then report the value in each asset class one day later, then repeat the process. But that means `252*20` inputs from the user for a 20-year period. This is surely unacceptable to the user. At a very optimistic 3 seconds per input, that is still over 4 hours for one run of the program.

It would be far better to let the user say how many days to let the money ride before a new decision about allocations is made. Then the program can calculate the change in value of each asset class over that many days, report those values, then repeat the process. That way, if a speculator wants to make allocation decisions once a week, the waiting period can be 5 trading days. If an investor wants to make allocation decisions once each quarter, the waiting period can be 63 days (one-fourth of the 252 trading days per year).

The program should make it very easy to give a default allocation of leaving the money ride by just pressing the ENTER key, and a default number of trading days to wait also by pressing the ENTER key. Probably the most useful default number of trading days is whatever the investor chose the previous time. You go back to the client to see if this is acceptable (sometimes revisions in the analysis come during the test-plan and design steps). You rework this preliminary design as shown in the accompanying design block.

The object design

You see that there are three kinds of objects mentioned in the top-level design: a single asset class, a portfolio consisting of five asset classes, and the user. You look carefully at what each object has to be able to do -- its operations. The user will give you the input and receive the output, which is what `IO.askDouble` and `IO.say` do. That is, the IO class from Listing 10.2 can be in essence the virtual user.

STRUCTURED NATURAL LANGUAGE DESIGN of the main logic

1. Create a portfolio of the 5 asset classes with no money in it.
2. Describe each one of the 5 asset classes (name, bad year, average year, good year).
3. Ask the user for the initial balance and the monthly deposit.
4. Repeat the following until the user says to stop...
    - 4a. Ask the user whether assets are to be reallocated; if yes, then...
        Find out what the reallocations are and make them.
    - 4b. Ask the user how many days to wait until the next reallocation decision.
    - 4c. Change the amounts in the five asset classes for that many days.
    - 4d. Display the amounts in each asset class at the end of that period.
    - 4e. Ask if the user wants to stop this processing loop.
5. Terminate the processing.

Preliminary designs of the other two kinds of objects, derived from the main logic design, are in Listing 10.4 (see next page) as classes with stubbed methods. The methods for the Portfolio class are designed to match up with the steps in the main design fairly closely. The methods in the Asset class follow from the Portfolio methods:

- To construct a Portfolio object you must construct several Asset objects.
- To describe a whole Portfolio object you must describe the individual Asset objects.
- To let a number of days pass, you must let one day pass at a time for each Asset object.

The Asset class is to be used only with a Portfolio class, so the Asset class is not declared as a public class. This allows the convenience of putting it in the same `Portfolio.java` file. A text file to be compiled in Java can contain several separate classes, as long as only one is public (other than classes inside other classes); the name of that public class has to be the name of the text file.

The constant values for one month of trading days and one year of trading days will be used in the Asset class (to convert between daily and annual multipliers), and in the Portfolio class (for calculations involving the monthly deposit), as well as probably the program itself. Since the program uses the Portfolio class which uses the Asset class, it makes sense to put these constants in the Asset class.

Exercise 10.18 If you let an investment ride for three years, and it makes 50% each of the first two years and loses 50% the third year, what is your average return?
Exercise 10.19 How much would $700 grow to after 20 years if it earned 10% every year in a Roth IRA? (use a spreadsheet or calculator for this exercise and the next).
Exercise 10.20 How much would $700 grow to after 20 years if it earned 10% every year in an investment subject to 30% taxes each year? What is the ratio of your answer to that of the preceding exercise?
Exercise 10.21 If you invest your IRA in a way that earns 7% every year for 13 years, and Jo makes a taxable investment on which tax is paid each year at a 30% combined tax rate, at what rate does Jo have to earn in order to keep up with you in after-tax value? (You can do this in your head).
Exercise 10.22* For the preceding problem, call the answer X%. If Jo invested in something with a fluctuating annual return averaging X%, would Jo keep up with you? How do you know? If Jo's annual return were fixed at X% and yours fluctuated but still averaged 7%, would you keep up with Jo?
Exercise 10.23* Do some research to find out why non-deductible IRAs and annuities are far worse than either deductible IRAs or Roth IRAs.

Listing 10.4 Description of the Portfolio and Asset classes

```
public class Portfolio extends Object     // stubbed version
{
   /** Construct portfolio with five standard asset classes. */
   public Portfolio()                                          { }

   /** Print description of all available investment choices. */
   public void describeInvestmentChoices()                     { }

   /** Add the given amount to the Money Market fund.  */
   public void addToMoneyMarket (double amountAdded)           { }

   /** Accept a reallocation of investments by the user.  */
   public void reallocateAssets()                              { }

   /** Make the given amount the monthly deposit.  */
   public void setMonthlyDeposit (double deposit)              { }

   /** Update for changes in price levels for several days. */
   public void letTimePass (int numDays)                       { }

   /** Show the values of each holding and the total value.  */
   public void displayCurrentValues()                          { }
}
//#################################################################


class Asset extends Object                 // stubbed version
{
   /** Number of trading days in one month and in one year. */
   public static final int MONTH = 21;
   public static final int YEAR  = 12 * MONTH;
   ////////////////////////

   /** Create 1 Asset object with the given daily multipliers. */
   public Asset (String name, double avg, double volatility)  { }

   /** Describe the asset to the user in one line of output, in
    *  terms of a good year, an average year, and a bad year. */
   public void describe()                                      { }

   /** Return the multiplier for the next day's market activity,
    *  e.g., 1.02 for a 2% increase. */
   public double waitOneDay()                        { return 0; }
}
```

10.6 Version 1 Of Iterative Development

The main logic begins by creating a Portfolio object. This creates the asset classes, each with its two numeric parameters. Then it prints out a description of the five asset classes. This tells the user what kind of results can be expected in good years and bad.

The main logic is overly complex, so you should have a separate InvestorAide object to carry out some of the subtasks. The initialization of this portfolio for the amount invested, and the determination of the amount of the monthly deposit or withdrawal, is a separate

well-defined task, so it makes sense to have an InvestorAide `agent` get this initial setup. This cuts down on the length of the main method, which makes it clearer.

Similarly, asking the user for one of three choices, reallocate/stop/continue, and checking the value to make sure it is acceptable, is a little complex. That task can also be delegated to the InvestorAide `agent` in a separate method. Since the InvestorAide's jobs are tailored for the Investor class, we can make it a non-public class so it can be compiled in the same file as the Investor class.

You use `IO.askInt` to ask for the number of days to wait for the next decision. `IO.askInt` returns zero if the user just presses the ENTER key. So you need to get the result of `IO.askInt` and see if it is positive. If so, you may store it in the variable that keeps track of the number of days to wait, otherwise you leave that variable with its earlier value. The initial value for this `days` variable should be 21, the number of days in one month. A reasonable implementation for the main logic so far is in Listing 10.5.

Listing 10.5 The Investor class application

```
public class Investor
{
   /** Ask the user how much money to invest and how much to
    *  add or subtract monthly.  Then process a sequence of
    *  investment decisions for as long as the user wants.
    *  Developed by Dr. William C. Jones       January 2001 */

   public static void main (String[] args)
   {  IO.say ("This program tests your success at investing.");
      Portfolio wealth = new Portfolio();
      wealth.describeInvestmentChoices();
      InvestorAide agent = new InvestorAide();
      agent.askInitialSetup (wealth);

      int days = Asset.MONTH;  // trading days in one month
      char choice = agent.askUsersMenuChoice();
      while (choice != 'S' && choice != 's')  // S means STOP
      {  if (choice == 'Y' || choice == 'y')
            wealth.reallocateAssets();
         int answer = IO.askInt ("How many days before the "
                + "next decision \n(ENTER if no change)? ");
         if (answer > 0)   // ENTER returns zero
            days = answer;
         wealth.letTimePass (days);
         wealth.displayCurrentValues();
         choice = agent.askUsersMenuChoice();
      }
      System.exit (0);
   }  //=====================
}
```

What if the user enters a negative number when asked for the number of days? That must be a mistake, and it is not mentioned in the initial design. That is a defect in the design that has to be corrected at this point. You would find it easiest to just pretend that the user pressed the ENTER key in that case. You check with the clients and they say that is acceptable. The coding in Listing 10.5 makes this adjustment.

The InvestorAide methods

The `askInitialSetup` method is sequential logic: (a) Ask for the total assets invested; (b) Put it (for now) in the money market ("cash"); (c) Ask for the amount to be deposited each month; (d) Record that deposit amount. This will be a negative number if the user is making monthly withdrawals. It will be zero if the user is making no changes.

The `askUsersMenuChoice` method gets the user's decision at the end of the chosen wait period. Begin by displaying the three choices: stop the program, reallocate assets and continue for another period, or leave the assets allocated as they are and continue for another period. If the user simply presses the ENTER key, that should be interpreted as leaving things ride for another period. Listing 10.6 has the logic for these two methods. Figure 10.3 shows the UML class diagram for the Investor class.

Listing 10.6 The InvestorAide class

```
class InvestorAide  // Helper class for Investor application
{
   /** Get the starting values for the portfolio.  */

   public void askInitialSetup (Portfolio wealth)
   {  double value = IO.askDouble ("Invest how much initially?");
      wealth.addToMoneyMarket (value);
      IO.say ("Initially it is all in the money market.");
      value = IO.askDouble ("How much do you add each month "
               + "(Use a negative number for a withdrawal)? ");
      wealth.setMonthlyDeposit (value);
   }  //=====================

   /** Get the first letter of user's choice from the menu.  */

   public char askUsersMenuChoice()
   {  String input = IO.askLine ("Enter YES (or anything "
               + "starting with Y or y) to reallocate assets; "
               + "\nEnter STOP (or anything starting with S "
               + "or s) to stop); or"
               + "\nPress ENTER or anything else to continue:");
      return  (input.length() == 0)  ?  ' '  :  input.charAt(0);
   }  //=====================
}
```

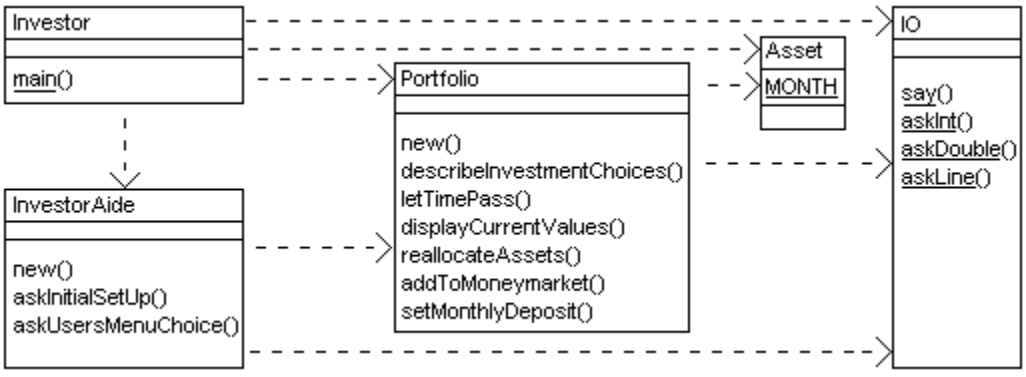


Figure 10.3 UML class diagram for the Investor program

A portfolio consists primarily of five Asset objects. All five of the Asset classes are treated the same, except that the money market is sometimes handled differently. So one Portfolio instance variable should be an array of Asset objects, with the money market stored in component zero for convenience. You also need to keep track of how much money is stored in each asset class, so another Portfolio instance variable should be an array of double values.

You could call these two variables `itsAsset` and `itsValue`. Then `itsValue[0]` is the amount of money currently invested in the money market `itsAsset[0]`, `itsValue[1]` is the amount of money currently invested in `itsAsset[1]`, etc. These two arrays are called **parallel arrays**, because the kth component of one tells you something about the kth component of the other. Parallel arrays do not often occur in Java because, more often than not, a single array of objects that combine both parts is preferable. In the present situation, however, calculations involving a single asset will probably be complex enough without involving the current value of the holding. That is, the Asset class has enough to do if it just deals with the nature of the asset; the Portfolio class can track the current values.

The Portfolio constructor initializes the array values. The `itsValue` array defaults to all zeros, but it is clearer to explicitly state this as an initializer list `{0,0,0,0,0}`. For this first version of the software, you could simply keep all the investments in the money market fund. This really simplifies the development. So you may as well use the same average return and volatility for all five investments; for now, the only difference between any two Asset objects is in the name. A typical statement in the constructor is:

```
itsAsset[2] = new Asset("large caps", 1.0002, 1.0001)
```

The second parameter is the daily average of returns (0.02% daily) and the third parameter is the daily volatility of returns (plus or minus 0.01% daily). These values are unrealistic, because different asset classes have different averages and volatilities. So these values will be changed when Bill the expert reports back with his conclusions.

Six Portfolio methods

The `describeInvestmentChoices` and `reallocateAssets` methods can be left undone for now; there is nothing to describe or reallocate in this Version 1, since all assets are kept in the money market. The `addToMoneyMarket` method simply adds the parameter value to `itsValue[0]`; the other values in that array are initially zero. And the `displayCurrentValues` method simply prints out the value of the money market.

When you consider the `setMonthlyDeposit` method, you realize that a Portfolio object needs an instance variable to keep track of this amount so it can use it every 21 days. And a Portfolio object also needs to keep track of how many days have passed since it was first created, so it knows when the next 21-day period has passed.

These Portfolio instance variables could be called `itsMonthlyDeposit` and `itsAgeInDays`. You could initialize `itsMonthlyDeposit` to zero for every newly constructed Portfolio object, indicating that no deposits or withdrawals are made monthly. To simplify Version 1, ignore the monthly deposit in the calculations. Similarly, `itsAgeInDays` can be initialized to zero in the variable declaration, since it will always be zero at the time a Portfolio object is created.

The `letTimePass` method is to find the effect of waiting a given number of days. Since all assets are in the money market, you only need to multiply the assets by the daily interest rate multiplier of 1.0002 once for each day that passes. The standard library method call `Math.pow(amt,power)` gives the exponential amt^{power}.

The Portfolio class so far is in Listing 10.7; the last four methods listed will require further work in the next version of the software. You can use the stubbed version of the Asset class in the earlier Listing 10.4 as is for this Version 1 of the software.

Listing 10.7 The Portfolio class, Version 1

```
public class Portfolio extends Object
{
   public static final int NUM_ASSETS = 5;
   ////////////////////////////////
   private Asset[]  itsAsset = new Asset[NUM_ASSETS];
   private double[] itsValue = {0,0,0,0,0};
   private double   itsMonthlyDeposit = 0;
   private int      itsAgeInDays = 0;

   public Portfolio()
   {  itsAsset[0] = new Asset ("money mkt   ", 1.0002, 1.0001);
      itsAsset[1] = new Asset ("2-yr bonds  ", 1.0002, 1.0001);
      itsAsset[2] = new Asset ("large caps  ", 1.0002, 1.0001);
      itsAsset[3] = new Asset ("small caps  ", 1.0002, 1.0001);
      itsAsset[4] = new Asset ("emerg mkts  ", 1.0002, 1.0001);
   }  //=====================

   public void addToMoneyMarket (double amountAdded)
   {  itsValue[0] += amountAdded;
   }  //=====================

   public void setMonthlyDeposit (double deposit)
   {  itsMonthlyDeposit = deposit;
   }  //=====================

   public void describeInvestmentChoices()
   {  IO.say ("No description of choices for Version 1");
   }  //=====================

   public void reallocateAssets()
   {  IO.say ("No reallocation of assets for Version 1");
   }  //=====================

   public void letTimePass (int numDays)
   {  itsAgeInDays += numDays;
      itsValue[0] *= Math.pow (1.0002, numDays);
   }  //=====================

   public void displayCurrentValues()
   {  System.out.println (itsAgeInDays + ", " + itsValue[0]);
   }  //=====================
}
```

Exercise 10.24 Revise the `askInitialSetup` method in Listing 10.6 to force the user to enter a positive number for the initial investment.

Exercise 10.25** Write an IO method `public static int askChar()` that returns the first character of the line of input, capitalizing it if it is a lowercase letter. It should return a blank if the user simply presses the ENTER key. Then rewrite the logic of the investor classes to use this method appropriately.

10.7 Version 2 Of Iterative Development

You should not try to do too much at one time when developing the implementation of a program:

- Set a reasonable target that will give meaningful results when the program is run.
- Develop the logic to reach that target, which is Version 1 of the Iterative Development.
- Test the program thoroughly to make sure it does what it is supposed to do at that point.
- Repeat this partial development and testing through several versions until the program is complete.

This program has so much complexity already that you should try for the minimum possible additional logic in Version 2 to give a useful result. So you decide to have Portfolio's `letTimePass` method simply add `itsMonthlyDeposit` to the money market fund. You also ignore negative asset values. Now the logic can be worked out as shown in the accompanying design block.

DESIGN for the letTimePass method

1. For each of the `numDays` days do...
    - 1a. Get that day's multiplier from each Asset object and multiply it by the current value in that asset class.
    - 1b. Add 1 to `itsAgeInDays`.
    - 1c. If `itsAgeInDays` is a multiple of 21, then...
        - 1cc. Add the amount of the monthly deposit to the value of the money market fund.

The displayCurrentValues method

The user will want to be able to look at the status of the holdings for the last few times a reallocation was made. To have so much data on the screen at once, you have `displayCurrentValues` print to the terminal window. This method begins by printing a title for the column of five lines and then printing out the name and current value of each of the five asset classes. This makes you realize that you need to add a `getName()` method to the Asset class. The values printed should be in columns, so the tab character `'\t'` is needed.

Also, investors will not be interested in the amount to the exact penny (or beyond), since they have several thousand dollars in each investment. So the values should be rounded to the nearest dollar. The standard way to do that is to add 0.5 and then truncate with the `(int)` cast. The coding for the displayCurrentValues method is in the middle part of Listing 10.8 (see next page).

The describeInvestmentChoices method

The `describeInvestmentChoices` method calls `itsAsset[k].describe()` for each of the Asset objects in order, after giving an appropriate heading. The description of an Asset object should include the possible outcomes of investing in that asset: what one could expect in a good year and in a bad year, as well as in an average year.

Listing 10.8 Revisions of four Portfolio methods for Version 2

```
   /** Update the status for changes in price levels for
    *  several days. */

   public void letTimePass (int numDays)
   {  for (;  numDays > 0;  numDays--)
      {  for (int k = 0;  k < NUM_ASSETS;  k++)
            itsValue[k] *= itsAsset[k].waitOneDay();
         itsAgeInDays++;
         if (itsAgeInDays % Asset.MONTH == 0)
            itsValue[0] += itsMonthlyDeposit;
      }
   }  //=====================

   /** Show the values of each holding and the total value.  */

   public void displayCurrentValues()
   {  System.out.println ("\nname \t\tcurrent value");
      for (int k = 0;  k < NUM_ASSETS;  k++)
         System.out.println (itsAsset[k].getName()
                             + "\t$" + (int) (itsValue[k] + 0.5));
   }  //=====================

   /** Print description of all available investment choices.  */

   public void describeInvestmentChoices()
   {  Asset.printHeading();
      for (int k = 0;  k < NUM_ASSETS;  k++)
         itsAsset[k].describe();
   }  //=====================

   /** Accept a reallocation of investments by the user.  */

   public void reallocateAssets()
   {  IO.say ("How will you invest your current assets?"
              + "\nUnallocated amounts go in a money market.");
      for (int k = 1;  k < NUM_ASSETS;  k++)
      {  double amount = IO.askDouble ("How much in "
                         + itsAsset[k].getName() + "? ");
         itsValue[0] += itsValue[k] - amount;
         itsValue[k] = amount;
      }
      IO.say ("That leaves $" + (int) (itsValue[0] + 0.5)
                    + " in the money market.");
   }  //=====================
```

The Asset class "knows" how many values it will display and what they mean, and this may change in future revisions. So you should ask the Asset class to provide appropriate headings for the columns of values. You need to add a `printHeading` method to the Asset class for this purpose. This method should be a class method, since the headings do not depend on any particular Asset object's status. That is, you are asking a question of the Asset class as a whole, not of any particular Asset object.

The reallocateAssets method

For the `reallocateAssets` method, go through each of the asset classes one at a time, skipping the money market fund (cash is special). For each non-cash asset class,

ask the user for the amount that is to be in it. Then replace `itsValue[k]` by that amount (`k` ranging 1 through 4). You also need to add to or subtract from the money market fund the amount of change in `itsValue[k]`. Listing 10.8 gives Version 2 of these four methods for the Portfolio class.

Note that you are changing the object design, which you may have thought was complete several stages ago. This just means that the Asset class is required to provide certain additional services that were not apparent from the main logic alone.

Exercise 10.26 Modify Portfolio's `letTimePass` method to charge 5% higher interest when the money market balance is negative. This roughly corresponds to interest rates on margin loans. Hint: You need to add `0.05 / Asset.YEAR` to get 5% annual interest compounded daily.
Exercise 10.27* Modify Portfolio's `displayCurrentValues` method to print out with the heading, "After Y years, M months, and D days:" using the correct values for Y, M, D.
Exercise 10.28** Modify Portfolio's `letTimePass` method to distribute the monthly deposit proportionally over all five asset classes (e.g., if large-caps are 40% of the whole and the deposit is $200, put $80 in large-caps).
Exercise 10.29** Modify Portfolio's `reallocateAssets` method to charge 1% of all assets sold other than money market assets. This approximates the real-life cost of buying and selling investments, for commissions and the like. Debit the money market fund for this cost. Example: If the only change is for small caps to go up from $30,000 to $40,000 and for large caps to go down from $50,000 to $44,000, charge 1% of the $6,000 sale. So the money market fund will go down by a total of $4,060.

10.8 Version 3 Of Iterative Development

When you look at what an Asset object has to be able to do, you can deduce what it needs to know: `itsName`, `itsReturn`, and `itsRisk`. Those are the obvious instance variables used to calculate daily changes. But when asked to print the descriptions of an asset, the user does not want to see daily returns and volatility. The user wants an indication of what the return would be in a good year, what it would be in a bad year, and what it would be in an average year. So the `describe` instance method should print `itsName` and those three numbers calculated from `itsReturn` and `itsRisk`. The `printHeading` class method should print four corresponding column titles.

Question: How do you calculate a good year and a bad year? Since the sequence of daily returns is a set of random numbers, theory of statistics can provide the answers. However, its answers for the distribution of values from a random distribution are for cases where you have the sum of random values. They do not directly apply to the product of random values.

Solution: Do a statistical analysis on the logs of the random values. The result for one year of 252 days is the sum of the 252 random daily values if they are expressed in logarithms. The **Central Limit Theorem** says that the potential values for a sum of many random values are extremely close to a normal distribution (the notorious bell-shaped curve). Specifically, the distribution of the sum of 252 random daily values will be very close to a normal distribution that has a mean of 252 times the daily mean and a standard deviation of 15.87 times the daily standard deviation (because the square root of 252 is 15.87).

Suppose Bill the expert tells you that the two multipliers that describe the long-term behavior of a certain kind of stock are the **daily average** of 100.04% and the **daily volatility** of 101%. This daily average means that you multiply the previous day's value by 100.04% to get the new day's value if it is an average day. Over a year of 252 days,

that comes to 10.6% gain, since 1.0004^{252} is 1.106 (you can check this in a spreadsheet with the expression `power(1.0004,252)` or just `1.0004^252`).

But mutual funds go up and down in value; they do not always go up by a fixed amount. The daily volatility takes this into account. For a good day in the market, multiply the average day's result by 101%; for a bad day, divide it by 101%. Bill says you should (for now) approximate market uncertainty by calculating either a good day or a bad day at random with equal likelihood. This means that you either add or subtract `log(1.01)` to `log(1.0004)` to modify the previous day's result, so `log(1.01)` is the standard deviation of the daily returns (when everything is expressed in logs).

To simplify the calculations, you should store the logarithms of daily averages and volatilities rather than the numbers themselves. The return for N days is just the sum of the N returns. And the variability in returns for N days is `Math.sqrt(N)` times the variability for one day. That is, we can apply standard statistical analysis to logarithms of returns. Since the square root of 252 is 15.87, the annual volatility according to the Central Limit Theorem is $1.01^{15.87}$, which is 1.1711, or 17.11% in everyday terms.

Math methods

This program uses four class methods from the standard library Math class that are described in Chapter Six:

- `Math.sqrt(x)` gives the square root of a non-negative number `x`;
- `Math.pow(x, n)` gives x to the nth power;
- `Math.log(x)` gives the logarithm of a positive number `x`; and
- `Math.exp(x)` gives the antilog of a number `x`.

In particular, `Math.exp(Math.log(x))` is the same as `x` for any positive number `x`, and the logarithm of the product of two numbers is found by adding the logarithms of the two individual numbers. The Asset class also uses a random-number-producing object from the `java.util.Random` class: `random.nextInt(n)` is one of the equally-likely numbers in the range 0 to n-1 inclusive.

The Asset constructor takes the three values passed in as parameters and stores them in its three instance variables, first converting the return and risk to logarithms. The `getName` method has the standard logic, and the `waitOneDay` method returns one of two results with equal (50%) probability: The market goes 1% higher or 1% lower than the daily average, which in the very long run produces something relatively close to the daily average (according to the Law of Large Numbers).

The describe method

In a normal distribution, only 16% of the time do you get a value more than one standard deviation above the mean. A good year can be considered to be one standard deviation above the mean, because the market is that good (or better) only one year in six. Similarly, a bad year can be considered to be one standard deviation below the mean, because something that bad (or worse) happens only one year in six. One standard deviation is `itsReturn * Math.sqrt(YEAR)`, which is the (log) daily return times the square root of the number of observations. These Asset methods are in Listing 10.9 (see next page).

The clients say they would like to see expected returns from investments as annual percentage rates rounded to the nearest tenth of a percent. For instance, a multiplier of 1.0317 should be expressed as 3.2%, and a multiplier of 1.03148 should be expressed as 3.1%. The standard way to do this is as follows:

Listing 10.9 The Asset class, Version 2

```
import java.util.Random;   // goes at the top of the file

class Asset extends Object
{
   public static final int MONTH = 21;
   public static final int YEAR = 12 * MONTH;
   private static Random random = new Random();
   ////////////////////////////////
   private String itsName;
   private double itsReturn;
   private double itsRisk;


   public Asset (String name, double average, double volatility)
   {  itsName   = name;
      itsReturn = Math.log (average);
      itsRisk   = Math.log (volatility);
   }  //======================

   public String getName()
   {  return itsName;
   }  //======================

   public static void printHeading()
   {  System.out.println ("name\tgood year\tavg year\tbad year");
   }  //======================

   public void describe()
   {  double rate = itsReturn * YEAR;
      double risk = itsRisk * Math.sqrt (YEAR);
      System.out.println (itsName + toPercent (rate + risk)
               + "%\t" + toPercent (rate)
               + "%\t\t" + toPercent (rate - risk) + "%");
   }  //======================

   private double toPercent (double par)
   {  return ((int)(1000.0 * Math.exp(par) + 0.5) - 1000) / 10.0;
   }  //======================

   public double waitOneDay()
   {  return random.nextInt (2) == 0   // 50-50 chance
               ?  Math.exp (itsReturn + itsRisk)
               :  Math.exp (itsReturn - itsRisk);
   }  //======================
}
```

1. Multiply the number to be rounded by 1000, e.g., 1.0317 -> 1031.7, 1.03148 -> 1031.48. Use 1000 because you want it rounded in the third decimal place.
2. Add 0.5, which pushes XX.5 or higher to the next higher place but leaves the other cases the same in the part before the decimal point, e.g., 1031.7 -> 1032.2, 1031.48 -> 1031.98.
3. Truncate with the `(int)` cast, e.g., 1032.2 -> 1032, 1031.98 -> 1031.
4. Subtract 1000 and then divide by 10.0 to get the percentage accurate to tenths, e.g., 1032 -> 3.2, 1031 -> 3.1.

A private `toPercent` method in the Asset class accepts a number as input, "unlogs" it, and makes the conversion just described, so it is in a form ready to print. This completes Version 2 of the software.

The bell-shaped curve is not a close approximation for distributions of sums of less than about thirty random values. But it is worth seeing how close it comes for just five values (since you can calculate those values in your head). `Math.sqrt(5)` is roughly 2.24, so somewhat more than 16% of the sums of five values randomly chosen from {+1, -1} should theoretically be higher than 2.24 (one standard deviation). A 5-day result has 1 chance in 32 of going up every day (since 2^5 is 32), and a 5-in-32 chance of going up on 4 of the 5 days (and thus down on the other day). So there is a 6-in-32 chance of going up by 3 or more of `itsRisk`. And 6-in-32 is roughly 19%. Conclusion: There is a roughly 19% probability of a week going up by at least 1 standard deviation. Similarly, there is a roughly 19% probability of a week going down by at least 1 standard deviation. 19% is reasonably close to 16%.

Bill the expert has finished his research into market performance and has three results for you. First, assuming a long-term fairly steady inflation rate of 4%, the average annual returns from the various investments should be around 5% for money markets, 7% for 2-year bonds, 10% for large-cap American stocks, 11% for small-cap American stocks, and 12% for foreign stocks with significant exposure to emerging markets. A quick spreadsheet calculation (e.g., `1.05^(1/252) = 1.00020` for money markets) gives the daily averages to use in the Asset constructors.

Bill also has found that daily volatility for money markets is quite low, but it rises very quickly as the daily average rises. He gives you the values in Listing 10.10 to use for the Asset constructors.

Listing 10.10 Replacement for the Portfolio constructor, Version 3

```
public Portfolio()
{  itsAsset[0] = new Asset ("money mkt   ", 1.00020, 1.0001);
   itsAsset[1] = new Asset ("2-yr bonds  ", 1.00027, 1.004);
   itsAsset[2] = new Asset ("large caps  ", 1.00038, 1.011);
   itsAsset[3] = new Asset ("small caps  ", 1.00041, 1.015);
   itsAsset[4] = new Asset ("foreign     ", 1.00045, 1.020);
}  //====================
```

Second, Bill suggests using an actual normal distribution to approximate daily market behavior. You already have a random-number-producing object named `random` from the Random class. The **`nextGaussian`** method of the Random class produces a double value that is distributed normally with a mean of 0.0 and a standard deviation of 1.0. So the body of Asset's `waitOneDay` becomes as follows:

```
return Math.exp (itsReturn + random.nextGaussian() * itsRisk);
```

The messiest part of what Bill has for you is an estimate of the influence of the trend of past market changes on the next day's market change. Many stock market experts say that such trends cannot be used profitably by individual investors, but other stock market experts disagree. In any case, the clients want to have a day's market change depend about one-third on previous trends and about two-thirds on chance. That is left as a major programming project.

Exercise 10.30 Explain why the algorithm for rounding to the nearest thousandth does not work if you subtract 1000 before applying `(int)` instead of after.
Exercise 10.31 What changes would you make in Listing 10.9 to have it round to hundredths of a percent instead of tenths of a percent?
Exercise 10.32 What changes would you make in Listing 10.9 if a good year is considered 1.64 standard deviations above the mean (1-chance-in-20 of that) and a bad year is 1.64 standard deviations below the mean (also 1-chance-in-20)?
Exercise 10.33 Write an independent method that finds the common logarithm of a given number, i.e., using base ten. Throw an IllegalArgumentException when the parameter is not positive.
Exercise 10.34* Calculate the probability that an asset goes up by at least 1 standard deviation in a 6-day period, similar to the calculation given for a 5-day period.
Exercise 10.35* Revise Listing 10.10 to get the ten numbers from a BufferedReader.
Exercise 10.36** When you look at the prospectus for a mutual fund, you see a chart showing how an initial investment of $10,000 would have grown over the years if you let it ride. Add an instance variable to each Asset object named `itsIndex` to track this amount. This `itsIndex` value is initially $10,000. Each time you call Asset's `waitOneDay`, it changes `itsIndex` by the appropriate amount. Add an instance method `getIndex()` to obtain the current value of `itsIndex` for that mutual fund. Have Portfolio's `displayCurrentValues` method print the current value of the index.

*10.9 Additional Java Statements (*Enrichment)*

Java has several statements that are not used elsewhere in this book. For three of them, the `switch` statement, the alternate constructor invocation, and the simple `return` statement, that is because the proper conditions for them to be used do not come up very often. The other statements mentioned here, in the opinion of this author, should never be used in well designed coding except for the use of the `break` statement in a `switch` statement.

The switch statement

The `switch` statement is used only when you need a multiway-selection structure that chooses one of many possibilities (usually four or more) depending on the value of a particular integer expression. For example, let us say the "weekday number" of Sunday is 1, of Monday is 2, of Tuesday is 3, etc. The method in Listing 10.11 (see next page) then returns the weekday number of a day of the week, given the String of characters that form its name. It does this by computing the sum of the integer equivalents of the first and fourth characters and going directly to the corresponding `case` clause. The seven cases are listing here in increasing order of the **control value** in the parentheses after `switch`, though they need not be. The last `return` statement is executed if none of the seven cases apply (perhaps the value in `day` is misspelled).

The requirements for a **switch statement** are strict, which make it less useful than it otherwise might be:

- The control value must be an int value or its equivalent (char, short, or byte). So the method in Listing 10.11 could not have `switch (day) { case "Sunday":...`
- The expression for each case must be computable at compile time. So it cannot involve a non-final variable.
- All case expressions must differ in their integer values.

The advantage of the `switch` statement is speed: It takes the flow-of-control directly to the right statement. A multiway if-statement would take less room in a program, but some `day` values could require testing five or six conditions.

Listing 10.11 Example of a switch statement

```
/** Return the number of the weekday with the given name. */

public static int weekdayNumber (String day)  // independent
{  switch (day.charAt (0) + day.charAt (3)) // 1st & 4th chars
   {  case 'F' + 'd':   // Friday
         return 6;
      case 'M' + 'd':   // Monday
         return 2;
      case 'S' + 'd':   // Sunday
         return 1;
      case 'W' + 'n':   // Wednesday
         return 4;
      case 'T' + 'r':   // Thursday
         return 5;
      case 'T' + 's':   // Tuesday
         return 3;
      case 'S' + 'u':   // Saturday
         return 7;
   }
   return 0;
}  //=====================
```

You can have the last alternative of a `switch` statement say `default:` rather than specifying a case. All values of the control value for which no case is listed end up in that default case. Without the default case, the switch statement does nothing for such values.

The fall-through effect for switch statements

The statements in each case of the `switch` statement normally finish with a `return` statement or some other way to leave the case. Otherwise the flow-of-control will continue into the next case ("fall-through"). You usually do not want this. For instance, you might write the following logic to assign the month number for a month beginning with 'J', where any month name with the wrong fourth letter switches into the default case:

```
switch (month.charAt (3)) // 4th char if month begins with 'J'
{  case 'e':
      num = 6;
   case 'u':
      num = 1;
   case 'y':
      num = 7;
   default:
      num = 0;
}
```

But that would be wrong. For instance, "January" switches to the second case which sets `num` to 1; then it falls through to the next case, setting `num` to 7; then it falls through to the default case, setting `num` to 0. You can prevent this fall-through if you put a `break` statement at the end of each case except the default. The `break` statement terminates the `switch` statement immediately, going on to execute the next statement in the method. The corrected version is shown in Listing 10.12.

Listing 10.12 A switch statement with breaks

```
switch (month.charAt (3))
{  case 'e':
      num = 6;
      break;
   case 'u':
      num = 1;
      break;
   case 'y':
      num = 7;
      break;
   default:
      num = 0;
}
```

The advantage of this fall-through property is that you can have several case phrases for a single group of statements. For instance, the following returns the first letter of the day with the given day number:

```
switch (dayNumber)
{  case 1:  case 7:
      return 'S';
   case 3:  case 5:
      return 'T';
   case 4:
      return 'W';
   case 6:
      return 'F';
   case 2:
      return 'M';
}
```

Caution If the statements within one of the `case` clauses include a looping statement, a `break` statement inside that looping statement will not break out of the `switch` statement. That `break` statement will only break out of the looping statement.

The simple return statement

The word `return` with a semicolon immediately after it is a legal Java statement. It can only be used in a constructor or in a method that returns nothing, i.e., has the word void in the heading. The effect is to terminate the method immediately, without going to the end of its coding. For instance, you might have some coding with the structure shown below on the left. Many people feel this is clearer if rewritten as shown on the right:

```
// without return
if (s != null)
{  s = s.trim();
   if (s.length() != 0)
   {  // several statements
   }
}
```

```
// same thing with return
if (s == null)
   return;
s = s.trim();
if (s.length() == 0)
   return;
// several statements
```

You have seen several cases of a loop with a continuation condition formed with `&&`. Often these can be written without `&&` if a `return` statement is used in the body of the loop. Many people prefer to do this. For instance, the Collision JApplet in Listing 8.12 has the following logic at the end of one action method:

```
int c;
while (c < SIZE && itsItem[c].misses (guy))
   c++;
if (c < SIZE && itsItem[c].isHealthy())
   itsItem[c].getsPoison();
```

This can be written instead as follows. Note that the loop control variable does not have to be declared outside the for-statement, because it is not used after the loop:

```
for (int c = 0;  c < SIZE;  c++)
   if ( ! itsItem[c].misses (guy))
   {  if (itsItem[c].isHealthy())
         itsItem[c].getsPoison();
      return;
   }
```

A new use of this

The first statement in a constructor can be `this(someParameters)` instead of `super(someParameters)`, where the parentheses are filled with whatever parameters are appropriate for the constructor method that the `this` statement calls. This "alternate constructor invocation" calls another constructor in the same class instead of a constructor from the superclass.

For instance, if you have two constructors with the headings

```
public Whatever (String name, int age, boolean inUnion)
public Whatever (String name)
```

then the first statement in the second constructor can be the following:

```
this (name, 21, true);
```

This statement replaces the usual `super` statement. It means that the first constructor is executed before anything else is done in the second constructor. In effect, it says that the second constructor uses a default age of 21 and a default unionization status `true`.

Multiple declarations and assignments

You may declare several variables in the same statement if they have the same type. For instance, the following statement declares three int variables, initializing the second one to 3. This should only be done for variables that are strongly related to each other, such as the <x, y> position of a graphical object:

```
int x, y = 3, z;
```

You may also assign several variables the same value at once. The following statement assigns 5 to each of the three named variables:

```
x = y = z = 5;
```

A bare semicolon is considered a statement. For instance, the following coding finds the first positive array component (empty braces `{ }` would be clearer):

```
for (k = 0;  k < size && a[k] > 0;  k++)
   ;
```

The break and continue statements

If you execute the statement `break;` inside the body of a loop, the loop terminates immediately. If you execute the statement `continue;` inside the body of a loop, the current iteration terminates and execution continues with the next iteration. You should know these facts so you can understand Java written by people who think it is alright to do these things. You can also put a label such as `ralph:` on a statement and then use the command `break ralph;` or `continue ralph;` to terminate looping constructs all the way out to the statement labeled as `ralph`.

Exercise 10.37 Write a `switch` statement to print the Roman numeral corresponding to 10, 50, 100, or 500; print "No" in all other cases.
Exercise 10.38 Rewrite the `switch` statement in Listing 10.12 as a multiway-selection structure.
Exercise 10.39* Write a `switch` statement to print the month number for any of twelve different strings of characters recording months of the year. Assume each string is at least four characters long (add a blank for May). Use a control value that is the sum of two character values, as in Listing 10.11.
Exercise 10.40* Explain why the compiler will not allow you to use `day.charAt(0) + day.charAt(1)` in Listing 10.11 as the control value for the seven alternatives.

*10.10 About Throwable And Error (*Enrichment)*

This chapter tells you all you are likely to need to know about the Exception class and its subclasses. The Exception class is a subclass of the Throwable class, which has one other subclass named Error.

The Throwable class (from java.lang)

The **Throwable** class has the following two constructors and four methods:

- `new Throwable(messageString)` creates a Throwable object with the specified error message.
- `new Throwable()` creates a Throwable object with a null error message.
- `someThrowable.getMessage()` returns the error message belonging to the executor.
- `someThrowable.toString()` returns `ClassName: ErrorMessage` where ClassName is the full class name for the subclass of the object being thrown (e.g., `java.lang.NullPointerException`) and ErrorMessage is the executor's error message. But if the Exception object was created with no error message, then all that it returns is the full class name alone.
- `someThrowable.printStackTrace()` prints the `toString()` value followed by one line for each method that is currently active, as described next. It prints to `System.err`, which is normally the terminal window. This method is overloaded with one version for printing to a PrintStream parameter and another version for printing to a PrintWriter parameter.
- `someThrowable.fillInStackTrace()` puts the current values in the StackTrace record so that `printStackTrace` gives the correct value.

A sample output from `printStackTrace` is as follows:

```
java.lang.NullPointerException: null String!
   at SomeClass.meth (SomeClass.java:20)
   at OtherClass.main (OtherClass.java:33)
```

This StackTrace indicates that the statement

```
throw new NullPointerException ("null String!");
```

was executed at line 20 in the SomeClass class, within the method named `meth`. This method was called from line 33 of the OtherClass, from within the `main` method. And that `main` method was called from the runtime system (since otherwise more method calls would be active). How do you know it was caused by an explicit throw statement rather than a simple attempt to evaluate an expression that put a dot on a null value? Because the runtime system throws a NullPointerException without an error message in the latter case.

The purpose of the `fillInStackTrace()` method is to correct the StackTrace when a throw statement re-throws an Exception it caught; that Exception has the StackTrace from its original point of creation, and `fillInStackTrace()` replaces it by the StackTrace for the statement that re-throws it. You can re-throw with a statement such as

```
throw e.fillInStackTrace();
```

because the method returns the Throwable object that is its executor.

The Error class (from java.lang)

The Error subclass of Throwable is generally for abnormal problems that are too serious to be caught -- you should just let the program crash. It has four primary subclasses, all of which are in the `java.lang` package:

- **AWTError** is thrown when a serious Abstract Windowing Toolkit problem happens (something in the graphics part of the program).
- **LinkageError** is thrown when a class depends on another class which has been altered since the dependent class was compiled. For instance, a **ClassCircularityError** indicates that initialization of class X requires prior initialization of another class Y which requires prior initialization of class X. This cannot be resolved without correcting and recompiling the classes. A **NoClassDefFoundError** occurs when the current class X requires another class that existed at the time X was compiled but does not exist at runtime. An **ExceptionInInitializerError** occurs for a problem initializing a class variable, e.g., `private static x = s.length()` when `s` is `null`.
- **VirtualMachineError** is thrown when the Java Virtual Machine breaks or is out of RAM or otherwise cannot operate correctly. For instance, a **StackOverflowError** happens when a program has a recursive call that goes thousands of levels deep (this generally indicates either an inappropriate application of recursion or an infinite loop). An **OutOfMemoryError** occurs when there is no more RAM available for calling a constructor and the garbage collector cannot find any to recover.
- **ThreadDeath** is thrown by a statement of the form `someThread.stop()`. This statement is **deprecated**, i.e., it is from an earlier version of Java, it is outdated, and it should never be used.

10.11 Review Of Chapter Ten

About the Java language:

- The **Exception** class has several subclasses including the **RuntimeException** class. If a method contains only statements that can throw a RuntimeException, the Exception does not have to be caught. But the program will crash if that Exception arises and is not caught.
- A non-RuntimeException is called a **checked Exception**. Any method containing a statement that can throw a checked Exception must handle the potential Exception in one of two ways: Put the statement within a try/catch statement that can catch it, or put a **throws clause**, `throws SomeException`, at the end of the method heading. The SomeException must be the same class or a superclass.
- A **try/catch statement** is of the form `try {...} catch (SomeException e) {...}` It has just the one **try-block** but possibly many **catch-blocks**, each with its own class of Exceptions. If an Exception is thrown by a statement in the try-block, the first catch-block that can handle that kind of Exception is executed.
- The Exception class has two constructors that every subclass of Exception should copy. One has no parameters and the other has a String parameter for supplying an error message.
- Every subclass of Exception inherits a `getMessage` method for retrieving the message part of an Exception object.
- You can throw an Exception using the statement `throw new Exception (someMessage)`. The message is optional. This throws a checked Exception. To throw an unchecked Exception, use `throw new RuntimeException (someMessage)` instead. Or use any other subclass of Exception that is appropriate.
- `someRandom.nextGaussian()` returns a value that normally distributed with a mean of 0.0 and a standard deviation of 1.0. The Random class is in the `java.util` package of the Sun standard library.
- Read through the documentation for Exception and its subclasses (partially described in this chapter). Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

About some Sun standard library RuntimeException subclasses:

- `ArithmeticException` is thrown when e.g. you divide by zero.
- `IndexOutOfBoundsException` has two commonly occurring subclasses: When you refer to `b[k]` and `k < 0` or `k >= b.length`, you get an ArrayIndexOutOfBoundsException object, and when you refer to `someString.charAt(k)` and `k < 0` or `k >= someString.length()`, you get a StringIndexOutOfBoundsException object.
- `NegativeArraySizeException` is thrown when e.g. you have `new int[n]` and `n` is a negative number.
- `NullPointerException` is thrown when you put a dot on an object variable that has the `null` value or you refer to `b[k]` and `b` has the `null` value. However, the former does not apply if the dot is followed by a class variable or a class method call.
- `ClassCastException` is thrown when e.g. you use `(Worker) x` for a Person variable `x` but `x` refers to a Client or Student object. `(ClassName) x` is allowed only when `x` refers to an object of that class, or a subclass of that class, or an implementation of that interface. However, it is also allowed that `x` be `null`.
- `NumberFormatException` is thrown by e.g. `parseDouble` or `parseInt` for an ill-formed numeral.
- All of the above Exception subclasses are in the `java.lang` package, which is automatically available to every class.

Other vocabulary to remember:

- Two (or more) arrays are called **parallel arrays** when the kth component of one tells you something about the kth component of the other. Parallel arrays do not often occur in Java because an object that combines the two (or more) parts is generally preferable.
- When N amounts are to be added to a base value, the average amount is found by adding them up and dividing by N. But when N amounts are to be multiplied by a base value, the average amount is the **geometric average**, found by multiplying them together and taking the N^{th} root. Alternatively, you take the usual average of their logarithms and then take the anti-log of the result.

Answers to Selected Exercises

10.1
```
if (k < a.length)
    System.out.println (a[k]);
if (k < str.length())
    System.out.println (str.charAt (k));
```
10.2
```
if (num >= 0)
    array = new int [num];
```
10.3
```
Put "try{" in front of the declaration of the rate variable.
Put the following after the right brace at the end of the body of the while-statement:
} catch (NumberFormatException e)
{   JOptionPane.showMessageDialog (null, "That was an ill-formed numeral.");
}
```
10.4
```
Replace the first statement in the actionPerformed method by the following:
int d;
try
{   d = Integer.parseInt (itsDaysRented.getText());
}
catch (NumberFormatException e)
{   d = 1;
}
```
10.5
```
The first prompt is "Entry?"    The second prompt is "Ill-formed integer: Entry?"
The third prompt is "Ill-formed integer: Ill-formed integer: Entry?"
```
10.8
```
public boolean checkError()
{   return isKeyboard;
}
```
10.9
```
public static void printStuff (Object[] toPrint)
{   try
    {   for (int k = 0;  k < toPrint.length;  k++)
            printOneItem (toPrint[k]);
    }
    catch (java.awt.print.PrinterAbortException e)
    {   System.out.println (e.getMessage());
    }
}
```
10.10
```
public static void printStuff (Object[] toPrint)
{   for (int k = 0;  k < toPrint.length;  k++)
        try
        {   printOneItem (toPrint[k]);
        }
        catch (java.awt.print.PrinterAbortException e)
        {   System.out.println (e.getMessage());
        }
}
```
10.14
```
public class BadStringException extends Exception
{
    public BadStringException()
    {   super();
    }
    public BadStringException (String message)
    {   super (message);
    }
}
```

10.15 Insert the following statement at the beginning of each of the two methods:

```
if (itself.length() == 0)
    throw new BadStringException();
```

Also revise the two method headings as follows, since BadStringException is checked:

```
public void trimFront (int start) throws BadStringException
public String firstWord() throws BadStringException
```

10.16 No modification is needed in the heading, since it is not a checked Exception. Insert the following statement at the beginning of the method body:

```
if (interestRate <= 0)
    throw new ArithmeticException ("The rate is negative");
}
```

10.18 4%. The end result is 1.50 * 1.50 * 0.50 = 1.125, for a 12.5% overall return. This is an average (geometric) return of 4.00%, since 1.0400 to the third power is 1.125.

10.19 1.10 to the 20^{th} power is 6.73, so $700 grows to about 6.73 times $700, i.e., $4710.

10.20 1.07 to the 20^{th} power is 3.87, so $700 grows to about 3.87 times $700, i.e., $2709. The ratio is 57.5%. So failure to have it in a Roth IRA costs you 42.5%.

10.21 10%, because Jo keeps only 7% after taxes, so Jo's investment grows at a rate of 7% in after-tax value.

10.24 Replace the assignment to value in Listing 10.6 by the following:

```
value = IO.askDouble ("How much do you add each month ");
while (value <= 0)
    value = IO.askDouble ("The initial investment must be POSITIVE; try again:");
```

10.26 Replace the heading of the for-loop by the following five lines:

```
if (itsValue[0] < 0)
    itsValue[0] *= itsAsset[k].waitOneDay() + 0.05 / Asset.YEAR;
else
    itsValue[0] *= itsAsset[k].waitOneDay();
for (int k = 1;  k < NUM_ASSETS;  k++)
```

10.30 If the number was originally 1.0317, You multiply by 1000 and add 0.5, giving 1032.7. You can then subtract 1000 and truncate in either order and it makes no difference. But if the number was originally 0.9924, you multiply by 1000 and add 0.5 to get 992.9. If you subtract 1000 before truncating, you get -7.1 which truncates to -7.0, which is the wrong answer. If you truncate before subtracting, you get -8.0, which is the right answer. That is, the problem arises with truncating negative numbers.

10.31 Replace 1000.0 by 10000.0 in two places, and replace 10.0 by 100.0.

10.32 Replace the second statement of the describe method by the following:

```
double risk = 1.64 * itsRisk * Math.sqrt (YEAR);
```

10.33

```
public static double commonLog (double x)
{   if (x <= 0)
        throw new IllegalArgumentException();
    return log (x) / log (10);
}
```

10.37

```
switch (num)
{   case 10:
        JOptionPane.showMessageDialog (null,  "X");
        break;
    case 50:
        JOptionPane.showMessageDialog (null,  "L");
        break;
    case 100:
        JOptionPane.showMessageDialog (null,  "C");
        break;
    case 500:
        JOptionPane.showMessageDialog (null,  "D");
        break;
    default:
        JOptionPane.showMessageDialog (null,  "No");
}
```

10.38

```
if (month.charAt (3) == 'e')
    num = 6;
else if (month.charAt (3) == 'u')
    num = 1;
else if (month.charAt (3) == 'y')
    num = 7;
else
    num = 0; // this is 6 lines shorter, though it may execute somewhat more slowly.
```

11 Abstract Classes And Interfaces

Overview

This chapter presents some additional standard library classes from the `java.lang` package and an extended example illustrating polymorphism. It is preferable to study Sections 10.1-10.3 (Exceptions and the elements of input files) before reading this chapter. Arrays are used heavily starting in Section 11.8.

- Section 11.1 introduces a new software design problem, the Numeric class.
- Sections 11.2-11.4 define and illustrate all of the new language features for this chapter: abstract classes, interfaces, and the instanceof operator. They also describe the standard wrapper classes Integer, Double, Long, etc.
- Sections 11.5-11.8 develop three different subclasses of the Numeric superclass and a class of objects representing arrays of Numeric values.
- Sections 11.9-11.11 are enrichment topics: a concept from theoretical computability, an elementary introduction to the meaning of threads, and some details on the Math and Character classes from the Sun standard library.

11.1 Analysis And Design Of The Mathematicians Software

You are hired by a think-tank of mathematicians to write software that they will use in their work. When you discuss with them the kinds of software they will need, you find that they work with various kinds of numbers that the Java language does not have. One kind of number is fractional. A fraction is one whole number divided by another. A fraction such as 1/3 cannot be represented exactly using a single double value. For some software situations, these mathematicians want the answer calculated as a fraction reduced to lowest terms, not as a decimal number.

Another category of numbers that Java does not provide is complex numbers, and a third category is very long integers of perhaps 40 to 50 digits. Many of the things you do with fractions you will also want to do with complex numbers and with very long numbers. So you will want basically the same methods for each of the Complex and VeryLong classes that you have for the Fraction class (though the details of how the methods calculate will differ).

You decide to make all three classes subclasses of a Numeric superclass. You can begin by developing what is common to all three of the Fraction, Complex, and VeryLong subclasses, then add to the individual subclasses whatever extra operations they need.

In developing object classes, you consider what the objects know (instance variables) and what they can do (instance methods). You start with what they can do for you. After you have that figured out, you decide what the objects have to know in order to do it. That is, analysis and design first develops the object operations involved and uses those to decide later on the object attributes.

Analysis

The first step is to develop documentation describing the operations in the Numeric class that you need. Part of getting the specifications clear is deciding how these methods are to be called by other methods. That is best expressed as a method heading.

Your talks with the mathematicians reveal four kinds of operations that they want their Numeric objects to have:

1. They want to be able to add, subtract, and multiply two Numerics to get a new Numeric result.
2. They want to be able to test two Numerics to see which is larger or if they are equal.
3. They want to be able to convert from a standard String form (e.g., "2/3" for a fraction) to the object and back again, and also convert any Numeric to a decimal number equivalent or approximation.
4. They want several other operations, such as finding the largest or smallest of two Numerics, finding the square, adding numbers obtained from an input source to a running total, etc. You decide this list will do for now; you can always add more later.

This discussion leads to the sketch in Listing 11.1. The bodies of the methods are the minimum needed for compiling, since they are irrelevant at this point. The sketch is design documentation, not implementation. The `toString` and `equals` methods override those standard methods from the Object class (the standard methods require that the parameter be of type Object). The `valueOf` method for Fractions returns the Fraction object corresponding to the String value "3/7"; the `valueOf` method for Complexes returns the Complex object corresponding to the String value "7.0 + 4.3i".

Listing 11.1 Documentation for the Numeric class

```
public class Numeric extends Object    // stubbed documentation
{
   /** Convert to the appropriate subclass. Return null if par is
    *  null or "".  Throw NumberFormatException if wrong form. */
   public Numeric valueOf (String par)          { return null; }

   /** Convert to string form. */
   public String  toString()                    { return null; }

   /** Convert to a real-number equivalent. */
   public double  doubleValue()                    { return 0; }

   /** Tell whether the executor equals the parameter. */
   public boolean equals (Object ob)            { return true; }

   /** Like String's compareTo: a positive means ob is "smaller".
    *  Throw NullPointerException if ob is null.
    *  Throw ClassCastException if ob is not the same type. */
   public int compareTo (Object ob)                { return 0; }

   /** Return a new Numeric that is the sum of both.
    *  Return null if they are the same subtype but the answer
    *  is not computable.  Throw ClassCastException if the
    *  executor and par are different subtypes. */
   public Numeric add (Numeric par)             { return null; }

   /** Same as add, except return executor minus par. */
   public Numeric subtract (Numeric par)        { return null; }

   /** Same as add, except return executor times par. */
   public Numeric multiply (Numeric par)        { return null; }

   // miscellaneous operations (described later)

   public Numeric square()                      { return null; }
   public Numeric addMore (BufferedReader source){ return null; }
}
```

Test data

You begin by writing a small test program that uses one of these special kinds of numbers. This helps you fix in your mind what you are trying to accomplish and how the methods will be used, and it gives you a way of testing the correctness of the coding you develop. A simple example is to have the user enter a number of Fraction values on the command line and respond with the total of the values entered and the smallest of the values entered. Your plan is that, if the program is executed using e.g.

```
java AddFractions 2/3 -9/10 3/4
```

then the output from the program will be a dialog box saying

```
total = 31/60; smallest = -9/10
```

The design of this program requires that you first check that at least one value was entered on the command line. If so, you assign the first value entered to a Numeric variable `total` and also to a Numeric variable `smallestSoFar`. To apply the `valueOf` method to the first String `args[0]`, you have to supply an executor (an object reference before the dot in ".valueOf") so the compiler knows to use Fraction's `valueOf` method instead of some other `valueOf` method. Each subclass of Numeric will define a ZERO value for this purpose. You may then have a loop that goes through the rest of the entries and (a) adds each one to `total`; (b) replaces `smallestSoFar` by the most recently seen value if that one is smaller.

Listing 11.2 contains a reasonable coding. Since the `valueOf` method can throw an Exception if a String value is not in the right format, you need to have a try/catch statement to catch any Exception thrown and give an appropriate message.

Listing 11.2 Application program to test Fraction operations

```
import javax.swing.JOptionPane;

/** Add up all Fraction values given on the command line.
 *  Print the total and the smallest one entered. */

public class AddFractions
{
   public static void main (String[] args)
   {  String response = "No values entered.";    // just in case
      if (args != null && args.length > 0)
         try
         {  Numeric total = Fraction.ZERO.valueOf (args[0]);
            Numeric smallestSoFar = total;
            for (int k = 1;  k < args.length;  k++)
            {  Numeric data = total.valueOf (args[k]);
               total = total.add (data);
               if (smallestSoFar.compareTo (data) > 0)
                  smallestSoFar = data;
            }
            response = "total = " + total.toString()
                  + "; smallest = " + smallestSoFar.toString();
         }catch (Exception e)
         {  response = "Some entry was not fractional in form.";
         }
      JOptionPane.showMessageDialog (null, response);
      System.exit (0);
   }  //=====================
}
```

Exercise 11.1* Write the heading of a `divide` method to divide a Numeric value by a whole-number value and the heading of a `before` method that tells whether a Numeric value is less than another one.

11.2 Abstract Classes And Interfaces

At first you think you might make Numeric an interface, since it makes no sense to give most of these methods bodies in the Numeric class. But then you realize that some of the methods can have definitions that make sense for all three subclasses. An interface has only method headings, with no method bodies allowed. So you need something in between an interface (where all methods must be overridden) and a regular superclass (all methods defined, so overriding is always optional). The recommended Java solution in this case is an **abstract class**.

You can put `abstract` in an instance method heading and replace the method's body by a semicolon. This makes the class it is in an abstract class. You also must put `abstract` in the class heading to signal this. Any class that extends the abstract class must implement each of its abstract methods if you want to be able to construct objects of that class. You cannot have an abstract class method, because class methods cannot be called polymorphically. You may have a constructor if you wish, but you cannot call it except by using `super` in a subclass. A reasonable abstract Numeric class is shown in Listing 11.3 (see next page), leaving eight methods to override in each subclass. The comments describing these methods are in the earlier Listing 11.1.

You might at some point want to read some Fractions in from the keyboard or from a disk file and add them to a Fraction object that already has a value. The command `y = x.addMore(someBufferedReaderObject)` would do this, using the `addMore` method from the Numeric class and a Fraction variable `x`. Since the executor is a Fraction object and `addMore` is an instance method, the runtime system uses the `valueOf` and `add` methods from the Fraction class. Reminder on disk files:

- `new BufferedReader (new FileReader (someString))` opens the file with the given name and returns a reference to it.
- `someBufferedReader.readLine()` produces the next line in the file (with the `\n` removed), except it produces the null String when it reaches the end of the file.
- Both of these uses of BufferedReader can throw an IOException, which your coding must either catch (with a try/catch statement) or throw further. IOException and BufferedReader are both in the `java.io` package.

Interfaces

The phrase `implements Comparable` in the heading for the Numeric class means that any Numeric object, or any object from a subclass of Numeric, can be used in any situation that requires a Comparable object. This is because the Numeric class contains the standard `compareTo` method. The `compareTo`, `add`, `subtract`, and `multiply` methods throw a ClassCastException or NullPointerException if the parameter is not a non-null object of the same subclass as the executor.

You may declare a variable or parameter to be of Comparable type, but you cannot use the phrase `new Comparable()` to create Comparable objects. Comparable is the name of an **interface**, not a class. When you put the word `interface` in a heading instead of `class`, it means that all you can have inside the definition of the interface are (a) instance method headings with a semicolon in place of the method body, and (b) final class variables.

Listing 11.3 The abstract Numeric class

```
import java.io.BufferedReader;

public abstract class Numeric extends Object
                              implements Comparable
{
   public abstract Numeric valueOf (String par);
   public abstract double doubleValue();
   public abstract String toString();
   public abstract boolean equals (Object ob);
   public abstract int compareTo (Object ob);
   public abstract Numeric add (Numeric par);
   public abstract Numeric subtract (Numeric par);
   public abstract Numeric multiply (Numeric par);


   /** Return the larger of the two Numeric values.
    *  Precondition:  they are both of the same Numeric type. */

   public static Numeric max (Numeric data, Numeric other)
   {  return   data.compareTo (other) > 0   ?   data   :   other;
   }  //=====================

   /** Return the square of the executor. */

   public Numeric square()
   {  return this.multiply (this);
   }  //=====================

   /** Read Numeric values from the BufferedReader and add them
    *  to the executor, until null is seen.  Return the total. */

   public Numeric addMore (BufferedReader source)
   {  Numeric total = this;
      try
      {  Numeric data = this.valueOf (source.readLine());
         while (data != null && total != null)
         {  total = total.add (data);
            data = this.valueOf (source.readLine());
         }
      }catch (Exception e)
      {   // no need for any statements here; just return total
      }
      return total;
   }  //=====================
}
```

You must compile a file containing an interface definition before you can use it. The Comparable interface in the standard library has a single method heading (and is already compiled for you). The complete compilable file `Comparable.java` is as follows:

```
package java.lang;
public interface Comparable
{  public int compareTo (Object ob);
}
```

The two primary reasons why you might have a class implement an interface instead of extend another class are as follows:

1. You will not or cannot give the logic (the body) of any of the methods in a superclass; you want each subclass to define all methods in the superclass. So you use an interface instead of a superclass.
2. You want your class to inherit from more than one class. This is not allowed in Java, because it can create confusion. A class may extend only one other class. But it may implement many interfaces. A general class heading is as follows:

```
class X extends P implements Q, R, S.
```

The reason an interface cannot contain a method heading for a class method is that, if `someMethod` is a class method, the method definition to be used for `sam.someMethod()` is determined at compile-time from the class of the variable `sam`, not at run-time from the class of the object.

The phrase X **implements** Y is used only when Y is an interface; X should have every method heading that Y has, but with a method body for each one. That is, Y declares the methods and X defines them (unless X is an "abstract" class).

Early and late binding

When you call Numeric's `addMore` method in Listing 11.3 with a Fraction executor, every statement in that `addMore` method refers to a Fraction object. For instance, since `this` is a Fraction object, the first statement makes `total` refer to a Fraction object. So the runtime system uses the `valueOf` method from the Fraction class, which returns a reference to a Fraction object to be stored in `data`. If, however, you called the `addMore` method with a Complex executor, then every statement in that method would refer to a Complex object.

Each of the method calls in the method definitions of Listing 11.3 is polymorphic, except of course `source.readLine()`. For instance, the `square` method calls the `multiply` method of the Fraction class if the executor is a Fraction object, but it calls the `multiply` method of the Complex class if the executor is a Complex object.

The runtime system decides which method is called for during execution of the program, depending on the class of the executor. This is called **late binding** (since it binds a method call to a method definition). When a class has no subclasses, the compiler can do the binding; that is **early binding**. Late binding is somewhat slower and more complex, but it gives you greater flexibility. And tests have shown that it is no slower than an efficiently implemented logic using if-statements or the equivalent.

Language elements

The heading of a compilable unit can be: public abstract class ClassName
or: public interface InterfaceName

You may replace "public" by "public abstract" in a method heading if (a) it is in an abstract class, (b) it is a non-final instance method, and (c) you replace the body of the method by a semicolon.
If your class extends an abstract class and you want to construct objects of your class, your class must have non-abstract methods that override each abstract method in the abstract class.
All methods in an interface must be non-final instance methods with a semicolon in place of the body. They are by default "public abstract" methods. Field variables must be final class variables.
You may add the phrase "implements X" to your class heading if X is an interface and if each method heading in the interface appears in your class. Use the form "implements X, Y, Z" to have your class implement several interfaces.

Exercise 11.2 Write a method named `min3` that could be added to the Numeric class: It finds the smallest of three Numeric parameters.
Exercise 11.3 Write a method `public Numeric smallest (BufferedReader source)` for Numeric: it finds the smallest of a list of Numeric values read in from `source`, analogous to `addMore`.
Exercise 11.4** Write a method `public boolean equalPair (BufferedReader source)` for Numeric: It reads in a list of Numeric values from a given BufferedReader source and then tells whether or not any two consecutive numbers were equal. Explain how the runtime system knows which method definition to use.

11.3 More Examples Of Abstract Classes And Polymorphism

A new language feature illustrated in this section is that you can put the word **final** in a method heading, which means no subclass can override that method. This allows the compiler to apply early binding to calls of that method, which makes the program run a bit faster. For instance, you would probably want to make the Fraction methods mentioned in the previous section final to speed execution, since you do not see why anyone would want to extend the Fraction class.

Also, you may put the word final in a class heading, which means that the class cannot have subclasses. You may have wondered why Chapter Six declared a StringInfo class with a String object as its only instance variable instead of making StringInfo a subclass of String. It is because the developers of the String class made it a final class.

Abstract games

The BasicGame class in Listing 4.3 describes how to play a trivial game in which the user guesses the secret word, which is always "duck". Its true purpose is to serve as a parent class for interesting games such as Mastermind and Checkers. It would make more sense to make BasicGame an abstract class, thereby forcing programmers to override some of its methods. The seven methods in the BasicGame class are as follows:

```
public void playManyGames()   // calls playOneGame
public void playOneGame()     // calls the 5 below
public void askUsersFirstChoice()
public boolean shouldContinue()
public void showUpdatedStatus()
public void askUsersNextChoice()
public void showFinalStatus() // just says the player won
```

Of these seven methods, a subclass should not override the first two, since their logic is what is common to all games. A subclass may choose to override `showFinalStatus` from the BasicGame class (as did Nim in Listing 4.8) or may leave it as inherited (as did GuessNumber in Listing 4.6 and Mastermind in Listing 4.9). Every subclass of BasicGame should override the other four methods.

You can enforce these three rules as follows: Declare the first two methods as final (a method declared as final cannot be overridden in a subclass). Leave the `showFinalStatus` method as a normal "concrete" method, so overriding is optional. Declare the other four as abstract, which means that they must be overridden. That gives the compilable class shown in Listing 11.4 (see next page). If this definition replaces the BasicGame class in Chapter Four, then the three different game subclasses in Listings 4.6, 4.8, and 4.9 will all compile and run correctly as is.

Listing 11.4 The BasicGame class as an abstract class

```
import javax.swing.JOptionPane;

public abstract class BasicGame extends Object
{
   public final void playManyGames()
   {  playOneGame();
      while (JOptionPane.showConfirmDialog (null, "again?")
                     == JOptionPane.YES_OPTION)
         playOneGame();
   }  //=====================

   public final void playOneGame()
   {  askUsersFirstChoice();
      while (shouldContinue())
      {  showUpdatedStatus();
         askUsersNextChoice();
      }
      showFinalStatus();
   }  //=====================

   public abstract void askUsersFirstChoice();
   public abstract boolean shouldContinue();
   public abstract void showUpdatedStatus();
   public abstract void askUsersNextChoice();

   public void showFinalStatus()
   {  JOptionPane.showMessageDialog (null,
                     "That was right.  \nCongratulations.");
   }  //=====================
}
```

Abstract Buttons

Sun's Swing classes include several different kinds of graphical components called buttons: JButton (the only kind that can be a default button), JToggleButton (click it to change its state from "selected" to "deselected" or back again), and JMenuItem (one of several buttons on a menu). All have different appearances, but they have some behaviors in common, such as `someButton.setText(someString)` to change what the button says on it and `someButton.setMnemonic(someChar)` to say what character can be used as the "hot key".

The AbstractButton class in the Swing library contains the methods that are common to all of these kinds of buttons, including `setText` and `setMnemonic`. The advantage is that each of these common methods only has to be defined once, in the AbstractButton class, but a common method can be called for any of the three kinds of buttons.

Abstract Animals

A particular application may require you to keep track of fish and other inhabitants of the ocean. You might have a Shark class, a Tuna class, a Porpoise class, and many others. Any behaviors that are common to several classes of animals should be specified in a superclass and inherited by its subclasses. For instance, all animals eat. So you could have an Animal superclass that declares a method for eating. Animal should be an abstract class, since any concrete animal you create will be of a specific animal subclass. You could define the Animal class as follows:

```
public abstract class Animal
{  public abstract void eat (Object ob);
}
```

There are two kinds of animals, those that move around in the ocean and those that do not. It makes sense to abstract the behavior of swimming for a subclass of Animals. So you could have an abstract subclass of the Animal class as follows:

```
public abstract class Swimmer extends Animal
{  public abstract void swim();
}
```

Note that the Swimmer class does not have to override the `eat` method, since Swimmer is abstract. But any concrete class (i.e., a class that you want to create an instance of) that extends Swimmer must implement both `eat` and `swim`. For instance, a prototype for the Shark class could be as follows:

```
public class Shark extends Swimmer
{  public void swim()
   {  System.out.println ("shark is swimming");
   }
   public void eat (Object ob)
   {  System.out.println ("shark eats " + ob.toString());
   }
   public String toString()
   {  return "shark");
   }
}
```

The instanceof operator

A class that overrides the `equals` method from the Object class must have its parameter be of Object type. If the parameter is of the same class as the executor, you test something such as the names or other instance variables to see if they are equal. If the parameter is not of the same class as the executor, then of course it is not equal to the executor. The problem is, how do you ask an object whether it is of the right class without throwing a ClassCastException?

You will need the **instanceof** operator for this purpose: If x is an object variable and Y is a class, then `x instanceof Y` is true when x is not `null` and x is an object of class Y or of a subclass of Y; if Y is an interface, x must be of a class that implements Y.

The Time class in Listing 4.5 should have an `equals` method that overrides the Object method. Since its instance variables are two int values `itsHour` and `itsMin`, the `equals` method could be defined as follows:

```
public boolean equals (Object ob)    // in the Time class
{  if ( ! (ob instanceof Time))
         return false;
   Time given = (Time) ob;
   return this.itsHour == given.itsHour
               && this.itsMin == given.itsMin;
}  //======================
```

The initial tests guards against having the third line of this method throw a ClassCastException method; calling the `equals` method should never throw an Exception. Similarly, the Worker class in Listing 7.6 has an `equals` method whose parameter is a Worker. It would be good to have another `equals` method that can be

used for any Object parameter. The following logic calls on the existing Worker `equals` method to do most of the work (overloading the `equals` identifier):

```
public boolean equals (Object ob)   // in the Worker class
{  return ob instanceof Worker  ?  this.equals ((Worker) ob)
                                :  false;
}  //======================
```

If a class of objects implements the Comparable interface, its objects may be passed to a Comparable parameter of some method. That method might use the `equals` method as well as the `compareTo` method, which is one good reason for having an `equals` method that overrides the Object method whenever you have a `compareTo` method. Note that both Time and Worker implement Comparable.

Another good reason to have an `equals` method with an Object parameter is that you might have an array of Animals or Numerics, where the objects in the array could be of several different subclasses. Then you could call the `equals` method with any one of those objects as the executor, knowing the runtime system will select the right definition of this polymorphic `equals` call according to the actual class of the executor.

Caution If you get the compiler message "class X must be declared as abstract", it may not be true. You might simply have forgotten to implement one of the abstract methods in a superclass of X, or you might have forgotten to implement one of the methods required by an interface that X implements.

Language elements
If you put "final" in a method heading, you cannot override it in a subclass.
If you put "final" in a class heading, you cannot make a subclass of it.
If Y is a class, x instanceof Y means x is an object in class Y or in a subclass of Y (so x != null).
The equals method in the Object class has an Object parameter. Any method you write that overrides that equals method should never to throw an Exception, so the first statement in such a method is usually if (! (ob instanceof Whatever)) return false;.

Exercise 11.5 Write an `equals` method for the Person class in Listing 5.2. Have it override the Object method. Two Persons are equal if they have the same values of the instance variables `itsLastName` (a String) and `itsBirthYear` (an int).
Exercise 11.6 Write a prototype for the Tuna subclass of the Swimmer class. However, Tunas only eat things that swim, so make sure the `eat` method states that it is still hungry if the object it is given to eat is not a Swimmer.
Exercise 11.7* Modify the answer to the preceding exercise so that, if a Tuna is given a Shark to eat, then the Shark eats the Tuna instead.
Exercise 11.8* Override Object's `equals` method for the Worker class without calling on the other equals method. Just test directly whether two Worker objects have the same last name and the same first name (`itsFirstName` and `itsLastName` are String instance variables).
Exercise 11.9** Improve the Animal class and its subclasses to add an instance method `getWeight()` for all Animals and a corresponding instance variable set by a parameter of an Animal constructor. Also add an Animal instance variable `itsFuelReserves` that is to be updated by the weight of whatever the Animal eats. Then modify the Shark's `eat` method to call on the appropriate methods.

11.4 Double, Integer, And Other Wrapper Classes

Quite a few of the programs the mathematicians need involve whole numbers that are extremely large, up to 40 to 50 digits (somewhat over one septillion of septillions). The problem is that the Java language does not provide for whole numbers that big. The only primitive numeric types in Java are byte, short, int, long, double, and float.

The six primitive number types

A **byte of storage** is 8 on-off switches, or 8 high/low voltages, or 8 binary digits (1's and 0's). So a byte can store any of 256 different values (since the eighth power of 2 is 256).

You may define a variable as **byte x**, which means that x stores a number in the range from -128 to 127. Note that there are exactly 256 different values in that range, so it is stored in one byte of space in RAM.

You may define a variable as **short x**, which means that x stores a number in the range from -32,768 to 32,767. Note that there are exactly 65,536 different values in that range, which is the second power of 256. So it is stored in two bytes of space in RAM. This is 16 bits, since one byte is 8 bits.

You may define a variable as **int x**, which means that x stores a number in the range of plus or minus just over 2 billion. The reason is the fourth power of 256 is just over 4 billion. So an int value takes up four bytes of space in RAM, which is 32 bits.

You may define a variable as **long x**, which means that x stores a number in the range of plus or minus just over 8 billion billion. The reason is that the eighth power of 256 is just over 16 billion billion (a number with 19 digits). So a long value takes up eight bytes of space in RAM. This is 64 bits, since one byte is 8 bits. You can indicate that an integer value in your program is of the long type with a trailing capital L, as in 47L or 1000000L.

Figure 11.1 Difference in sizes of whole-number values

You may define a variable as **float x**, a value with 32 bits, which takes up four bytes of storage. It is a decimal number with about 7 decimal digits of accuracy, since it is stored in scientific notation with a base of 16.

You may define a variable as **double x**, a value with 64 bits, which takes up eight bytes of storage. It has only about 15 decimal digits of accuracy; the rest of the storage is used for the minus sign (if there is one) and to note the power of sixteen it is multiplied by (scientific notation, but with a base of 16 rather than 10).

You may write a double value in **computerized scientific notation** within a Java class, e.g., 3.724E20 is 3.724 times 10-to-the-twentieth-power, and 5.6E-15 is 5.6 times 10-to-the-negative-fifteenth-power. The `toString()` method of the Double class produces this scientific notation unless the number is at least 0.001 and less than 10 million.

Wrapper classes for primitives

The Sun standard library offers eight special **wrapper classes**, one for each of the eight primitive types. Their names are **Double**, **Integer**, **Float**, **Long**, **Short**, **Byte**, **Character**, and **Boolean**. These classes give you an easy way to convert one of the primitive types of values to an object and back again. They are all in the `java.lang` package, which means that they are automatically imported into every class; you do not have to have an explicit import directive.

One use of these classes is for polymorphic processing of several different kinds of objects. Since each of these eight wrapper classes inherits from the Object class, you can apply the `equals` method and the `toString` method to any of their objects. Each of these wrapper classes overrides these two methods to provide the obvious meaning to them. For instance, you might have an array of Objects which can be any of the eight kinds of values. You might then search for an Object equal to a target Object with this logic:

```
for (int k = 0;  item[k] != null;  k++)
   if (item[k].equals (target))
      System.out.println (item[k].toString());
```

Numeric wrappers

The Double class has the class method `parseDouble` for converting a String value to double value. It can throw a NumberFormatException if the numeral in the string of characters is badly formed. Similarly, the Integer class has the `parseInt` class method and the Long class has the `parseLong` class method. All of these can throw a NumberFormatException. The `parseDouble` method allows spaces before or after the numeral, but `parseInt` does not.

Each of the six numeric wrapper classes has a constructor with a parameter of the corresponding primitive type and a constructor that has a String parameter. So you can create a new object of one of these wrapper types using e.g. `new Double(4.7)` or `new Integer("45")`. The latter is equivalent to `new Integer (Integer.parseInt("45"))`.

All six of the numeric wrapper classes are subclasses of the abstract **Number** class in the `java.lang` package. This chapter would have used Number rather than Numeric except that the Number class does not have many of the methods that the clients wanted; Number only has six parameterless instance methods for converting to each of the six primitive types: `doubleValue`, `intValue`, `longValue`, `byteValue`, `shortValue`, and `floatValue`. Each of the six numeric wrapper classes overrides those six methods to convert the object to any of the six numeric primitive types.

All wrapper classes except Boolean implement the Comparable interface. That is, they have the standard `compareTo` method required to be Comparable. And they each have final class variables MAX_VALUE and MIN_VALUE with the appropriate meaning.

Integer methods for different number bases

The **binary form** of a number is a sequence of 1's and 0's. Its value is the sum of several numbers: 1 if the last digit is 1, 2 if the next-to-last digit is 1, 4 if the third-to-last digit is 1, 8 if the fourth-to-last digit is 1, etc. That is, a 1 at any place has twice the value of a 1 at the place to its right. For instance, "1111" represents `1*8 + 1*4 + 1*2 + 1` which is 15.

The **octal form** of a number is a sequence of digits in the range 0 to 7 inclusive. Its value is the sum of several numbers: n if the last digit is n, 8*n if the next-to-last digit is n, 64*n if the third-to-last digit is n, 512*n if the fourth-to-last digit is n, etc. That is, a digit at any place indicates 8 times the value of a digit at the place to its right. For instance, "2537" represents `2*512 + 5*64 + 3*8 + 7` which is 1375.

The **hexadecimal form** of a number is a sequence of digits in the range 0 to F inclusive (we run out of ordinary digits after 9, so we use A=10, B=11, C=12, D=13, E=14, and F=15). Its value is the sum of several numbers: n if the last digit is n, 16*n if the next-to-last digit is n, 256*n if the third-to-last digit is n, 4096*n if the fourth-to-last digit is n, etc. That is, a digit at any place indicates 16 times the value of a digit at the place to its right. For instance, "B4E" can be thought of as (11)(4)(14), which represents `11*16*16 + 4*16 + 14`, which is `2816 + 64 + 14`, which is 2894.

`Integer.toString (n, 2)` gives the binary form of the int value `n`. If you replace the 2 by 8, 16, or 10, you get the octal, hexadecimal, or decimal form of the int value `n`, respectively. Conversely, `Integer.parseInt (someString, 2)` returns the int value corresponding to the string of binary digits (and similarly for 8, 16, and 10).

Useful Boolean methods

The phrase `new Boolean(someString)` gives the Boolean object `Boolean.TRUE` if `someString` is the word `TRUE` (in any combination of upper- and lowercase), otherwise it gives `Boolean.FALSE`. And `new Boolean(someBoolean)` gives the wrapper object for the primitive value.

If `x` is a Boolean object, then `x.booleanValue()` gives the boolean primitive value `true` or `false` as appropriate. Boolean does not implement the Comparable interface, but it does have instance methods `equals` and `toString` that override the corresponding methods in the Object class.

Useful Character methods

The only constructor for the Character class has a char parameter, as in `new Character('B')`. If `x` and `y` are Character objects, then `x.compareTo(y)`, `x.equals(y)`, and `x.toString()` all have the standard meanings. `x.charValue()` is the char equivalent of `x`.

Language elements
You may declare variables of type byte, short, or float. The primary reason to do so is to save space. Most people do not use these types except for an array of thousands of values. This book does not use these three types outside of this section.

Exercise 11.10 Convert each of these numbers to binary, octal, and hexadecimal: 5, 11, 260.
Exercise 11.11 Convert each of these hexadecimal numbers to decimal: F, 5D, ACE.

11.5 Implementing The Fraction Class

A Fraction object is a virtual fraction representing e.g. 2/3. Since you already know the eight operations you want (described in the upper part of Listing 11.3), you can move on to deciding about the instance variables. You will also find it useful to have a final class variable representing ZERO.

The concrete form of a Fraction

After mulling the situation over for a while, you decide that the best concrete form of a Fraction object is two int instance variables representing the numerator and the denominator of the fraction, reduced to lowest terms. The logic for addition, multiplication, and other operations can be worked out from there. Figure 11.2 shows a representation of a Fraction object. It is a good idea to write down the internal invariant separately (the condition that each Fraction method will make sure is true when the method exits):

Internal invariant for Fractions A Fraction has two int instance variables `itsUpper` and `itsLower`. `itsLower` is positive and the two values have no integer divisor in common. This pair of int values represents the quotient `itsUpper/itsLower`.

Figure 11.2 A Fraction variable and its object

The Fraction constructor

You will need a constructor that creates a Fraction object from two given int values that will be the numerator and the denominator. You could just set the two instance variables of the Fraction to those two values, except for three problems your logic must manage:

1. If the denominator is zero, there is no such number. A reasonable response is to just create the fraction 0/1.
2. If the denominator is negative, then multiply both the upper and lower parts of the fraction by -1 before proceeding, since the denominator is supposed to be positive.
3. You have to reduce the fraction to lowest terms.

You may have several other methods that require reducing a fraction to lowest terms. So that reducing logic should be in a private method. The command `this.reduceToLowestTerms()` in the constructor has the object being constructed execute the `reduceToLowestTerms` method. That is, `this` refers to the object being constructed within constructor methods and to the executor within ordinary methods. The coding for this constructor is in the upper part of Listing 11.5 (see next page).

The valueOf method

For the `valueOf` method, you return `null` if the String value is `null` or the empty string. Otherwise you apply the `Integer.parseInt` method to the parts before and after the slash to get `itsUpper` and `itsLower` parts. This may throw a NumberFormatException. You may then call the Fraction constructor to check for zeros, negatives, or values that should be reduced. The coding for the `valueOf` method is in the lower part of Listing 11.5.

Listing 11.5 The Fraction class, partial listing

```
public class Fraction extends Numeric
{
   /** A constant representing ZERO. */

   public static final Fraction ZERO = new Fraction (0, 1);
   ////////////////////////////////
   private int itsUpper;  // the numerator of the fraction
   private int itsLower;  // the denominator of the fraction


   /** Construct a Fraction from the given two integers. */

   public Fraction (int numerator, int denominator)
   {  super();
      if (numerator == 0 || denominator == 0)
      {  itsUpper = 0;
         itsLower = 1;
      }
      else if (denominator < 0)
      {  itsUpper = - numerator;
         itsLower = - denominator;
         this.reduceToLowestTerms();
      }
      else
      {  itsUpper = numerator;
         itsLower = denominator;
         this.reduceToLowestTerms();
      }
   }  //======================

   /** The parameter should be two ints separated by '/', e.g.
    *  "2/3".  Return null if par is null or "".  Otherwise
    *  throw a NumberFormatException if the wrong form. */

   public Numeric valueOf (String par)
   {  if (par == null || par.length() == 0)
         return null;
      int k = 0;
      while (k < par.length() - 1 && par.charAt (k) != '/')
         k++;
      return new Fraction
                  (Integer.parseInt (par.substring (0, k)),
                   Integer.parseInt (par.substring (k + 1)));
   }  //======================
}
```

The toString and equals methods

The `toString` method simply returns the two int values with a slash between them, so that just takes one statement to implement. The `equals` method is more difficult. It has an Object as the given parameter, not a Fraction. This is required to have it override the `equals` method in the Object class. But then you cannot refer to `par.itsUpper` in the body of `equals`. Such an expression requires that `par` be a Fraction variable.

No outside class should call Fraction's `equals` method unless the parameter does in fact refer to a Fraction object. You can test for this using the `instanceof` operator defined earlier: `ob instanceof Fraction` is `true` if `ob` refers to a Fraction object at runtime, otherwise it is `false`. Once you know that `ob` in fact refers to a Fraction object, you may refer to the `itsUpper` instance variable of that Fraction object with the phrase `((Fraction) ob).itsUpper`. This coding is in the upper part of Listing 11.6.

Listing 11.6 Three more methods in the Fraction class

```
/** Express the Fraction as a String, e.g., "2/3". */

public String toString()
{  return this.itsUpper + "/" + this.itsLower;
}  //=====================

/** Tell whether the two Fraction objects are equal. */

public boolean equals (Object ob)
{  return ob instanceof Fraction
         && ((Fraction) ob).itsUpper == this.itsUpper
         && ((Fraction) ob).itsLower == this.itsLower;
}  //=====================

/** Return the sum of the executor and par. */

public Numeric add (Numeric par)
{  Fraction that = (Fraction) par;    // for simplicity
   return new Fraction (this.itsUpper * that.itsLower
                      + this.itsLower * that.itsUpper,
                        this.itsLower * that.itsLower);
}  //=====================
```

A method that overrides the basic Object class's `equals` method is never to throw a NullPointerException or ClassCastException, so you need to use the `instanceof` operator to guard against these Exceptions.

The `(Fraction)` part of that phrase is a cast, just as `(int)` is. It says that `ob` can be treated as a reference to a Fraction object. However, if you use a phrase such as `(Fraction) ob` more than once or twice in some coding, it is clearer and more efficient if you assign the value to a Fraction variable and use that Fraction variable instead.

The add method

The addition of one Fraction to another gives a new Fraction as a result. You should remember that you add two fractions by "cross-multiplying": The new `itsUpper` value is the first's `itsUpper` times the second's `itsLower`, added to the first's `itsLower` times the second's `itsUpper`. And the new `itsLower` value is the first's `itsLower` times the second's `itsLower`. Once you make this calculation, you can create a new Fraction object out of the two results and return it. This coding is in the lower part of Listing 11.6.

The reduceToLowestTerms method

The `reduceToLowestTerms` method is supposed to reduce its executor object to lowest terms. So it should divide both `itsUpper` and `itsLower` by the same whole number wherever possible. You could proceed in this manner:

1. If both `itsUpper` and `itsLower` are divisible by 2, divide out the 2 and repeat.
2. If both `itsUpper` and `itsLower` are divisible by 3, divide out the 3 and repeat.
3. If both `itsUpper` and `itsLower` are divisible by 5, divide out the 5 and repeat.
4. If both `itsUpper` and `itsLower` are divisible by 7, divide out the 7 and repeat, etc.

After the first step, you have divided out 2 until one of the two numbers is odd. So it is sufficient to try only odd divisors thereafter. A little more thought shows that you do not have to try any divisor that is more than the smaller of `itsUpper` and `itsLower`. To calculate the smaller of `itsUpper` and `itsLower`, you have to use the absolute value of `itsUpper`, because it could be a negative number.

Dividing out a whole number is done in two places in the main logic of `reduceToLowestTerms`, so a separate private method is desirable. This method simply divides both parts of a Fraction by the parameter until it will not go evenly into one of them. Listing 11.7 has this `reduceToLowestTerms` method, as well as the obvious logic for the `doubleValue` method. The other three methods of the Fraction class are left as exercises. Figure 11.3 gives the UML class diagram for the whole Fraction class.

Listing 11.7 Additional methods in the Fraction class

```
/** Return the approximate value as a double value. */

public double doubleValue()
{  return itsUpper * 1.0 / itsLower;
}  //=====================

/** Reduce the fraction to lowest terms.  Precondition:  the
 *  denominator of the fraction is positive. */

private Fraction reduceToLowestTerms()
{  divideOut (2);
   int limit = Math.min (Math.abs (itsUpper), itsLower);
   for (int divider = 3;  divider <= limit;  divider += 2)
      divideOut (divider);
   return this;
}  //=====================

private void divideOut (int divider)
{  while (itsUpper % divider == 0 && itsLower % divider == 0)
   {  itsUpper /= divider;
      itsLower /= divider;
   }
}  //=====================

// these three are stubbed and left as exercises
public int     compareTo (Object ob)            { return 0; }
public Numeric subtract (Numeric par)        { return null; }
public Numeric multiply (Numeric par)        { return null; }
```

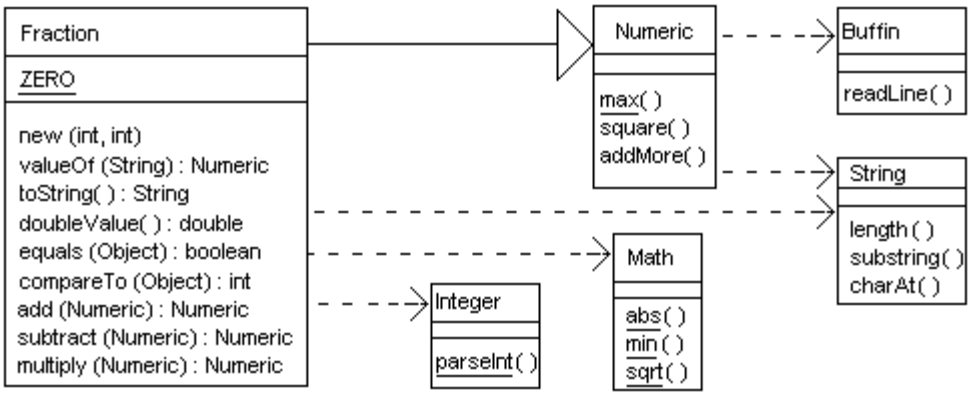


Figure 11.3 UML class diagram for the Fraction and Numeric classes

Exercise 11.12 The String class has a method `indexOf(someChar)` that returns the index where the first instance of `someChar` is found in the String. It returns `-1` if `someChar` is not there. Use it to rewrite the `valueOf` method of the Fraction class.
Exercise 11.13 Explain why you cannot replace `{itsUpper = 0; itsLower = 1;}` by `return ZERO;` in the logic for the Fraction constructor.
Exercise 11.14 In Listing 11.6, the `equals` method tests `ob instanceof Fraction` but the `add` method does not. Explain why it is not necessary.
Exercise 11.15 Write the Fraction method `public Numeric subtract (Numeric par)`, giving the difference of two Fractions (analogous to the `add` method).
Exercise 11.16 Write the Fraction method `public int compareTo (Object ob)`.
Exercise 11.17* Write the Fraction method `public Numeric multiply (Numeric par)`, giving the product of two Fractions (analogous to the `add` method).
Exercise 11.18* How would you revise the `add` method to return `Fraction.ZERO` when a ClassCastException arises?

11.6 Implementing The Complex Class

Another category of software that the mathematicians need involves the use of complex numbers. These are numbers that have a real part and an imaginary part, such as `3 - 4i`. The `i` stands for the square root of negative 1.

The Complex class extends the Numeric class and defines objects that represent complex numbers. Since you already know what operations you want, you can move on to deciding about the instance variables. Clearly, each Complex object should have a real part and an imaginary part. The following is a rather obvious internal invariant.

Internal invariant for Complexes A Complex object has two double instance variables `itsReal` and `itsImag`. It represents the complex number `itsReal + itsImag * i`.

You need a constructor to create a Complex object from two given numbers for the real and imaginary parts in that order. And you need methods that let you add, subtract, multiply and compare Complex numbers, and to convert to and from Complex numbers. The Complex class in Listing 11.8 is a start. It has the constructor and four of the Numeric methods; the other four are left for exercises.

Listing 11.8 The Complex class, partial listing

```
public class Complex extends Numeric
{
   public static final Complex ZERO = new Complex (0, 0);
   ////////////////////////////////
   private final double itsReal;
   private final double itsImag;


   public Complex (double realPart, double imagPart)
   {  super();
      itsReal = realPart;
      itsImag = imagPart;
   }  //======================

   public String toString ()
   {  String operator = itsImag < 0  ?  " "  :  " +";
      return itsReal + operator + itsImag + "i";
   }  //======================

   public double doubleValue()
   {  return Math.sqrt (itsReal * itsReal + itsImag * itsImag);
   }  //======================

   public int compareTo (Object ob)
   {  double diff = this.doubleValue()
                  - ((Complex) ob).doubleValue();
      if (diff == 0)
         return 0;
      else
         return   diff > 0  ?  1  :  -1;
   }  //======================

   public Numeric add (Numeric par)
   {  Complex that = (Complex) par;
      return new Complex (this.itsReal + that.itsReal,
                          this.itsImag + that.itsImag);
   }  //======================

   // the following are left as exercises
   public Numeric valueOf (String par)          {return ZERO;}
   public boolean equals (Object ob)            {return true;}
   public Numeric subtract (Numeric par)        {return ZERO;}
   public Numeric multiply (Numeric par)        {return ZERO;}
}
```

The `doubleValue` method calculates the **magnitude** of the complex number. If the complex number `x + y*i` were a point <x, y> in the plane, then the magnitude would be the distance from the origin point. That calculates as the square root of x-squared plus y-squared.

The `compareTo` method compares two Complex numbers on the basis of magnitude. The one with the greater magnitude is considered larger. The `(Complex)` cast is required, because otherwise the phrase `ob.doubleValue()` is not acceptable to the compiler -- Object objects in general do not have a `doubleValue` method.

The `add` method saves the trouble of making two casts by storing the cast value in a local variable of the Complex type and using it in the calculations. Both `compareTo` and `add` will throw a ClassCastException or a NullPointerException if the parameter is not a non-null Complex object.

You might be wondering how the group of mathematicians handle cases where they just want to use ordinary decimal numbers mixed in with the Fractions and the Complex class. The answer is, you need to develop a subclass of Numeric for decimal numbers as well. Until you get around to it, you can just use Complex objects with the imaginary part equal to zero. Figure 11.4 shows a representation of a Complex object.

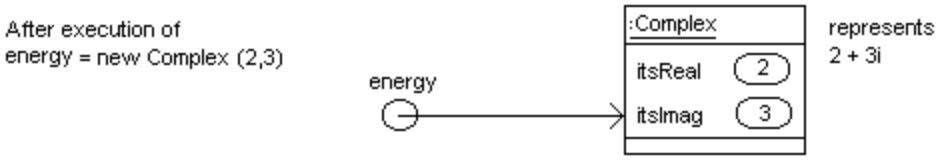


Figure 11.4 A Complex variable and its object

Exercise 11.19 Write the `equals` method of the Complex class, testing whether the two Complex values have the same real and imaginary parts.
Exercise 11.20 Write the `multiply` method of the Complex class.
Exercise 11.21 Revise the Complex `compareTo` method so that it accepts any Numeric object as the parameter and gives a reasonable answer.
Exercise 11.22* Write the `subtract` method of the Complex class.
Exercise 11.23* Write the `valueOf` method for the Complex class. Assume the input is of the same form that `toString` produces, but possibly with extra whitespace.

11.7 Implementing The VeryLong Class Using Arrays

You have to define a VeryLong object class for extremely large whole numbers, up to 40 or 50 digits. Since you already know what they can do (the methods in Numeric), you have to decide what they know (the private instance variables that store the value).

You could use three long values to store up to 54 digits, which is quite enough for your needs. However, that is not easily generalized for longer numbers, and problems arise when you try to perform multiplication. It is better to use an array of six int values, one for each group of nine digits of the number being stored. You can multiply two int values and store the exact result in a long variable; you have no easy way to store the exact result of multiplying two long values.

The internal invariant

You might decide to call the int array `itsItem`. `itsItem[0]` could contain the first (leftmost) nine digits of the VeryLong number, and the rest could go on from there. You would find it easiest to not allow negative VeryLong numbers. You need to check with your clients to make sure that this is acceptable to them.

Internal invariant for VeryLongs A VeryLong object is stored as six int values in `itsItem[0]...itsItem[5]`. Each must be non-negative and have no more than 9 digits. The whole-number value that the object represents is `itsItem[0] * 10`^{45} + `itsItem[1] * 10`^{36} + ... + `itsItem[4] * 10`^{9} +`itsItem[5]`.

This concrete description tells you all you need to know to be able to write methods that add, multiply, etc. with VeryLong numbers: It will be true when you start; make sure it is true when you finish. A partial listing of the VeryLong class of numbers is in Listing 11.9. Note: During the development, it turns out to be more efficient to have two different forms of one billion, an int value and a long value (you want to minimize unnecessary casts).

Listing 11.9 The VeryLong class, partial listing

```
public class VeryLong extends Numeric
{
   public static final VeryLong ZERO = new VeryLong (0, 0, 0);
   private static final int BILLION = 1000000000;
   private static final long LONGBILL = 1000000000L;
   private static final int MAX = 6;
   ////////////////////////////////
   private final int[] itsItem = new int[MAX];  // all zeros


   public VeryLong (long left, long mid, long right)
   {  super();
      this.itsItem[0] = (int) (left / LONGBILL);
      this.itsItem[1] = (int) (left % LONGBILL);
      this.itsItem[2] = (int) (mid / LONGBILL);
      this.itsItem[3] = (int) (mid % LONGBILL);
      this.itsItem[4] = (int) (right / LONGBILL);
      this.itsItem[5] = (int) (right % LONGBILL);
   }  //====================

   public String toString ()
   {  String s = "" + itsItem[0];
      for (int k = 1;  k < MAX;  k++)
         s += "," + (BILLION + itsItem[k] + "").substring(1);
      return s;
   }  //======================

   public Numeric add (Numeric par)
   {  VeryLong that = (VeryLong) par;
      VeryLong result = new VeryLong();
      for (int k = 0;  k < MAX;  k++)
         result.itsItem[k] = this.itsItem[k] + that.itsItem[k];
      for (int k = MAX - 1;  k > 0;  k--)
         if (result.itsItem[k] >= BILLION)
         {  result.itsItem[k] -= BILLION;
            result.itsItem[k - 1]++;
         }
      return result.itsItem[0] >= BILLION  ?  null  :  result;
   }  //====================

   private VeryLong()     // only used by other VeryLong methods
   {  super();
   }  //====================
}
```

The constructor and the toString method

How will people supply a numeric form of a very long number to be made into a VeryLong object? A reasonable way is to have them break the number up into three 18-digit parts, left to right, and supply those as three long values. So `new VeryLong (0, 217, 333333333222222222)` would give the number 217,333,333,333,222,222,222. The constructor only needs to split each of the three long values into two 9-digit parts and store them in the appropriate six components of the array.

What if one of the three parameters is negative or has nineteen digits? Just as with Fractions, you should create the equivalent of ZERO when one of the parameters is unacceptable this way. This adjustment is left as an exercise.

The `toString` method should put the commas in the written form of the number. Without them, the numeral would be too hard to read. Grouping digits by threes would give up to 18 groups, which is probably not helpful. So the client agrees that groups of nine digits is better. If the `itsItem` array contains e.g. {0, 0, 0, 37, 12345, 1234567}, you have to supply the missing zeros to make it 37,000012345,001234567.

A simple trick adds the right number of leading zeros: Add a billion to the up-to-9-digit number, convert it to a string of characters, then throw away the initial 1. You do not need to do this for the first part, `itsItem[5]`, but you do for the rest of the components. These methods are in the top part of Listing 11.9.

The add method

To add two VeryLong numbers, you first create a VeryLong object named `result`. You then add up corresponding components in the two things being added to get the same component of the `result`. But what if one of the sums goes over nine digits? You carry the 1. That is, you subtract a billion from that component of the `result` and add 1 to its next component. If the leftmost component goes over nine digits, the sum is too big to store, so you are to return `null` according to the specifications. The accompanying design block expresses the algorithm in Structured English.

DESIGN for the add method in the VeryLong class

1. Name the two values to be added `this` and `that`.
2. Create a VeryLong object to store the result of the addition. Call it `result`.
3. For each of the six possible indexes do the following...
   Add the current components of `this` and `that` to get the corresponding component of `result`.
4. For each components of `result` except one, starting from the rightmost digits of the number and working towards the left, do the following...
   If the current component has more than nine digits then...
   a. Subtract a billion to reduce it to nine or fewer digits.
   b. Add 1 to the component to its left.
5. If the leftmost component of result has more than nine digits then...
   Return `null`, since the answer cannot be expressed using six components.
   Otherwise...
   Return the `result`.

The rest of the VeryLong methods are left as exercises, except that the `multiply` method is sufficiently complicated that it is a major programming project. Note that no public method in the Numeric class or any of its subclasses is an action method. So an outside class cannot change the value of a Numeric object once it is created: Objects from Numeric and its subclasses are **immutable**. This is similar to the String class, in that no method in the String class allows you to change a String object once you create it.

Exercise 11.24 Write the `doubleValue` method for the VeryLong class.
Exercise 11.25 Write the `equals` method for the VeryLong class.
Exercise 11.26 Modify the VeryLong constructor to create the equivalent of ZERO when any one of the parameters is negative or has more than eighteen digits.
Exercise 11.27* Write a Real subclass of Numeric for ordinary numbers with one double instance variable. This lets the clients mix in ordinary numbers with the special ones.
Exercise 11.28* Write the `subtract` method for the VeryLong class.
Exercise 11.29* Write the `compareTo` method for the VeryLong class.
Exercise 11.30* The Complex instance variables are final but the Fraction instance variables are not, even though both are immutable classes. Explain why.
Exercise 11.31* The VeryLong `toString` method produces leading zeros when the leftmost one or more components of `itsItem` are zero. Revise it to fix this problem.
Exercise 11.32** Write the `valueOf` method for the VeryLong class. The input may or may not contain commas among the digits.
Exercise 11.33** Discuss the changes that would be needed to allow negative VeryLong numbers.

11.8 Implementing The NumericArray Class With Null-Terminated Arrays

These mathematicians often deal with numbers in big bunches. They may read in a bunch of numbers from a file once, then calculate the average of the whole bunch, find the smallest and the largest, add a value in order, etc. For this, you decide to store a lot of Numeric objects in an array of Numeric values. Call it the **NumericArray** class.

Once you define an array, you cannot change its size. So you need to make it big enough for the largest number of values you can expect. But then the array is generally only partially filled. So you have to have some way of noting the end of the array. You could keep track of the size, but you decide instead to put the `null` value in all the components after the end of the actual Numeric values in the array, as shown in Figure 11.5. For instance, if the array has size 1000 and currently contains only 73 values, then those 73 values will be stored in components 0 through 72 and the `null` value will be stored in each of components 73...1000.

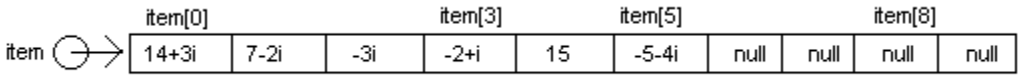


Figure 11.5 picture of a null-terminated array with six Complex values

What if the array is full? We do not allow that. A precondition for these **null-terminated arrays** is that they contain at least one instance of `null`.

Precondition for NumericArrays For the array parameter `item`, there is some integer `n` such that `0 <= n < item.length` and `item[k]` is not `null` when `k < n` and `item[k]` is `null` otherwise. The non-null values are the values on the conceptual list in order, with the first at index 0.

The find method

To illustrate how to work with such a null-terminated array, consider the problem of searching through the array to find whether a particular non-null value is there. You process each value of an int variable k from 0 on up, until `item[k]` is `null`. At each array value before that point, you compare it with the target value. If they are equal, you

return `true`. If you reach the point where `item[k] == null`, you know the target value is not in the array, so you return `false`.

What if the calling method passed in a target value of `null`? The condition `item[k].equals(target)` returns `false` when you test whether a non-null value equals a `null` value. So each time through the loop in the `find` method, the if-condition is false. Eventually `find` returns `false`. This logic is in the upper part of Listing 11.10.

Listing 11.10 The NumericArray class

```
import java.io.*;

public class NumericArray
{
   /** Tell whether target is a non-null value in the array. */

   public static boolean find (Object[ ] item, Object target)
   {  for (int k = 0;  item[k] != null;  k++)
         if (item[k].equals (target))
            return true;
      return false;
   }  //=====================

   /** Return the sum of the values in the array.
    *  Precondition: All non-null values are of the same
    *  Numeric type and their sum is computable. */

   public static Numeric sum (Numeric[ ] item)
   {  if (item[0] == null)
         return null;
      Numeric total = item[0];
      for (int k = 1;  item[k] != null && total != null;  k++)
         total = total.add (item[k]);
      return total;
   }  //=====================

   /** Print each non-null value on the screen. */

   public static void display (Object[ ] item)
   {  for (int k = 0;  item[k] != null;  k++)
         System.out.println (item[k].toString());
   }  //=====================

   /** Return the array of values read from the source, to a
    *  maximum of limit values.  Precondition: limit >= 0 and
    *  data is the same Numeric subtype as all input values. */

   public static Numeric[ ] getInput (Numeric data,
             int limit, BufferedReader source) throws IOException
   {  Numeric[ ] item = new Numeric [limit + 1];  // all nulls
      for (int k = 0;  k < limit;  k++)
      {  item[k] = data.valueOf (source.readLine());
         if (item[k] == null)
            return item;
      }
      return item;
   }  //=====================
}
```

This `find` method has been written with Object in place of Numeric. The reason is that only the `equals` method is used in the logic, and every Object has an `equals` method. The method goes in the NumericArray class because that is where you need it. But as a general principle, you should make your methods apply more generally when nothing is lost by it. You are allowed to assign a `Numeric[]` array to an `Object[]` array, though not vice versa. The runtime system chooses the right `equals` method for each Object (another use of polymorphism).

The sum and display methods

Listing 11.10 shows the NumericArray class with some other useful class methods. The `sum` method finds the sum of all the non-null values in the array. It throws a ClassCastException if they are not all of the same Numeric type. It uses a logic you have seen before: Initialize the `total` to the first value in the array. Then add the second value to it, then the third value to that, etc. The `total.add(item[k])` expression is polymorphic: The runtime system chooses the `add` method in the subclass of Numeric that `total` belongs to. That requires that each item be of the same Numeric subtype.

Further thought indicates you need to allow for the possibility that the result of adding two values may be `null`, which occurs when the sum is not computable. The accompanying design block records the logic in detail. Remember, a primary purpose of the design in Structured English is to verify that all possibilities have been handled properly before you attempt to translate the logic to Java.

DESIGN for the sum method in the NumericArray class

1. If the given list contains no values at all, i.e., the first component is `null`, then...
   Return `null`.
2. Create a Numeric object to store the result of the addition. Call it `total`.
   Initialize it to be the first value in the list (at index 0).
3. For each additional value in the list do the following...
   Add the next value in the list to `total` (but stop if the result becomes `null`).
4. Return the `total` as the answer.

The `display` method prints every value in the array. Since the only method called in the body of the `display` method is the `toString` method, which every Object has, the `display` method is written more generally to handle any array of Objects whatsoever, even from different classes. The runtime system chooses the right `toString` method for each Object (another use of polymorphism).

The getInput method

The `getInput` method returns a new array containing all the values read in. It requires a Numeric data value to do its job, even though it totally ignores the value supplied. It would normally be the ZERO of a subclass of Numeric. The only purpose of the data value is to act as the executor of the `valueOf` method, so that the runtime system knows which of the subclass methods to use at that point (another use of polymorphism).

For the `getInput` method, every non-null value that is read in must be stored in the array. What if the `limit` is ten and the source contains ten or more Numeric values? That tenth value has to be put into the array as well. Since the array has to have a `null` value after all the useable Numeric values, it has to have at least eleven components. That is why the array size is made larger than the given limit.

A program using a NumericArray

Every method in the NumericArray class is a class method. This is because the object you are working with, an array of Numeric values, is a parameter rather than an executor. So this is a utilities class analogous to the Math class.

Listing 11.11 illustrates how simple programs can be that involve arrays of Fractions, with just the methods in Listing 11.10. The program reads in up to 100 Fraction values from a file named "numeric.dat". Then it prints all of the values as fractions reduced to lowest terms. Finally, it prints out the sum of the values as a fraction reduced to lowest terms. Figure 11.6 is the UML diagram for this AddFractions class.

Listing 11.11 An application program using Fractions and NumericArrays

```
import java.io.*;

public class AddFromFile
{
   /** Read in up to 100 Fraction values from a file,
    *  then display them all on the screen with their sum. */

   public static void main (String[ ] args) throws IOException
   {  Numeric [ ] values = NumericArray.getInput
                  (Fraction.ZERO, 100, new BufferedReader
                        (new FileReader ("numeric.dat")));
      NumericArray.display (values);
      System.out.println ("Their sum is "
                        + NumericArray.sum (values));
   }  //=====================
}
```

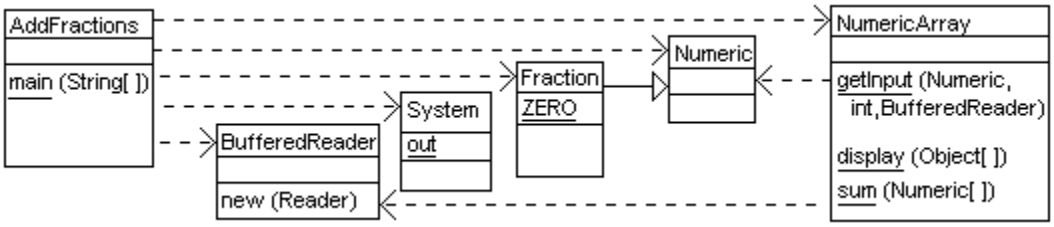


Figure 11.6 UML class diagram for the AddFractions class

An alternative for storing a large number of Numeric values is to use a NumericList object having two instance variables, a **partially-filled array** of Numerics `itsItem` and an int `itsSize` that tells how many components at the front of the array have useable Numeric values. Then the `display` method of Listing 11.10 would be expressed as follows:

```
private Numeric[] itsItem;
private int itsSize;
public void display()   // for NumericList
{  for (int k = 0;  k < itsSize;  k++)
      System.out.println (itsItem[k]);
}  //======================
```

Exercise 11.34 Write a NumericArray method to find the sum of the `doubleValue` values of the objects in the array, even when the objects are of various subclasses of Numeric.

Exercise 11.35 Write a NumericArray method `public static int size (Numeric[ ] item)` to find the number of non-null values in the array.

Exercise 11.36 Write a NumericArray method `public static boolean allSmall (Numeric[ ] item, Numeric par)` to tell whether every value in the array is smaller than `par`. Precondition: All values are comparable to each other.

Exercise 11.37 Write a NumericArray method `public static int indexSmallest (Numeric[ ] item)` to find the index where the smallest value is stored. Precondition: The array has at least one value, and all values are comparable to each other.

Exercise 11.38 Write a NumericArray method `public static void delete (Numeric[ ] item, int n)` to delete the value at index `n` and keep the rest of the values in the same order they were originally in. No effect if there is no value at index n.

Exercise 11.39 Write a NumericArray method `public static Numeric[] getComplexes (Numeric[ ] item)` to return a new null-terminated array containing just the values in the given array that are Complex objects.

Exercise 11.40* Rewrite the `getInput` method efficiently to have the condition of the for-statement be `k < limit && data != null`.

Exercise 11.41** Write a NumericArray method `public static double fractional (Numeric[ ] item)` to find the fraction of values that are Fraction objects (e.g., return 0.25 if a quarter are Fractions). Return zero if there are no values in the array.

Exercise 11.42* Write a NumericArray method with the heading `public static Numeric maximum (Numeric[ ] item)` to find the largest value in the array. Avoid all NullPointerExceptions. Return `null` if the array contains zero values. Precondition: Any two values in the array are comparable to each other.

Exercise 11.43* Write a NumericArray method with the heading `public static void insert (Numeric given, Numeric[ ] item)` to insert the given value in the array and keep it in ascending order. Precondition: The values in the array when the method is called are already in ascending order, the array has enough room, and any two values are comparable to each other. Restriction: Go to the end of the array and work backwards.

Exercise 11.44* Revise the entire Listing 11.10 so that it has one instance variable of type `Numeric[ ]`. Remove the `item` parameter of the first three methods, since it will be in the executor. Only the `getInput` method should be a class method; it should return a NumericArray object.

Exercise 11.45* Revise the `getInput` method in Listing 11.10 to catch any Exception and return the array as it stands at that time.

Exercise 11.46** Write a NumericArray method `public static Numeric[] reverse (Numeric[ ] item)` to return a null-terminated array containing the same values but in the opposite order.

Exercise 11.47** Write a NumericArray method `public static boolean isNulled (Object[] item)` that tells whether the parameter is in fact a null-terminated array. This method has no preconditions at all. Hint: The first statement should test `item[item.length-1] != null`.

*11.9 Too Many Problems, Not Enough Solutions (*Enrichment)*

Problem: Tell whether a given positive integer is odd.
Solution: The following boolean class method answers the question:

```
public static boolean isOdd (int num)
{  return num % 2 == 1;
}  //=====================
```

Problem: Tell whether a given positive integer is a prime.
Solution: The following boolean class method answers the question:

```
public static boolean isPrime (int num)
{  for (int k = 2;  k <= num / 2;  k++)
      if (num % k == 0)
         return false;
   return num > 1;
}  //=====================
```

In general, a **decision problem** is of the form "Tell whether a given positive integer has a certain property." A **solution** to a decision problem is a boolean class method with one integer parameter, that returns `true` for parameters that have the property and `false` for those that do not. Another example is:

Problem: Tell whether a given positive integer is the sum of two odd primes. For instance, `6 = 3+3, 8 = 5+3, 10 = 5+5, 12 = 7+5`, so those happen to all be the sum of two odd primes.
Solution: The following boolean method answers the question:

```
public static boolean isGoldbach (int num)
{  for (int k = 3;  k <= num / 2;  k += 2)
      if (isPrime (k) && isPrime (num - k))
         return true;
   return false;
}  //=====================
```

It would be nice if every decision problem had a solution. Of course, that cannot be true if there are more decision problems than there are solutions to decision problems. Let us see how many there are of each.

Every Java function is written as a sequence of characters. Each character can be stored in one byte of storage, which is 8 bits. The (extremely long) sequence of bits you get this way can be interpreted as a number base 2. Call that the **Java integer** of the function. So every solution to a decision problem has a Java integer, and different solutions have different Java integers. Therefore, we may conclude:

Deduction 1: The number of solutions to decision problems is no more than the number of positive integers.

For any given decision problem testing for a certain property, you can visualize a corresponding base 2 number as follows:

1. Write a decimal point (correction, make that a binary point).
2. Write 1 for the 1st digit after the binary point if 1 has the property and 0 if it does not.
3. Write 1 for the 2nd digit after the binary point if 2 has the property and 0 if it does not.
4. Write 1 for the 3rd digit after the binary point if 3 has the property and 0 if it does not.
5. Write 1 for the 4th digit after the binary point if 4 has the property and 0 if it does not, etc.

In general, the n^{th} digit of the number is 1 if the property is true for n and is 0 if the property is false for n. Call this the **property number** of the decision problem. For instance, the decision problem solved by `isOdd` has 0.101010101... as its property number (which is 2/3; check this by adding up 1/2 + 1/8 + 1/32 + 1/128 ad infinitum).

So every number between 0 and 1, written in base 2, is the property number of some decision problem, and different numbers between 0 and 1 have different decision problems (since if the numbers differ in even one place, say the 17^{th}, then for the two corresponding decision problems, 17 has one property but not the other). Therefore, we may conclude:

Deduction 2: The number of decision problems is no less than the number of real numbers between 0 and 1.

Put those two deductions together with the mathematical fact that the number of real numbers between 0 and 1 is far greater than the number of positive integers to get:

Deduction 3: Almost all decision problems have no solution in Java.

This is a result from the **Theory of Computability**, which you will learn more about in advanced courses in computer science. You will learn the reasoning behind the fact that, for any attempt to match up the positive integers one-to-one with the real numbers between 0 and 1, you will leave over 99% of the real numbers without a match. So over 99% of all decision problems have no Java method that solves them. In a sense, over 99% of all sets of yes-no questions about positive integers have no answers.

Exercise 11.48* (Essay Question) Which has more, the set of odd numbers or the set of primes? Why?

*11.10 Threads: Producers And Consumers (*Enrichment)*

Situation #1: A portion of a program displays a continually changing scene on the monitor, but as soon as the user clicks a button or moves the mouse or presses a key, the scene disappears and the program reacts to the user's action. An example of this is a screensaver.

Situation #2: A portion of a program is waiting for part of a web page to download. It cannot continue with what it is doing until the download completes. So it checks every tenth of a second or so and, when the download is still going on, lets another portion of the program do something useful for the next tenth of a second.

Situation #3: A process performs a long series of calculations to eventually produce a result which it deposits in a variable. A second process is waiting for that result. When it appears, the second process uses it as the starting point for its own long series of calculations. Meanwhile, the first process is working on producing a second result. Each time the second process (the **consumer**) needs a new result, it waits for it to appear and then uses it. Each time the first process (the **producer**) computes a new result, it checks that the previous result has been taken and, when it has, deposits the new result.

In all of these situations, it would be extremely useful to have two or more independent computer chips, each executing its own method and interacting with the other chips as needed. That is much simpler than trying to keep track of where you are in each of several processes and switching back and forth between them.

Concurrent execution

Java provides an equivalent of this called Threads. Each of several threads of execution run their own methods "simultaneously". What really happens (unless your program actually has more than one computer chip available for its use) is that the one chip switches back and forth between several different **threads of execution**, doing each one for such a short period of time that it may appear to the human observer that all are executing simultaneously. We say they run **concurrently** rather than simultaneously.

The graphical user interface is one thread of execution. If that thread is executing a screensaver kind of method, repeatedly making changes in the screen display, it is not available to listen for a button click or other action by the user. So the user would be clicking with no effect. The screensaver action might continue forever.

If, however, the main thread of execution performs statements that create a new Thread object `process` to execute the screensaver method, the main thread can then go back to listening for some event to happen. When it detects say a button click, it can then react by sending a message to the `process` object to stop what it is doing.

The Runnable interface

The **Runnable** interface specifies a method with the heading `public void run()`. When a class `implements Runnable`, its `run` method can control a single thread of execution. If you have a Thread variable named `process` that is to execute the `run` method in a ScreenSaver class, you can create a new thread of execution and have it start execution with this coding:

```
Runnable action = new ScreenSaver();
Thread process = new Thread (action);
process.start();
```

The Thread constructor creates a new thread of execution that will use the `run` method of the parameter. When you call the `start` method for a Thread object, the object initializes the concurrent thread of execution and then calls the `run` method specified. If you later wish to stop execution of that thread, execute the following statement:

```
process.interrupt();
```

This statement notifies the process thread that some other thread wants it to terminate, but it does not force the termination. That is up to the `process` thread. It can find out whether it has received an interrupt request by testing the following condition:

```
Thread.interrupted()
```

Listing 11.12 contains an example of the use of these language features. It omits the messy details of how the screen display changes during execution of the screensaver, since that is not relevant to the overall concurrency logic. The `actionPerformed` method would be in some class that can refer to the `startButton` and the `process`. The Thread class and Runnable interface are in the `java.lang` package, so they can be used in a class without having import directives.

Listing 11.12 The ScreenSaver class and the method that uses it

```
// reaction to a click of either startButton or stopButton

   Thread process;
   Button startButton, stopButton;


   public void actionPerformed (ActionEvent e)
   {  if (e.getSource() == startButton)
      {  process = new Thread (new ScreenSaver());
         process.start();
      }
      else
         process.interrupt();
   }  //=====================



public class ScreenSaver implements Runnable
{
   public void run()
   {  while ( ! Thread.interrupted())
         changeTheScenerySome();
   }  //=====================

   private void changeTheScenerySome()
   {  // make a small change in the display on the monitor
   }  //=====================
}
```

The producer-consumer situation

If you have two Runnable objects called perhaps `producer` and `consumer`, you may create a Thread object for each and start both their `run` methods executing concurrently as follows:

```
Thread pro = new Thread (producer);
Thread con = new Thread (consumer);
pro.start();
con.start();
```

The `start` method for Thread objects sets up the concurrent process and then executes the `run` method of the given Runnable object (`producer` or `consumer` in this case). Think of the objects being produced as pies. The producer is continually producing pies and depositing them in the place where the consumer can find them, pausing only if the consumer gets behind in eating them. And the consumer is continuing consuming pies, pausing only if the producer gets behind in baking them.

The Producer and Consumer classes could be designed as shown in Listing 11.13. The messy details of the actual eating and baking are left unstated. The coding uses the conventional `for(;;)` notation to create an infinite loop.

Listing 11.13 The Producer and Consumer classes

```
public class Producer implements Runnable
{
   public void run()
   {  for (;;)
      {  Object dessert = produce();
         while (Resource.hasUneatenPie())
         {  }                          // wait until unoccupied
         Resource.setPie (dessert);    // mark it occupied
      }
   }  //=====================

   public Object produce()
   {  // extensive action required to produce a pie
   }  //=====================
}
//#################################################################


public class Consumer implements Runnable
{
   public void run()
   {  for (;;)
      {  while ( ! Resource.hasUneatenPie())
         {  }                              // wait until occupied
         consume (Resource.getPie());      // mark it unoccupied
      }
   }  //=====================

   public void consume (Object dessert)
   {  // extensive action required to consume a pie
   }  //=====================
}
```

These two classes presume the existence of the Resource class which serves as a pie depository. The Resource class has three class methods:

- `setPie(Object)` deposits the given pie. Only one can be there at a time.
- `getPie()` returns the pie currently on deposit; it returns null if there is no pie.
- `hasUneatenPie()` tells whether a pie is currently on deposit.

The Resource class could be implemented with the following two private class variables. Then the `setPie` method could set `ready` to `true` and assign the parameter value to `pie`, and the `hasUneatenPie` method could simply return the value of `ready`:

```
private boolean ready = false;
private Object pie = null;
```

Sleeping threads

The producer and consumer use a busywait to wait for something to happen, i.e., they execute statements that do nothing useful while waiting. This ties up the processor unnecessarily. A thread of execution can execute the following statement to free up the processor for n milliseconds:

```
Thread.sleep (n);
```

This method call can throw an InterruptedException that must be acknowledged (i.e., it is not a RuntimeException). So the method call should normally be used only within a try/catch statement. You could replace the no-action semicolon in the body of the Consumer's `while` statement by the following statement, if you define the ThreadOp utility class shown in Listing 11.14:

```
if (ThreadOp.pause (50))  // true when interrupted
   return;
```

Listing 11.14 The ThreadOp class

```
public class ThreadOp
{
   public static boolean pause (int millis)
   {  try
      {  Thread.sleep (millis);
         return Thread.interrupted();
      }catch (InterruptedException e)
      {  return true;
      }
   }  //====================
}
```

That statement relinquishes the processor for 50 milliseconds (0.050 seconds), then checks to see whether an interrupt signal was sent. If so, the `return` statement stops the execution of its `run` method. Otherwise it checks to see if a new pie is available. If so, it gets one more pie, eats it, and then returns to its waiting state. Note that it does not allow itself to be interrupted in the middle of eating a pie; that would waste the effort spent in partially processing its data, to say nothing of wasting a chunk of a perfectly good pie. The interrupt request does not force termination, it only suggests it.

The producer requires a more complex response to a request to stop. It would be a shame to waste the pie it is waiting to place in the depository. On the other hand, perhaps the consumer has also been interrupted and will therefore never get around to retrieving the pie currently on deposit. So let's say the producer waits for 200 more milliseconds to see if that pie is taken and, if so, deposits its new pie before it terminates. Thus the no-action semicolon in the body of the Producer's `while` statement could reasonably be replaced by the following statement:

```
if (ThreadOp.pause (50))  // true when interrupted
{  ThreadOp.pause (200);
   if ( ! Resource.hasUneatenPie())
      Resource.setPie (dessert);
   return;
}
```

Note that the producer does not pay attention to whether another interrupt is sent during that 200 millisecond pause, since it plans to terminate in any case. Note also that, if either object wished to ignore any interrupt request, but still use the `sleep` method, it could simply replace the no-action semicolon body of its `while` statement by the following. This discards the returned boolean value:

```
   ThreadOp.pause (50);
```

Synchronization

If the Resource class method `getPie` is coded as follows, the logic could fail to produce the desired result:

```
public static Object getPie()
{  ready = false;
   return pie;
}  //=====================
```

The problem is that a consumer may execute `getPie` when a chocolate meringue pie is on deposit and a producer is waiting to deposit a coconut creme pie. The consumer executes `ready = false` in `getPie`. Then before it can execute `return pie`, the producer may test `hasUneatenPie`, which now returns `false`, so the producer executes `setPie`, thereby depositing the coconut creme pie before the chocolate meringue pie has been taken. That means that the consumer gets coconut creme, which is far inferior to chocolate meringue. The chocolate meringue is totally wasted.

A solution is to reverse the order of operations in the `getPie` method, so the consumer first takes the `pie` out of the depository and then sets the boolean `ready` to `false`:

```
public static Object getPie()
{  Object valueToReturn = pie;
   ready = false;
   return valueToReturn;
}  //=====================
```

This solution works if there is only one consumer. However, if there were several consumers, then as soon as a pie became available, two of them could try executing `getPie` at the same time, which could produce a food fight.

Java provides a solution for this messy problem. A class method that has the word `synchronized` in its heading can only be executed by one thread at a time. That is, when a Consumer object begins execution of the method, it is given a **lock** on the method. When another Consumer object tries to execute that same method, it is locked out; it must wait until the first Consumer object completes the method. Now, with some revisions of the logic given in this section, the program can manage several producers and several consumers properly.

Technical notes

The recommended order of the words in a method heading is shown by the following legal method heading. All those that come before `void` are optional. `native` means that the method is written in another language than Java and thus its body is to be found elsewhere than at this point:

```
public static final synchronized native void test()
```

Java has three rarely-used declaration modifiers: A field variable can be declared as `volatile`, which forces frequent synchronization, or as `transient`, which affects whether it is saved when an object is written to permanent storage, or as `strictfp`, which puts strictures on floating-point computations.

Exercise 11.49* Write the Resource method `public static void setPie (Object ob)` to avoid synchronization problems when there is only one producer.

*11.11 More On Math And Character (*Sun Library)*

This section briefly describes all Math methods other than those discussed in Chapter Six (`sqrt(x)`, `abs(x)`, `log(x)`, `exp(x)`, `min(x,y)`, `max(x,y)`, `pow(x,y)`, and `random()`) plus some additional methods from the Character wrapper class.

Math rounding methods

Math has five methods that round off a value in some way. `x` denotes a double value:

- `ceil(x)` returns a double that is the next higher whole-number value, except it returns `x` itself if `x` is a whole number. So `ceil(4.2)` is 5.0, `ceil(-4.2)` is -4.0, and `ceil(32.0)` is 32.0.
- `floor(x)` returns a double that is the next lower whole-number value, except it returns `x` itself if `x` is a whole number. So `floor(4.2)` is 4.0, `floor(-4.2)` is -5.0, and `floor(32.0)` is 32.0.
- `rint(x)` returns a double that is the closest whole-number value, except it returns an even number in case of a tie. So `rint(4.2)` is 4.0, `rint(4.7)` is 5.0, `rint(4.5)` is 4.0, and `rint(5.5)` is 6.0.
- `round(x)` returns the long value equivalent of `rint(x)`. It returns Long.MAX_VALUE or Long.MIN_VALUE if it would be otherwise out of range.
- `round(someFloat)` for a float parameter returns the int equivalent of `rint(someFloat)`. It returns Integer.MAX_VALUE or Integer.MIN_VALUE if it would otherwise be out of range.

Math trigonometric methods

The nine trigonometric methods all take double arguments and produce a double result. The angles are measured in radians, so `x = 3.14159` is about 180 degrees.

- `cos(x)` returns the cosine of `x`.
- `sin(x)` returns the sine of `x`.
- `tan(x)` returns the tangent of `x`.
- `acos(x)` returns the angle whose cosine is `x`, ranging from 0.0 through pi.
- `asin(x)` returns the angle whose sine is `x`, ranging from -pi/2 through pi/2.
- `atan(x)` returns the angle whose tangent is `x`, ranging from -pi/2 through pi/2.
- `atan2(x,y)` returns the angle whose tangent is `y/x`, ranging from -pi/2 through pi/2.
- `toDegrees(x)` returns the angle in degrees equivalent to `x` radians, which is equal to `x * 180.0 / Math.PI`.
- `toRadians(x)` returns the angle in radians equivalent to `x` degrees.

The one remaining Math method is `IEEEremainder(x, y)`, which returns the remainder of `x` divided by `y` as specified by the IEEE 754 standard.

Character methods

The following boolean class methods for Character objects can be quite handy. Unicode values outside the range 0 to 255 are not considered here:

- `Character.isDigit(someChar)` tells whether it is '0' through '9'.
- `Character.isLowerCase(someChar)` tells whether it is 'a' through 'z'.
- `Character.isUpperCase(someChar)` tells whether it is 'A' through 'Z'.
- `Character.isWhiteSpace(someChar)` tells whether `c` has Unicode 9-13 or 28-32.
- `Character.isLetter(someChar)` means it is either lowercase or uppercase.
- `Character.isLetterOrDigit(someChar)` means it is either a letter or a digit.

The following methods return a char value:

- `Character.toLowerCase(someChar)` returns the lowercase equivalent of a capital letter, and returns the unchanged parameter otherwise.
- `Character.toUpperCase(someChar)` returns the capital letter equivalent of a lowercase letter, and returns the unchanged parameter otherwise.

11.12 Review Of Chapter Eleven

About the Java language:

- A method may be declared as **final**, which means it cannot be overridden. The phrase **final class** means that the class cannot have any subclasses.
- Declaring a class as **abstract** means (a) you may replace the body of any non-final instance method by a semicolon if you declare that method as abstract; (b) every non-abstract class that extends it must implement all of the abstract methods. An abstract class may have instance and class variables, constructors, and instance and class methods. Class methods in an abstract class cannot be abstract.
- The heading `public interface X` means X cannot contain anything but non-final instance method headings (with a semicolon in place of the body) and final class variables. An object class with the heading `class Y implements X` must define all methods in that **interface** unless it is an abstract class. A class may implement many interfaces (use `implement X,U,Z`) but may subclass only one class.
- The compiler binds a method call to a particular method definition when it can, namely, for a class method or for a final instance method. This is **early binding**, which reduces execution time compare with **late binding** (done at runtime).
- The four primitive integer types are long (8 bytes), int (4 bytes), **short** (2 bytes), and **byte** (1 byte). One byte of storage is 8 bits, so there are 256 different possible values for one byte. The two primitive decimal number types are double (8 bytes, 15 digits) and **float** (4 bytes, 7 digits). The other two **primitive types** are boolean and char.
- The operator **instanceof** can be used between an object variable and a class name; it yields a boolean value. `x instanceof Y` is true when x is not `null` and is an object of class Y or of a subclass of Y or (if Y is an interface) of a class that implements Y. The ! operator takes precedence over the instanceof operator, so a phrase of the form `! X instanceof Y` is never correct; use parentheses.

About the six Number subclasses:

- The six Number **wrapper classes** Double, Float, Integer, Long, Short, and Byte are in `java.lang` and are Comparable. The twelve methods given below for the Long class apply to the other five Number wrapper classes with the obvious changes:
- `Long.parseLong(someString)` returns a long value or throws a NumberFormatException.
- `new Long(someString)` creates a Long object parsed from the String. It throws a NumberFormatException if the string is badly-formed.
- `new Long(primitive)` creates a Long object from a given long value.
- `someLongObject.longValue()` returns the long equivalent.
- `someLongObject.doubleValue()` returns the double equivalent
- `someLongObject.intValue()` returns the int equivalent.
- `someLongObject.floatValue()` returns the float equivalent.
- `someLongObject.shortValue()` returns the short equivalent.
- `someLongObject.byteValue()` returns the byte equivalent.
- `someLongObject.toString()` overrides the Object method.
- `someLongObject.equals(someObject)` tells whether the corresponding primitive values are equal.
- `someLongObject.compareTo(someObject)` returns an int with the usual meaning: positive if the executor is larger, negative if it is smaller.
- `Integer.toString(n,16)` gives the **hexadecimal** (base 16) form of the int value n. Use 2 or 8 in place of 16 for **binary** (base 2) or **octal** (base 8).
- `Integer.parseInt(someString, 16)` produces the int equivalent of the hexadecimal digits in `someString`. Use 2 or 8 for binary or octal, respectively.

About the java.lang.Character class:

- `new Character(someChar)` gives the object equivalent of the primitive value given as a parameter.
- `someCharacter.charValue()` converts back from object to primitive value.
- `someCharacter.toString()` returns the same value as `charValue()`.
- `someCharacter.equals(someObject)` overrides the Object `equals`.
- `someCharacter.compareTo(someObject)` returns an int with the usual meaning: positive if the executor is larger, negative if it is smaller.

About the java.lang.Boolean class:

- `new Boolean(primitive)` gives the object equivalent of the primitive boolean value given as a parameter.
- `new Boolean(someString)` yields `Boolean.TRUE` if the parameter is "true", ignoring case, otherwise it yields `Boolean.FALSE`.
- `someBooleanObject.booleanValue()` returns the primitive form of the Boolean object.
- `someBooleanObject.toString()` returns "true" or "false" as appropriate.
- `someBooleanObject.equals(someObject)` overrides the Object `equals`.

Answers to Selected Exercises

11.2
```
public static Numeric min3 (Numeric one, Numeric two, Numeric three)
{   if (one.compareTo (two) < 0)
        return one.compareTo (three) < 0  ?  one  :  three;
    else
        return two.compareTo (three) < 0  ?  two  :  three;
}
```
11.3
```
public Numeric smallest (BufferedReader source)
{   try
    {   Numeric valueToReturn = valueOf (source.readLine());
        if (valueToReturn == null)
            return null;
        Numeric data = valueOf (source.readLine());
        while (data != null)
        {   if (data.compareTo (valueToReturn) < 0)
                valueToReturn = data;
            data = valueOf (source.readLine());
        }
        return valueToReturn;
    }catch (Exception e)
    {   return null;
    }
}
```
11.5
```
public boolean equals (Object ob)
{   if (! ob instanceof Person)
        return false;
    Person given = (Person) ob;
    return this.itsBirthYear == given.itsBirthYear
        && this.itsLastName.compareTo (given.itsLastName) == 0;
}
```
11.6
```
public class Tuna extends Swimmer
{   public void swim()
    {   System.out.println ("tuna is swimming");
    }
    public void eat (Object ob)
    {   if (ob instanceof Swimmer)
            System.out.println ("tuna eats " + ob.toString());
        else
            System.out.println ("tuna is still hungry");
    }
    public String toString()
    {   return "tuna";
    }
}
```
11.10 5 = 4 + 1, 11 = 8 + 2 + 1, 260 = 256 + 4. So:
In binary; 101, 1011, 100000100.
In octal: 5, 13, 404 (the last since 256 = 4 * 8 * 8).
In hexadecimal: 5, B, 104.

11.11 F is 15. 5D is 5*16 + 13 = 93.
ACE is 10 * 256 + 12 * 16 + 14 = 2560 + 192 + 14 = 2766.

11.12
```
public Numeric valueOf (String par)
{   if (par == null)
        return null;
    int k = par.indexOf ('/');
    return k == -1  ?  null  :  new Fraction (Integer.parseInt (par.substring (0, k)),
                                              Integer.parseInt (par.substring (k + 1)));
}
```
11.13 ZERO is defined using the constructor. That definition will be applied by the constructor when the program begins. How can the constructor return a value that does not yet exist? Besides, a constructor does not have a return type.

11.14 The specification for the equals method is that it not throw an Exception. But the Numeric add, subtract, and multiply methods are to throw an Exception if the parameter is not of the right type.

11.15 For subtract, simply replace the plus sign in the add method by the minus sign.

11.16
```
public int compareTo (Object ob)
//  You can return the numerator of the result of subtracting this minus ob:
{   return this.itsUpper * ((Fraction) ob).itsLower
        - this.itsLower * ((Fraction) ob).itsUpper;
}
```

11.19
```
public boolean equals (Object ob)
{   return ob instanceof Complex && ((Complex) ob).itsReal == this.itsReal
                                 && ((Complex) ob).itsImag == this.itsImag;
}
```
11.20
```
public Numeric multiply (Numeric par)
{   Complex same = (Complex) par;
    return new Complex (this.itsReal * same.itsReal - this.itsImag * same.itsImag,
                        this.itsReal * same.itsImag + this.itsImag * same.itsReal);
}
```
11.21 Change the (Complex) cast to a (Numeric) cast. The compiler accepts it because Numeric declares doubleValue. The runtime system will then choose the right subclass's doubleValue method.

11.24
```
public double doubleValue()
{   double total = itsItem[0];
    for (int k = 1;  k < MAX;  k++)
        total = total * BILLION + itsItem[k];
    return total;
}
```
11.25
```
public boolean equals (Object ob)
{   if (! (ob instanceof VeryLong))
        return false;
    VeryLong that = (VeryLong) ob;
    for (int k = 0;  k < MAX;  k++)
        if (this.itsItem[k] != that.itsItem[k])
            return false;
    return true;
}
```
11.26 You could insert the following directly after the super() statement:
```
long max = LONGBILL * LONGBILL - 1;
if (left < 0 || left > max || mid < 0 || mid > max || right < 0 || right > max)
{   left = 0L;
    mid = 0L;
    right = 0L;
}
```
11.34
```
public static double sumValues (Numeric[ ] item)
{   double valueToReturn = 0;
    for (int k = 0;  item[k] != null;  k++)
        valueToReturn += item[k].doubleValue();
    return valueToReturn;
}
```
11.35
```
public static int size (Object [ ] item)  // better if Object, not Numeric
{   int k = 0;
    while (item[k] != null)
        k++;
    return k;
}
```
11.36
```
public static boolean allSmall (Numeric[ ] item, Numeric par)
{   for (int k = 0;  item[k] != null;  k++)
        if (item[k].compareTo (par) >= 0)
            return false;
    return true;
}
```
11.37
```
public static in indexSmallest (Numeric[ ] item)
{   int valueToReturn = 0;
    for (int k = 1;  item[k] != null;  k++)
        if (item[k].compareTo (item[valueToReturn]) < 0)
            valueToReturn = k;
    return valueToReturn;
}
```
11.38
```
public static void delete (Object[ ] item, int n)  // best if Object, not Numeric
{   if (n >= 0 && n < item.length)  // no need to verify it is non-null
        for ( ;  item[n] != null;  n++)
            item[n] = item[n + 1];
}
```
11.39
```
public static Numeric[] getComplexes (Numeric[ ] item)
{   Numeric[ ] valueToReturn = new Numeric [item.length];
    int next = 0;
    for (int k = 0;  item[k] != null;  k++)
        if (item[k] instanceof Complex)
        {   valueToReturn[next] = item[k];
            next++;
        }
    return valueToReturn;
}
```

12 Files And Multidimensional Arrays

Overview

In this chapter you will develop software to manage information about email messages. The software reads information from hard-disk files into multi-dimensional arrays. None of the material in this chapter is required for other chapters. Be sure to read the first three sections of Chapter Ten on Exceptions before reading this chapter.

- Sections 12.2-12.3 introduce basic file-handling Sun library classes.
- Section 12.4 presents the StringTokenizer and StreamTokenizer library classes.
- Section 12.5 develops multi-dimensional arrays in detail.
- Sections 12.6 completes the development of Version 1 of the email software.
- Sections 12.7-12.9 cover more topics related to arrays and files.
- Sections 12.10-12.11 have some information on rarely-used language features and on Java bytecode.

12.1 Analysis And Design For The Email Software

Your employer wants you to analyze the email messages that are sent between employees. He wants this for various reasons, among which are:

- He is worried that some people are overburdening the email system.
- He needs to monitor content of messages to check for discriminatory acts that could cause him a lot of grief with the government if he does not detect them and do something about them.
- He wants to see if some employees are not doing their jobs, as indicated by a very low number of email messages sent.

The data describing the email is in files that are stored on a hard disk. Each day's email is in a separate file. Such a file contains two lines for each email sent that day. The first line is two words, the sender followed by the receiver (each employee has a one-word email name). The second line is the subject line of the email. The company only has forty-some employees, so one day's file only contains one or two thousand email messages. Your job is to process this file and produce various statistics describing it.

The first program you are to write is to read one day's email file and produce the following information:

- How many emails were sent that day.
- How many employees sent email that day.
- Who did not send any email at all that day.
- What was the average number of email messages sent per person.
- Which people sent at least twice as many emails as they received.
- Which people received at least twice as many emails as they sent.
- What kind of idiot sends himself email.
- Which people sent the least amount of email and which people sent the most.
- Which people were sent the least amount of email and which people were sent the most.

Later programs will look at the subject lines themselves, even though this first program does nothing with them. So the object design should include storing the subject lines. The top-level design of the program is easily developed from this analysis, as shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic
1. Ask the user for the name of the file that contains one day's data and the name of the output file where the results are to be stored.
2. Open the data file for reading.
3. Read in all the information about email messages and store it in some kind of database.
4. Ask questions of that database and print out the answers to the output file.

First you need to know how to work with text files stored on a hard disk. The next two sections introduce standard library classes for this. The Sun standard library also has classes for working with files containing numbers and String objects rather than just characters; those concepts are introduced in later sections of this chapter.

12.2 FileReader And BufferedReader For Input

A text file is a file stored on the hard disk as a plain-text file. That means that only the characters to be displayed, plus newline and tab characters, are in the file. In Word or Wordpad, make sure you select type `txt` when storing the file for a program to read, not type `doc`. The `doc` type contains special coding that specifies page numbering, margins, and other extra information that the Word program uses.

Reading from a text file

You can create an object to read characters from a text file. The statement

```
FileReader input = new FileReader ("data.txt");
```

creates a FileReader object connected to the file stored with the name `data.txt` in the current folder on the hard disk. However, this **FileReader** object is only capable of reading one character at a time or else a large array of characters. For instance, `input.read()` returns the next char value (or -1 if at the end of the file). And if `lotsOfChars` is an array of char values, `input.read (lotsOfChars, 100, 500)` attempts to read 500 characters and store them in `lotsOfChars` starting at index 100; it returns the number of chars actually read (or -1 if at the end of the file).

This is convenient for some purposes, but not for text files. A FileReader has no method for reading one line of characters at a time. The statement

```
BufferedReader file = new BufferedReader (input);
```

creates out of that FileReader a **BufferedReader** object that has a `readLine` method, as shown in Figure 12.1. This method gets one entire line of input, discards the end-of-line marker, and returns the rest as a String of characters. If you have already reached the end of the file, this method returns the `null` value. You get the next line of input by using a statement such as the following:

```
String s = file.readLine();
```

The TextFileOp class (not Sun standard library)

It is useful to have a class to collect several such independent class methods that work with text files. Call this utilities class **TextFileOp**. A start on this class is in Listing 12.1 (on the second page following). The `findDigits` method in the upper part of Listing 12.1 shows how you can read lines from a file to find the first line that begins with a digit. It tests `s != null` to make sure end-of-file has not been reached and tests `s.length() > 0` to ignore a blank line.

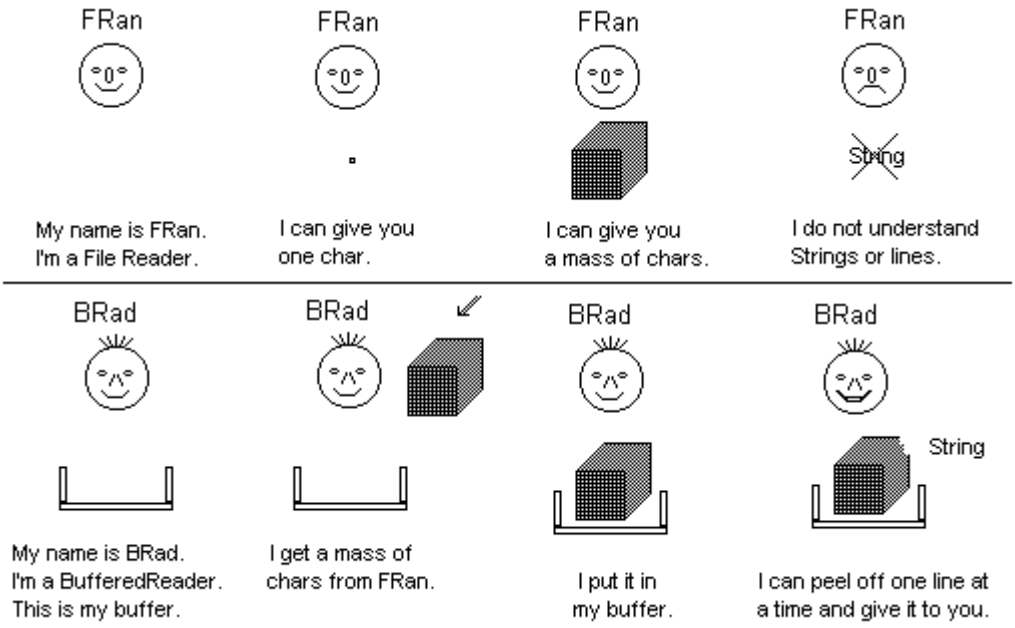


Figure 12.1 BufferedReader mediates between you and FileReader

The BufferedReader class, the IOException class, and other classes introduced in this section and the next, are all in the java.io package. So you need `import java.io.*` at the top of a compilable file that uses these classes (or the equivalent).

Another useful independent method would be a `compareFiles` method that determines whether two text files are identical in their contents. This requires that you repeatedly read one line from each file until you get to the end of the files. If you ever see a pair that are not equal, then the files are not identical. A reasonable structured design of this method is in the accompanying design block. The lower part of Listing 12.1 contains the coding for this `compareFiles` method.

DESIGN for the compareFiles method

1. Create two file objects connected to existing physical text files.
2. Read one line from each file.
3. Repeat the following until one of the files is finished and thus the read fails...
    - 3a. If the two current lines are not equal then...
        - 3aa. Return `false` as the result.
    - 3b. Read one more line from each file.
4. Return `true` if both files are finished, otherwise return `false`.

Reading from the keyboard

You can create an object to read characters from the keyboard. The statement

```
InputStreamReader input = new InputStreamReader (System.in);
```

creates an **InputStreamReader** object connected to the keyboard, because the System class has a InputStream object named `System.in` which is connected to the keyboard, and the InputStreamReader class has a constructor with an InputStream parameter. However, this InputStreamReader object can only read one character at a time (`read()` returns an int value 0 to 65535) or a large array of characters (`read(char[])` returns the number of chars read).

Listing 12.1 The TextFileOp utilities class; more methods later

```
import java.io.*;  // for FileReader, IOException, and others

public class TextFileOp
{
   /** Find the first line that begins with a digit. */

   public static String findDigit (String fileName)
                                    throws IOException
   {  BufferedReader fi = new BufferedReader
                         (new FileReader (fileName));
      String si = fi.readLine();
      while (si != null)
      {  if (si.length() > 0 && si.charAt (0) >= '0'
                             && si.charAt (0) <= '9')
            return si;
         si = fi.readLine();
      }
      return "";
   }  //======================

   /** Tell whether the two files have the same contents. */

   public static boolean compareFiles (String one, String two)
                                       throws IOException
   {  BufferedReader inOne = new BufferedReader
                            (new FileReader (one));
      BufferedReader inTwo = new BufferedReader
                            (new FileReader (two));
      String s1 = inOne.readLine();
      String s2 = inTwo.readLine();

      while (s1 != null && s2 != null)
      {  if ( ! s1.equals (s2))
            return false;
         s1 = inOne.readLine();
         s2 = inTwo.readLine();
      }
      return s1 == s2;  // true only when both are null
   }  //======================
}
```

BufferedReader's constructor accepts any **Reader** object; FileReader and InputStreamReader are both subclasses of the abstract Reader class. Figure 12.2 shows the relationships among these classes. So the statement

```
BufferedReader keyboard = new BufferedReader (input);
```

creates out of that InputStreamReader an object that has a `readLine` method, just as for the file object, because both are BufferedReader objects. Now you can read whole lines at a time instead of character-by-character. The `new BufferedReader` call is analogous to an upgrade from coach to business-class -- you get the same end result, but the way you do it is much more comfortable, as in

```
String s = keyboard.readLine();
```

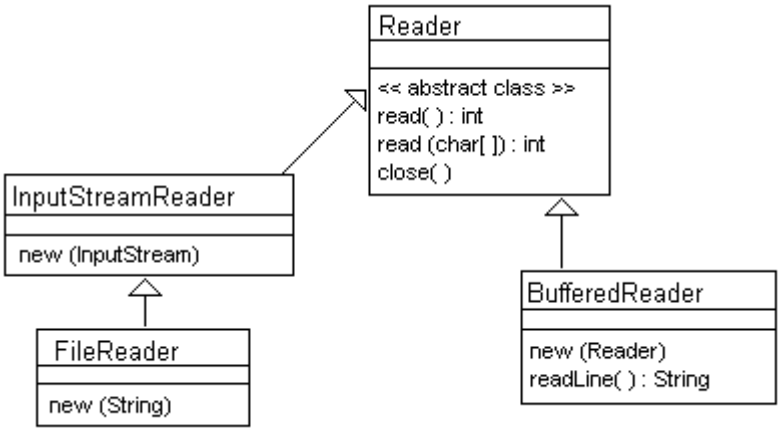


Figure 12.2 UML class diagram for Reader and some subclasses

By way of illustration, you could replace the first statement in the `findDigit` method of Listing 12.1 by the following if you want the method to be able to read from either the keyboard or a text file, depending on whether the `filename` is the empty string:

```
BufferedReader fi = fileName.length() > 0
     ? new BufferedReader (new FileReader (fileName))
     : new BufferedReader (new InputStreamReader (System.in));
```

Handling IOExceptions

Any call of the `readLine` method of the BufferedReader class could possibly throw an IOException which you must handle. So can `new FileReader`. Exception-handling is discussed in detail at the beginning of Chapter Ten. What you need to know about it for here is that you have four ways of handling this IOException:

1. Learn how to wrap each call of `readLine` or `new FileReader` in a try/catch statement. This is discussed in Chapter Ten.
2. Write a class to encapsulate the entire problem. The Buffin class in Chapter Ten does the wrapping for you -- the Buffin class is like an upgrade to first class. None of its methods throws an Exception under any circumstances.
3. If the proper handling of the Exception depends on circumstances known only to the method that calls your method, put `throws IOException` in the heading of your method and let the calling method use a try/catch statement to handle it properly.
4. Put the phrase `throws IOException` in the heading of every method that calls `readLine` or that calls a method that has that phrase in its heading. Then a failure of a file access immediately terminates the `main` method with an appropriate message. This is the appropriate handling for the programs in this chapter.

A **buffer** is a storage location where input values are kept until asked for. A BufferedReader object obtains roughly 8192 characters at a time from the file and stores them in its private buffer in RAM. Each time you call the BufferedReader's `readLine` method, it gives you some of the characters from its buffer. As a consequence, you may have only one retrieval of information from the physical hard disk file for each hundred or so times that you execute `readLine`. Buffering is a much more efficient way of getting information than reading one or a few characters at a time.

Exercise 12.1 Write a TextFileOp method `public static void findCaps (String name)`: It opens a text file with a given name and then prints to `System.out` each line of that file that begins with a capital letter.

Exercise 12.2 Write a method `public int readInt()` for a subclass of BufferedReader: The executor reads a single line of input from the file, converts it to an int value, and returns that value. Return 0 on any NumberFormatException.
Exercise 12.3 Write a TextFileOp method `public static int numChars (String name)`: It returns the number of characters in the text file of the given name.
Exercise 12.4 Revise the preceding exercise to return instead the average number of characters per line in the text file of the given name.
Exercise 12.5 Write a TextFileOp method `public static boolean descending (String name)`: It tells whether the lines in the file of the given name are in descending order (i.e., if `s1` is one line and `s2` is the line after it, then `s1.compareTo(s2) >= 0`).
Exercise 12.6* Write an independent method `public static void alternate (String one, String two)`: It opens two text files with the given names and then prints the lines of those two files to `System.out`, alternating between them. That is, first line#1 of the first file, then line#1 of the second, then line#2 of the first, then line#2 of the second, etc. Stop when one file is emptied.
Exercise 12.7* Write a TextFileOp method `public static String firstOne (String name)`: It returns the lexicographically first line of the text file with the given name (i.e., the line `s1` for which `s1.compareTo(s2) <= 0` for each line `s2` in the file). It returns `null` if the file is empty.
Exercise 12.8** Write an application program: It gets the name of a text file from the user at the keyboard, opens the file (but opens the keyboard if the name does not have positive length), and prints every line that is the same as the line before it in the file.
Exercise 12.9** Write an independent method `public static int numWords (String name)`: It returns the number of words in the text file of the given name. The end of a word is a non-whitespace character that is followed by either whitespace or the end of the line. Whitespace is any character that is less than or equal to a blank.

12.3 FileWriter And PrintWriter For Output

Writing to a text file on a hard disk is slightly simpler than reading from one. You first create a PrintWriter object using a statement as follows, with whatever String value you want for `filename`. The FileWriter constructor can throw an IOException:

```
PrintWriter out = new PrintWriter (new FileWriter (filename));
```

This **opens** the physical file for output. Printing starts at the beginning of the file, so you lose whatever was in that disk file before you opened it. However, an alternative is to open the file for appending data: `new FileWriter (fileName, true)` means that whatever is already in the file is preserved, and what you write is added after it.

A **FileWriter** object can write one character at a time, or a whole array of characters, but not a String of characters at once. A **PrintWriter** has the `print` and `println` methods that the `System.out` variable has, to write many characters at a time. That is why you "upgrade" to the PrintWriter class. In particular, these two statements

```
out.print (whatever);
out.println (whatever);
```

can be used to print a String, character, or number. The `println` method outputs a newline character `'\n'` after printing its parameter, but the `print` method does not. Both FileWriter and PrintWriter are subclasses of the abstract **Writer** class.

It is quite important that you call the `close` method of a PrintWriter object when you are finished writing to the file, before you exit the program. If you do not, you may lose the last few lines of output. The reason is that a PrintWriter may buffer the output to cut

down on the number of times it has to access the physical file. Closing the file **flushes** the buffer to the physical file (sends all output to the file that the buffer is holding onto as an efficiency measure):

```
out.close();
```

Writing to the terminal window

It is sometimes useful to have a PrintWriter object that writes to the terminal window. This can be done with the following statement. When you use this `screen` object to print something that you want to appear before a `readLine` method is executed, you should call `screen.flush()` before calling the `readLine` method:

```
PrintWriter screen = new PrintWriter (System.out);
```

A PrintWriter never throws an IOException. So there is no real need for an additional wrapper class analogous to the Buffin wrapper around a BufferedReader. Figure 12.3 shows the standard library classes mentioned in this section. The `write(int)` method accepts a single int value that it casts to a char to be written to the file.

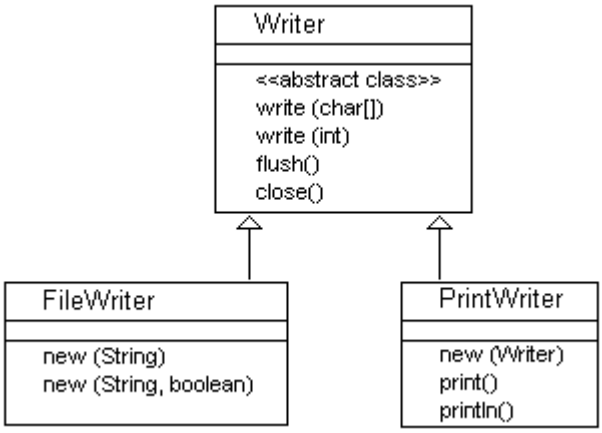


Figure 12.3 UML class diagram for text output files

More TextFileOp methods

Listing 12.2 (see next page) illustrates more class methods that work with files. The `createRandoms` method repeatedly picks a random integer in the range from 0000 to 9999 inclusive and writes it to the specified file. The `file.print` call in the inner for-loop produces a group of four random integers followed by tab characters, so that the numbers will be separated in the file and appear in nice columns when printed. Then `file.println` call prints the fifth random number on the line and starts a new line. The outer for-loop produces the number of lines requested. The file is then closed.

The `mergeTwo` method illustrates logic for sorting information that is too massive to store in RAM. This method takes two files that have their lines in ascending order and produces a new combined file containing all the lines in both files, all in ascending order. Each iteration of the loop determines which of the two String values, `s1` or `s2`, is to be written next to the merged file. It will be the string from the first file if the second file has been completely read in (signaled by `s2` being `null`) or if neither file has been completely read in and `s1` comes before `s2`. In all other cases, the string from the second file will be written next. The loop continues until both input files have been read.

Listing 12.2 More independent class methods working with files

```
// public class TextFileOp, continued

   /** Print the specified number of lines of random 4-digit
    *  non-negative integers, 5 integers per line. */

   public static void createRandoms (String name, int numLines)
                                     throws IOException
   {  PrintWriter file = new PrintWriter (new FileWriter (name));
      java.util.Random randy = new java.util.Random();
      for (int k = 0;  k < numLines;  k++)
      {  for (int i = 0;  i < 4; i++)
            file.print (randy.nextInt (10000) + " \t");
         file.println (randy.nextInt (10000));
      }
      file.close();
   }  //=====================

   /** Read from two BufferedReaders known to have their lines in
    *  ascending order.  Write all lines to a file named outName
    *  so that the file has its lines in ascending order.  */

   public static void mergeTwo (BufferedReader inOne,
         BufferedReader inTwo, String outName) throws IOException
   {  PrintWriter merged = new PrintWriter
                           (new FileWriter (outName));
      String s1 = inOne.readLine();
      String s2 = inTwo.readLine();

      while (s1 != null || s2 != null)
         if (s2 == null || (s1 != null && s1.compareTo (s2) < 0))
         {  merged.println (s1);
            s1 = inOne.readLine();
         }
         else  // the line from inTwo is next in order
         {  merged.println (s2);
            s2 = inOne.readLine();
         }
      merged.close();
   }  //=====================
```

Exercise 12.10 Write a TextFileOp method `public static void copy (BufferedReader infile, String outName)`: The method copies every line in the BufferedReader to the text file named `outName`.

Exercise 12.11 Write a TextFileOp method `public static void underForty (String inName, String outName)`: It opens a BufferedReader and a PrintWriter, then it writes to the output file every line in the input file that has less than 40 characters.

Exercise 12.12* Revise the `compareFiles` method in Listing 12.2 to return the first line you see in the first file that does not exactly match the corresponding line of the second file. Return null if the files are identical.

Exercise 12.13* Rewrite the `createRandomNumbers` method in Listing 12.2 to use a single for-loop, with an if-statement inside the for-loop to call either `print` or `println`.

Exercise 12.14* Write a TextFileOp method `public static void capsAndSmalls (String inFile, String outOne, String outTwo)`: It reads from one named text file, prints to the second named text file each line that begins with a capital letter, and prints to the third named text file each line that begins with a lowercase letter.

Exercise 12.15** Write a TextFileOp method `public static void reverse (String fileName)`: It reads all the lines in the text file of that name and writes them back to the same file in reverse order. The file has at most 1000 lines. Use an array to store the lines between reading and writing.

12.4 The StringTokenizer And StreamTokenizer Classes

The design for the main logic of the Email software is repeated below for your convenience:

STRUCTURED NATURAL LANGUAGE DESIGN for the main logic

1. Ask the user for the name of the file that contains one day's data and the name of the output file where the results are to be stored.
2. Open the data file for reading.
3. Read in all the information about email messages and store it in some kind of database.
4. Ask questions of that database and print out the answers to the output file.

This design indicates the need for several kinds of objects to implement this design:

- A BufferedReader object for input.
- A PrintWriter object for output.
- A Message object to represent one email message.
- An EmailSet object to store the number of emails from each person to each other person.

You already have BufferedReader and PrintWriter objects, so next consider what a Message object has to be able to do. You need to construct a Message object that contains the information about the next email message obtained from the BufferedReader object. And you need to be able to ask that Message object for its sender, its receiver, and its subject heading.

How will you recognize, when you construct a Message object for the next email in the BufferedReader object, that you have already come to the end of the file? A reasonable method is to have the constructor return a Message with a `null` sender when the end of the input is reached. That is, `someMessage.getSender()` returns `null` to signal the Message had no input to get data from. So we can use this sort of logic:

```
Message data = new Message (file);
while (data.getSender() != null)
{  // various statements go here to process one data value
   data = new Message (file);
}
```

The StringTokenizer methods

The Message constructor uses a new standard library class named StringTokenizer, from the `java.util` package, which includes several highly useful methods for breaking up a string of characters into its individual words:

- `new StringTokenizer (inputString)` establishes the parameter as the input string of characters for the executor. It throws a NullPointerException if the parameter is `null`.

- `someStringTokenizer.nextToken()` returns the next available token from the input string and advances in the sequences of tokens. It throws a java.util.NoSuchElementException if no more tokens are available.
- `someStringTokenizer.hasMoreTokens()` tells whether the input string has any tokens left that have not yet been returned by `nextToken`.
- `someStringTokenizer.countTokens()` tells how many more times you may call `nextToken` for this StringTokenizer object.

A StringTokenizer easily breaks up a String value into parts separated by blanks. These parts are called tokens, which is a more general term than "words". Specifically, a **token** within a String value is a sequence of non-whitespace characters with whitespace on each side; the beginning and end of the String value count as whitespace for purposes of this definition. Using a StringTokenizer is much easier than writing methods such as `trimFront` and `firstWord` in the StringInfo class of Chapter Six. (Whitespace includes blanks, tab characters, newline characters, and form feeds to start a new page.) Listing 12.3 illustrates the use of a StringTokenizer object for the Email software.

Listing 12.3 The Message object class for the Email software

```
import java.util.StringTokenizer;
import java.io.BufferedReader;
import java.io.IOException;

public class Message extends Object
{
   private String itsReceiver = null;
   private String itsSubject = null;
   private String itsSender = null;


   /** Create a Message object from the next two lines of the
    *  source file.  Set sender to null at end-of-file. */

   public Message (BufferedReader source) throws IOException
   {  String s = source.readLine();
      if (s != null)
      {  StringTokenizer line = new StringTokenizer (s);
         if (line.hasMoreTokens())
         {  itsSender = line.nextToken();
            if (line.hasMoreTokens())
            {  itsReceiver = line.nextToken();
               itsSubject  = source.readLine();
            }
         }
      }
   }  //=====================

   public String getSender()
   {  return itsSender;
   }  //=====================

   public String getReceiver()
   {  return itsReceiver;
   }  //=====================

   public String getSubject()
   {  return itsSubject;
   }  //=====================
}
```

The following logic illustrates the use of `hasMoreTokens` to count all tokens in a String that start with a numeric digit. The if-condition illustrates the kind of hard-to-follow logic some students produce, thinking it is classier. You would not do anything like that in your own programs, would you? An exercise has you clean it up:

```
public static int countNumerals (String s)    // independent
{  int count = 0;
   StringTokenizer data = new StringTokenizer (s);
   while (data.hasMoreTokens())
      if ((data.nextToken().charAt (0) - '0' + 10) / 10 == 1)
         count++;
   return count;
}  //=====================
```

The StreamTokenizer class

If you need more flexibility than what StringTokenizer offers, `java.io` contains the **StreamTokenizer** class, which reads in a file and separates it into tokens. It can be set to skip Java comments and to treat everything from a quote mark down to its end-quote as a single token, with proper adjustments for the backslash character. The following are a start on understanding StreamTokenizer, but you have to read the javadoc description in the java documentation on your hard disk or online to use it properly:

- `new StreamTokenizer(someReader)` creates the file processing object.
- `someStreamTokenizer.nextToken()` returns an int value such as TT_WORD (the current token is a word), TT_NUMBER (the current token is a number), TT_EOF (end-of-file has been reached), or the int equivalent of the one-character token.

Exercise 12.16 Revise the Message constructor in Listing 12.3 to return a Message with `itsSender` being `null` when the first line of the input has only one token.
Exercise 12.17 Revise the body of the while-statement in the `countNumerals` method to be much clearer to the reader.
Exercise 12.18 Explain why the `countNumerals` method in the text cannot test the simpler expression `(data.nextToken().charAt (0) - '0') / 10 == 0`.
Exercise 12.19* Write an independent method `public static int numTokens (String name)`: It returns the number of tokens in the text file of the given name.
Exercise 12.20* Write an independent method `public static boolean ascending (String given)`: It tells whether all of the tokens in a given String parameter are in ascending order. Use StringTokenizer.

12.5 Defining And Using Multidimensional Arrays

One part of the Email main logic requires storing the information from a Message object in some sort of EmailSet database object. This EmailSet object class needs a constructor and a method for adding the information from a single Message object. For this Version 1 of the software, that information is just a count of emails between two specific people without trying to store the subject line.

It also makes sense to have methods to retrieve the total number of Message objects added so far, the total number of employees who were sending email that day, and the total number sent from one specific person `from` to another specific person `to`. These methods supply all you need to calculate the average number sent.

The standard way of storing one piece of information for each <from,to> pair of values is a two-dimensional array. But this requires that <from,to> be non-negative integers rather

than Strings. If a Message object could somehow produce integer-valued codes for the employees, rather than their actual email names, you could use the structure shown in Listing 12.4 for the EmailSet objects: `itsItem[from][to]` is the number of emails sent from `from` to `to`.

Listing 12.4 The EmailSet object class, some methods postponed

```
public class EmailSet extends Object
{
   public static final int MAX = 50;
   ////////////////////////////////
   private int[][] itsItem = new int[MAX][MAX];
            // all components of the array are initially zero
   private int itsSize = 0;
            // non-zero entries are indexed 0..itsSize-1
   private int itsNumEmails = 0; // total of all non-zero entries


   /** Add the given message to the set of stored emails. */

   public void add (Message given)
   {  // logic to find <from,to> codes for sender and receiver
      itsItem[from][to]++;
      itsNumEmails++;
   }  //=====================

   /** Return the total number of emails sent. */

   public int getNumEmails()
   {  return itsNumEmails;
   }  //=====================

   /** Return the total number of employees sending email. */

   public int getNumEmployees()
   {  return itsSize;
   }  //=====================

   /** Return the number of emails sent from from to to. */

   public int numSent (int from, int to)
   {  return itsItem[from][to];
   }  //=====================
}
```

Defining multi-dimensional arrays

You declare an N-dimensional array the same way as a one-dimensional array, except that you give N empty pairs of brackets where you would otherwise give just one. In the constructor phrase of the new array, you give N pairs of brackets, each filled with the number of different index values you want to use in that position.

Examples: To declare and construct a 2-dimensional array to record a money amount owed to any of 7 different loan companies by any of 20 students, you could use this:

```
   double [] [] amountOwed = new double [7] [20];
```

So `amountOwed[5][2]` is the amount owed to loan company #5 by student #2, numbering from 0 on up. Figure 12.4 shows an illustration of this.

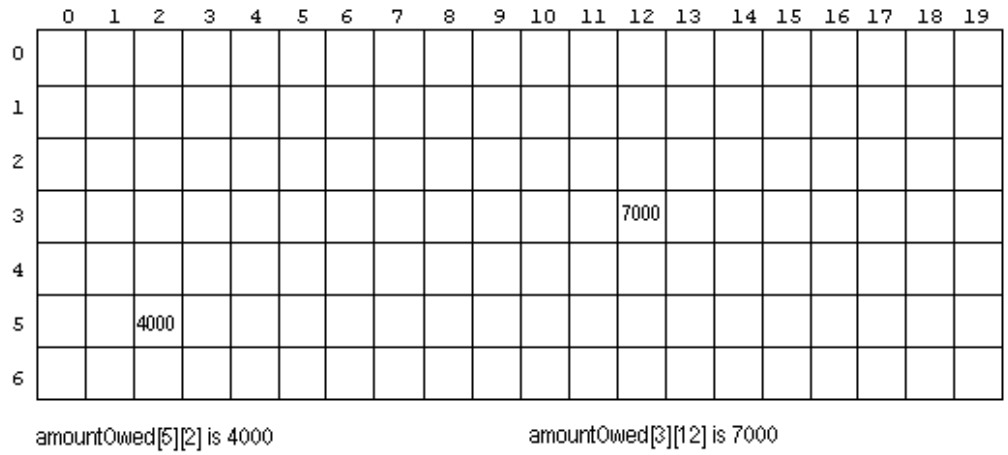


Figure 12.4 Two-dimensional array of double values

To record a Color value for each pixel in a display 300 pixels wide and 400 pixels tall, you could use this:

```
Color [] [] itsColor = new Color [300] [400];
```

To record whether any of 12 different computer professionals are competent in the use any of 30 different pieces of software, you could use this:

```
boolean [] [] isCompetent = new boolean [12] [30];
```

To record the character in any of 40 different books on a bookshelf, on any of up to 300 pages in the book, on any of the 45 different lines on that page, in any of the 65 different character positions (reading left to right) on that line, you could use this:

```
char [] [] [] [] letter = new char [40] [300] [45] [65];
```

However, Java programmers virtually never use an array of more than two dimensions. It is almost always clearer to have a one- or two-dimensional array of objects that embody several more dimensions. For instance, a Color is actually a sequence of three int values ranging 0 to 255, each representing an RGB value (red/green/blue), so that the `itsColor` array declared above is actually a 5-dimensional array in disguise.

The book example could be done as follows, where a `bookshelf` is a 2-dimensional array of Pages, and a Page is an object that contains a rectangular array of letters:

```
Page[][] bookshelf = new Page[40][300];
class Page { private char[][] letter = new char[45][65];...}
```

Applications to EmailSet objects

All indexes in all Java arrays start from 0 and go on up. An EmailSet object stores the number of emails sent from person `from` to person `to` in the variable `itsItem[from][to]`, part of a two-dimensional array of ints. The following EmailSet methods all adapt logic you have seen before with one-dimensional arrays or other kinds of sequences. Figure 12.5 shows what this array of int values might contain if there were only five employees.

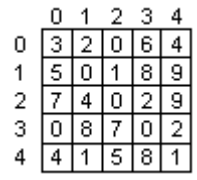


Figure 12.5

You may ask an EmailSet object for the total number of emails sent to a person with a particular code number, which it could answer if it had this method, running through all permissible values of the first index:

```
public int numSentTo (int code)   // in EmailSet
{  int count = 0;
   for (int from = 0;  from < itsSize;  from++)
      count += itsItem[from][code];
   return count;
}  //======================
```

You could ask an EmailSet object for the smallest number of emails sent from a person with a particular code, which it could answer if it had this method, running through the values of the second index:

```
public int minSentFrom (int code)  // version 1; in EmailSet
{  int min = itsItem[code][0];
   for (int to = 1;  to < itsSize;  to++)
      if (min > itsItem[code][to])
         min =  itsItem[code][to];
   return min;
}  //======================
```

Some people find this logic easier to understand when it is written as follows:

```
public int minSentFrom (int code)  // version 2; in EmailSet
{  int[] oneRow = itsItem[code];  // name the row, for clarity
   int min = oneRow[0];
   for (int to = 1;  to < itsSize;  to++)
      if (min > oneRow[to])
         min = oneRow[to];
   return min;
}  //======================
```

You could ask an EmailSet object for the code of the person who sent the largest number of emails to himself, which it could answer if it had this method, running along the **diagonal** of the array (where indexes are equal). Note that there is no need to have a separate variable `max` that keeps track of the largest value seen so far (on the diagonal). All you need to track is the row and column number `code` of the largest item.

```
public int codeOfMaxSelfSent()  // in EmailSet
{  int code = 0;
   for (int k = 1;  k < itsSize;  k++)
      if (itsItem[code][code] < itsItem[k][k])
         code = k;
   return code;
}  //======================
```

Note that all four of the preceding methods in this section return zero if `itsSize` is zero, since int array components are always initialized to zero by the runtime system.

You could ask an EmailSet object whether any person sent more than ten emails to any one person, which it could answer if it had this method, called with the parameter 10 and running through all possible combinations of the values of both indexes:

```
public boolean anyMoreThan (int cutoff)  // in EmailSet
{  for (int from = 0;  from < itsSize;  from++)
      for (int to = 0;  to < itsSize;  to++)
         if (itsItem[from][to] > cutoff)
            return true;
   return false;
}  //=====================
```

Exercise 12.21 Declare and construct a 3-dimensional array representing the names of the three best friends of the 25 different students in five different classrooms. What expression represents the name of the second friend listed of student #12 in classroom #2, numbering students and classrooms from 0 up?
Exercise 12.22 Declare and construct a 2-dimensional array representing the ages of each of two children of each of two people. Then write an expression for the average age of all four children.
Exercise 12.23 Revise the `anyMoreThan` method to ignore emails one sends to oneself.
Exercise 12.24 Write an EmailSet method `public int numPeopleSentBy (int code)`: The executor tells the number of people to whom one or more emails were sent by a person with a given code.
Exercise 12.25 Write an EmailSet method `public int maxSentTo (int code)`: The executor tells the largest number of emails sent to a person with a given code.
Exercise 12.26 Write an EmailSet method `public boolean allLessThan (int cutoff)`: The executor tells whether everyone sent less than `cutoff` emails to every other person. Call the `anyMoreThan` method to do most of the work.
Exercise 12.27* Write an EmailSet method `public boolean complex (int code, int cutoff, int num)`: The executor tells whether a given person sent more than `cutoff` emails to at least `num` different people.
Exercise 12.28** Write an EmailSet method `public int howMany (int cutoff, int num)`: The executor tells the number of people who sent more than `cutoff` emails to at least `num` different people.
Exercise 12.29** Write an EmailSet method `public boolean pariah (int cutoff)`: The executor tells whether there is any person who was sent less than `cutoff` emails by all other persons. That is, tell whether there is any column of the rectangular array where all non-diagonal components are less than 5.

12.6 Implementing The Email Software With A Two-Dimensional Array

The discussion so far leads to the implementation of the Email software shown in Listing 12.5 (see next page). Each line of the main logic design translates to just a few Java statements except for printing the statistics. So a `printStatistics` method goes in a helper class EmailOp to carry out that task separately.

The program can be run by entering e.g. the following

```
java EmailStats email09Sep02.dat results.dat
```

as the command line in the terminal window. That passes the two file names to the program in the command line arguments `args[0]` and `args[1]`. Since it is easy for a user to occasionally forget to enter one or both file names, the program begins by verifying that it has at least two String values for the file names. If it does not, it prints an appropriate message and terminates the program.

Listing 12.5 Application program for the Email software

```
import java.io.*;

public class EmailStats
{
   /** Read one day's email file and print several usage
    *  statistics about it. */

   public static void main (String[] args) throws IOException
   {
      // CONSTRUCT THE EMAIL FILE AND THE DATABASE
      if (args.length < 2)
      {  System.out.println ("Specify the two file names");
         System.exit(0);
      }
      BufferedReader inputFile = new BufferedReader
                                 (new FileReader (args[0]));
      EmailSet database = new EmailSet();

      // READ THE EMAIL FILE AND STORE IT IN THE DATABASE
      Message data = new Message (inputFile);
      while (data.getSender() != null)
      {  database.add (data);
         data = new Message (inputFile);
      }

      // PRINT THE RESULTS TO THE OUTPUT FILE
      if (database.getNumEmployees() > 0)
         EmailOp.printStatistics (database, new PrintWriter
                                  (new FileWriter (args[1])));
   }  //=====================
}
```

Reading the Messages uses the standard sentinel-controlled loop: You must check each Message read to see if it signals the end-of-file condition was present when it tried to get the data. As long as it was not, add it to the database. When it signals end-of-file, the program can stop reading values and start printing results.

The `printStatistics` method call has two parameters, the database and the output file, because the database provides the results and the `out` file accepts the output. The next thing to be done is to design the logic for the `printStatistics` method in the EmailOp class. In the process, you will see what additional methods the EmailSet class needs.

The printStatistics method

Statistics to be printed include the number of emails sent, the number of people who sent them, and the average sent per person. The earlier Listing 12.4 has EmailSet methods that provide the information needed to calculate these statistics.

You also need to go through the employees one at a time (code numbers going from 0 up to the number of employees) and print out for each one whether he did not send any email. If he sent email, you need to print out whether he received at least twice what he sent, or he sent at least twice what he received, and whether he ever sent email to himself.

To obtain this information, you need to be able to ask an EmailSet for the name of the employee with a given code number and for the number of emails sent to or from a given code number. Call the former method `findEmployee`. You should calculate the ratio `db.numSentTo(k)/db.numSentFrom(k)` and then print `db.findEmployee(k)` if that ratio is at least 2.0 or at most 0.5. Listing 12.6 contains an implementation of this logic for the `printStatistics` method.

Listing 12.6 The EmailOp helper class

```
class EmailOp   // helper class for Email
{
   /** Print the statistics about email usage. */

   public static void printStatistics (EmailSet db,
                                       PrintWriter out)
   {  // PRINT OVERALL AVERAGE
      int size = db.getNumEmployees();  // for speed of execution
      out.println (db.getNumEmails() + " email sent by "
               + size + " people; average sent per person = "
               + 1.0 * db.getNumEmails() / size);

      // PRINT THOSE WITH HEAVY IMBALANCE IN SENT VS RECEIVED
      for (int k = 0;  k < size;  k++)
      {  String name = db.findEmployee (k);
         if (db.numSentFrom (k) == 0)
            out.println (name + " did not send any email.");
         else
         {  double ratio = 1.0 * db.numSentTo (k)
                               / db.numSentFrom (k);
            if (ratio >= 2.0)
               out.println (name + " received twice the sent.");
            else if (ratio <= 0.5)
               out.println (name + " sent twice the received.");
            if (db.numSent (k, k) > 0)
               out.println (name + " sent himself email.");
         }
      }

      // PRINT THOSE WHO SENT THE LEAST AMOUNT OF EMAIL
      int smallest = db.leastSent();
      for (int k = 0;  k < size;  k++)
         if (db.numSentFrom (k) == smallest)
            out.println (db.findEmployee (k) + " sent least.");
      out.close();
   }  //====================
}
```

Programming Style: The `printStatistics` method contains three cases in which a value is obtained from the database by a method call and stored in a simple variable to make the execution faster or the logic clearer. Two cases are `size = db.getNumEmployees()` and `smallest = db.leastSent()`, to avoid making the method calls many times during the for-loops. The general principle is that, when you retrieve the same value three or more times, you should usually retrieve it once and remember it. The other case is `name = db.findEmployee(k)`. It does not save significant time, but it seems clearer to use a simple name in four places rather than the method call.

The EmailSet class

The most important addition to make to the EmailSet class is a way of shifting back and forth between the name (email address) of an employee and the code number used to store information about him in the database. When you add a given Message to the data base, you have to find the code that corresponds to `given.getSender()` and `given.getReceiver()`. And when `printStatistics` has information about a code number, it wants `findEmployee(k)`, the name corresponding to `k`, for its output.

An elementary way of doing this is to have each EmailSet object maintain another array of names entered so far. Call it `itsName`. The first name entered goes in `itsName[0]`, and 0 is its code. The second name entered goes in `itsName[1]`, and 1 is its code, and so forth. You can keep track of the total number of different names entered so far in a variable named `itsSize` (P.S. You know, don't you, that you can choose any name you like, such as `ralph`? It is just that `itsSize` is so much more informative).

When you get a Message object, you need to search for each of the sender and the receiver of the message. So you call a private method `findCode(employee)` to ask the EmailSet object to search through the `itsName` array to see if the employee (a String value) is already in it. If so, it returns the index where it found that employee. If not, it adds the employee at the end of the list in `itsName[itsSize]` and returns that index, after incrementing `itsSize` by 1.

The public `findEmployee` method that the application uses need only return `itsName[code]`. The `leastSent` method has to find the code of the person who sent the least amount of email. It can go through each of the code values to find the value of `numSentFrom(k)` that is smallest. This uses the standard logic for finding the minimum value in an array.

The finished EmailSet class is in Listing 12.7 (see next page), except for the parts already presented in the earlier Listing 12.4 and the `numSentTo` method which is in the next section. Two defects fixed in the exercises are (a) the `leastSent` method is unnecessarily slow in execution and (b) the program crashes if you have more than MAX employees.

Storing more information to reduce calculations

A different approach to the entire development of EmailSet is to keep track of sufficient information at all times so that answering most questions about the number of emails sent, the maximum sent, etc., can be answered without calculations. To do this, you have to add several more one-dimensional arrays and update their values each time an email is added. Then the body of the `numSentFrom` method would be no more than `return numSentFrom[code];`. For this approach, the `add` method could begin with the following statements:

```
int from = findCode (given.itsSender());
int to = findCode (given.itsReceiver());
itsNumEmails++;
itsItem[from][to]++;
numSentFrom[from]++;
numSentTo[to]++;
if (itsItem[from][to] > maxSentFrom[from])
   maxSentFrom[from]++;
```

Exercise 12.30 How would you rewrite Listing 12.5 and Listing 12.6 so that the PrintWriter is created within the `printStatistics` method rather than on the way to it?
Exercise 12.31 Rewrite the `leastSent` method in Listing 12.7 to avoid two calls to the same method, since that method inefficiently executes a for-statement on each call.

Listing 12.7 The rest of the EmailSet class except numSentTo

```
// public class EmailSet continued

   private int[][] itsItem = new int[MAX][MAX]; // initially zero
   private String[] itsName = new String[MAX];
   private int itsSize = 0;
   private int itsNumEmails = 0;

   public void add (Message given)
   {  int from = findCode (given.getSender());
      int to = findCode (given.getReceiver());
      itsItem[from][to]++;
      itsNumEmails++;
   }  //=====================

   /** Precondition:  0 <= code < getNumEmployees() */

   public String findEmployee (int code)
   {  return itsName[code];
   }  //=====================

   private int findCode (String employee)
   {  for (int k = 0;  k < itsSize;  k++)
         if (employee.equals (itsName[k]))
            return k;
      itsName[itsSize] = employee;
      itsSize++;
      return itsSize - 1;
   }  //=====================

   public int numSentFrom (int code)
   {  int count = 0;
      for (int k = 0;  k < itsSize;  k++)
         count += itsItem[code][k];
      return count;
   }  //=====================

   public int leastSent()
   {  int min = numSentFrom (0);
      for (int k = 1;  k < itsSize;  k++)
         if (min > numSentFrom (k))
            min = numSentFrom (k);
      return min;
   }  //=====================
```

Exercise 12.32* The variable `name` is mentioned four times in the `printStatistics` method after it is declared. How many times is it actually evaluated on each iteration?
Exercise 12.33* Perhaps the employer would at times like a shortened version of the statistics, giving only the total number sent and the average sent per person. This is to be indicated by a third command-line argument. What would you revise in Listing 12.5 and Listing 12.6 to terminate the `printStatistics` method after the one line is printed whenever the command line contains at least one more argument?
Exercise 12.34** What is the most efficient way of keeping the `minSentFrom` array up-to-date in the alternative Email logic described at the end of this section? Are two extra arrays likely to be more efficient than just calculating the value whenever it is requested?
Exercise 12.35** Revise the Email software to terminate gracefully when there are more than MAX employees. As it is now, it crashes with an ArrayIndexOutOfBoundsException in the `findCode` method.

12.7 Using A Two-Dimensional Array Of Airline Data

Suppose you are working on software to handle a database of airline flights between various cities. Each city has a code number from 0 up through `MAX-1`, where `MAX` is a positive integer. An appropriate data structure for this information is a two-dimensional array of Flight objects stored in RAM:

```
private Flight [][] plane = new Flight [MAX][MAX];
```

So `plane[dep][arr]` is one component of this array, a description of the airplane flight departing from the city with code `dep` and arriving in the city with code `arr`. If that particular city-to-city combination does not have a direct flight, then `null` would be stored in the array at that component.

The Flight class could have the following instance variables. Note that all are `final`, which means that, once the constructor assigns a value, it cannot be changed. The objects are immutable. So it does not violate encapsulation principles (and it is convenient) to make the instance variables all public:

```
// public class Flight:  five instance variables
   public final String aircraftType;
   public final double ticketPrice;
   public final int    numSeats; // available unreserved
   public final int    numReservations;
   public final long   filePosition;
```

These instance variables are self-explanatory except `filePosition`, which is explained shortly. This Flight class has one constructor and no other methods. The constructor simply initializes all fields:

```
public Flight (String type, double price, int seats,
               int reservations, long position)
{  aircraftType = type;
   ticketPrice = price;
   numSeats = seats;
   numReservations = reservations;
   filePosition = position;
}  //=====================
```

Suppose you have a FlightSet class with the `plane` array as one instance variable, a 2-dimensional array of Flight objects. The logic for a method to find the cheapest ticket price from any city to any other city could then be as follows, relying on the quite reasonable assumption that there must be at least one flight with a price under a million dollars: (1) Set `smallestSoFar` to be a million dollars. (2) Inspect each Flight object in the database and, wherever you see one that is not `null` and has a smaller ticket price than the value stored in `smallestSoFar`, replace that value. (3) return the value of `smallestSoFar`. Listing 12.8 contains this logic.

This technique of initializing `smallestSoFar` to a ridiculously high value, to assure it is replaced by the first real piece of data, is not as efficient and not as robust as the usual technique of initializing it to the first value in the data set. But the latter technique does not work as well when several values can be null, as in this example.

Listing 12.8 The FlightSet class, but with only one of its methods

```
public class FlightSet extends Object
{
   public static final int MAX = 20;
   ////////////////////////////////
   private Flight[][] plane = new Flight[MAX][MAX];

   /** Return the fare for the flight with the lowest fare.
    *  Precondition:  There is at least one non-null entry. */

   public double cheapestFare()
   {  double smallestSoFar = 1000000.0;     // a million dollars
      for (int dep = 0;  dep < MAX;  dep++)
         for (int arr = 0;  arr < MAX;  arr++)
            if (plane[dep][arr] != null
                  && plane[dep][arr].ticketPrice < smallestSoFar)
            {  smallestSoFar = plane[dep][arr].ticketPrice;
            }
      return smallestSoFar;
   }  //======================
}
```

Exercise 12.36 Write a FlightSet method `public int howManyArriveAt (int city)`: The executor tells how many flights arrive at a given city.
Exercise 12.37 Write a FlightSet method `public int numSeatsAvailableFrom (int city)`: The executor tells the total number of seats available (unreserved) out of a given city to all the rest of the cities.
Exercise 12.38 Write a FlightSet method `public int flightsOverHalfFull()`: The executor tells how many flights are more than half full.
Exercise 12.39 Write a FlightSet method `public int roundTrips()`: The executor tells how many cases there are of two different cities with direct flights both ways.
Exercise 12.40* Write a FlightSet method `public boolean fairlyDirect (int fromCity, int toCity)`: The executor tells whether you can get from one given city to another given city either directly or with one stopover.
Exercise 12.41** Write a FlightSet method `public void costlyDirect()`: The executor prints out all cases in which it is more expensive to fly directly from a city to another city than it is to go by way of some other city as the one stopover.
Exercise 12.42** Write a FlightSet method `public int planeTypes()`: The executor tells the number of different airplane types, computed over all flights. Hint: Study what `findCode` does in Listing 12.7.

12.8 The RandomAccessFile Class

The airline data described in the previous section is kept in an array in RAM because it is far cheaper in terms of execution time to answer requests for information using the array rather than reading it from a data file. But you have to worry that someone could kick the plug or spill a drink on the computer, thereby losing all your data. So each time a change is made in the data stored in RAM (in the `plane` array), this data is updated in a binary file on the hard disk.

A **binary file** is one that stores values in the form that they have in RAM, not in the form that they appear on the screen. For instance, the int values 3 and 47312 are one character and five characters, respectively, when written on the screen or stored in a String value. But in RAM, every int value takes up exactly four bytes of space. Writing information in binary form executes much faster than writing it in textual form.

The RandomAccessFile class

The FlightSet class could define its binary file instance variable as follows:

```
java.io.RandomAccessFile raf = new java.io.RandomAccessFile
                      ("flight.data", "rw");
```

Random-access means that you can read from or write to any point in the file without first going through all the values at the beginning of the file, as you have to do with a sequential file. `"rw"` signals that you can both read and write with this file; the other choice for the second parameter is `"r"` for a read-only file. You can write all the basic kinds of values to the file if you use the appropriate method, e.g.:

```
raf.writeDouble(-3.72)      // write 8 bytes
raf.writeInt(47)            // write 4 bytes
raf.writeChar('x')          // write 2 bytes, Unicode
raf.writeChars("test case") // write that many 2-byte chars
raf.writeBoolean (done)     // write 1 byte (1==true)
```

You also have the corresponding methods for reading values, except that the String input is somewhat different: The `readLine` method only handles the basic Unicode values 0 to 255 well, stops reading at a newline or end-of-file, and does not include the newline character in the String value returned. The following all return the value of the type specified:

```
raf.readDouble()    // read 8 bytes
raf.readInt()       // read 4 bytes
raf.readChar()      // read 2 bytes, Unicode
raf.readLine()      // read to newline or end-of-file
raf.readBoolean()   // read 1 byte; 0 is false, others true
```

Of course, you have the corresponding input and output methods for bytes, shorts, floats, and longs. The only other RandomAccessFile methods you need for now are as follows:

```
raf.close()  // disconnect when done with the file
raf.length() // return the long number of bytes in the file
raf.seek(n)  // set the point where reading or writing next
             // takes place.  n is a long value.
```

`raf.seek(n)` sets the read/write point, called the **file-pointer position**, at the byte of that number, numbering from 0 on up as always. So `raf.seek(1000);` `raf.writeInt(47)` writes four byte values at positions 1000, 1001, 1002, and 1003.

All the RandomAccessFile methods throw an IOException or a subclass of IOException. Figure 12.6 illustrates the use of these methods.

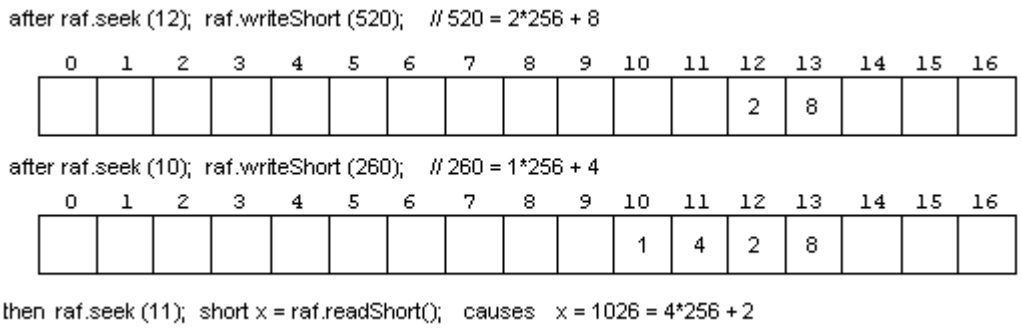


Figure 12.6 illustration of reading and writing with a random-access file

Now the purpose of the `filePosition` field in a Flight record should be clear. When you modify the data stored in a particular component of the `plane` array (replacing the existing Flight record by a new one), you backup the RAM data as follows:

```
public void writeRecord (Flight flit) throws IOException
{  raf.seek (flit.filePosition);
   raf.writeChars (flit.aircraftType + "\n");
   raf.writeDouble (flit.ticketPrice);
   raf.writeInt (flit.numSeats);
   raf.writeInt (flit.numReservations);
}  //====================
```

Each time you write to a random-access file, the file-pointer position moves forward by the number of bytes written. That is why the `writeRecord` method works without having the int values overwrite the chars and double. The same forward motion happens with reading from a random-access file.

Writing beyond the end of the file is allowed; it extends the file to that point. Reading beyond the end of the file throws an IOException of some kind.

Categories of standard file classes

The Sun standard library has six basic groups of classes of file objects, all in `java.io`:

1. Reader is an abstract class whose many subclasses read into arrays of chars.
2. Writer is an abstract class whose many subclasses write from arrays of chars.
3. InputStream is an abstract class whose many subclasses read into arrays of bytes.
4. OutputStream is an abstract class whose many subclasses write from arrays of bytes.
5. StreamTokenizer is a concrete class for treating a text file as a Java program.
6. RandomAccessFile is a concrete class for random access of a text file.

Exercise 12.43 Write a FlightSet method `public void readRecord (Flight flit, long filePosition)`: The executor reads a record from the random access file `raf` at a particular file position into a particular Flight variable.
Exercise 12.44* Write an independent method `public static swapTwo (long filePosition)`: It reads two consecutive double values from the random-access file `raf` at a given file position and swaps the two values in the file.

12.9 How Buffering Is Done

Let us pretend for a bit that you do not have the BufferedReader class, only the InputStreamReader class and its subclass FileReader. The purpose of this exercise is twofold: (a) You will understand the separate roles of the BufferedReader class and the other two classes, and (b) you will develop more competence with arrays, particularly char arrays.

A structure that occurs frequently in the Sun standard library is an array of characters. This can be used to store one character in each component. A text file, for instance, is just a long sequence of characters. If it is not overly large, you can read it into a single char array and process the information in it.

FileReader is a subclass of InputStreamReader that adds only some constructors. This is to make opening a file a one-step process rather than a two- or three-step process. The main non-constructor methods you have in InputStreamReader (and thus in FileReader by inheritance) are as follows. All three throw an IOException:

- `int read()` obtains a single character from the file. It returns -1 at end-of-file.
- `int read(char[] cbuf, int off, int len)` attempts to read `len` characters into the char array `cbuf` starting at component `off`. It returns the number of characters read, except it returns -1 if it previously reached end-of-file.
- `void close()` disconnects from the file.

You can create your own class to read values from the file 8192 characters at a time and respond to `readLine` requests. This Buffy class, shown in Listing 12.9 (see next page), has five instance variables. The `itsBuf` array is to hold 8192 characters you read from the `itsInput` file, or less if you have reached the end of the file. The `itsSize` variable keeps track of the number of useable characters in the array; this will be 8192 unless you have reached the end of the file. The `itsNextChar` variable keeps track of the index of the next available character in the array. So the characters indexed `itsNextChar... itsSize-1` have been read from the file but not "consumed" by `readLine` yet.

The logic for `readLine` is rather complex, so you need to design it carefully. The accompanying design block gives a reasonable plan.

STRUCTURED NATURAL LANGUAGE DESIGN for the readLine method
If the number of "unconsumed" characters in the buffer gets below 2000 and
there are more characters in the file then...
    Move the ones you have to the front of the array.
    Go get some more characters from the file.
If you are still out of characters then...
    Return `null` as the answer to the readLine query.
Record in `start` the current position in the buffering array.
Count the number of characters down to the next newline character `'\n'`.
Return a String consisting of that many characters starting from `start`.

You also need a private method to fill in the buffer starting from some particular index; you could call it using `fillBufferStartingAt(front)`. It uses the `read` method from InputStreamReader to get as much as possible into the array. A call of the `read` method must supply the char array to be filled in, the first index where the filling is to begin, and the number of characters to fill in. The array will be filled completely (`itsSize == MAX`) unless you get to the end of the file first.

The logic uses the String constructor that accepts a slice of a char array: `new String (itsBuf, start, count)`. This logic is simplified by the assumption that no line has more than 2000 characters. It is an exercise to remove this assumption.

Note that a file with e.g. 78,000 characters will have only ten read operations, which saves on execution time. That is basically what BufferedReader does, except with somewhat more care and slightly more speed.

Exercise 12.45* It is actually possible for the last line in the file to not have a newline character at the end of it. Revise the condition of the while-statement in Listing 12.9 to prevent a wrong result in such a case.
Exercise 12.46** Revise the `readLine` method in Listing 12.9 so it works no matter how many characters a line has. However, insert a newline character into any line with more than 8192 characters, to break it up.

Listing 12.9 The Buffy class

```
import java.io.*;

public class Buffy extends Object
{
   public static final int MAX = 8192;
   ////////////////////////////////
   private Reader itsInput;        // the file or keyboard input
   private char[] itsBuf;          // chars read but not used
   private int itsSize;            // number of chars stored
   private int itsNextChar;        // position of next char
   private boolean itsAtEndOfFile; // true when no more to read


   public Buffy (Reader given) throws IOException
   {  itsInput = given;
      itsBuf = new char[MAX];
      fillBufferStartingAt (0); // get the first characters
   }  //=====================

   private void fillBufferStartingAt (int loc) throws IOException
   {  itsSize = loc + itsInput.read (itsBuf, loc, MAX - loc);
      itsAtEndOfFile = itsSize < MAX;
      if (itsAtEndOfFile)
         itsInput.close();
      itsNextChar = loc;
   }  //=====================

   public String readLine() throws IOException
   {  if (itsNextChar >= itsSize - 2000 && ! itsAtEndOfFile)
         downShiftAndFill();
      if (itsNextChar >= itsSize)         // no more characters
         return null;
      int start = itsNextChar;
      int count = 0;
      while (itsBuf[start + count] != '\n')
         count++;
      itsNextChar = start + count + 1;   // prepare for next call
      return new String (itsBuf, start, count);
   }  //=====================

   private void downShiftAndFill() throws IOException
   {  for (int k = itsNextChar;  k < itsSize;  k++)
         itsBuf[k - itsNextChar] = itsBuf[k];
      itsSize -= itsNextChar;
      itsNextChar = 0;
      fillBufferStartingAt (itsSize);
   }  //=====================
}
```

*12.10 Additional Java Language Features (*Enrichment)*

The following are features of the Java language that are not used elsewhere in this book, but you may find them useful in your software development.

Visibility modifiers

You have only seen the two visibility modifiers public and private. Java has four levels of visibility: public, private, protected, and default. A member of a class can be marked as public, private, or protected; default visibility applies when you do not use any of them.

- **public visibility** of a member of class X: Any statement anywhere can mention the member.
- **protected visibility** of a member of class X: Only statements in the same package with X or in a subclass of X can mention the member.
- **default visibility** of a member of class X: Only statements in the same package with X can mention the member.
- **private visibility** of a member of class X: Only statements that are within X can mention the member.

Bitwise operators

Each byte value is a sequence of 8 bits and each int value is a sequence of 32 bits. Java provides several operators to work on the individual bits of byte and int values, as well as the bits of char, short, and long values. One is the **bitwise-and**, for which the symbol is `&`. This operator produces the value that has a 1-bit in each position where both of the two operands have a 1-bit, and a 0-bit in every other position. For instance, `13 & 10` is 8, because the binary notation is 00001101 for 13 and 00001010 for 10, and thus the result is 00001000 for 8.

The **bitwise-or** operator is `|`, which produces the value that has a 1-bit in each position where either one of the two operands has a 1-bit, and a 0-bit in every other position. For instance, `13 | 10` is 15, because the result is 00001111 for 15.

The **bitwise-negation** operator is `~`, which produces the value that has a 1-bit wherever the single operand has a 0-bit, and a 0-bit wherever the single operand has a 1-bit. For instance, `~13` is 242, because the result of "inverting" 00001101 for 13 is 11110010 for 242. This applies if 13 is a byte value. In general, the bitwise-negation of a byte value is 255 minus that byte value, and the bitwise-negation of a short value (16 bits; note that 2^{16} `- 1` is 65535) is 65535 minus that short value.

The **bitwise-exclusive-or** operator is `^`, which produces the value that has a 1-bit in each position where exactly one of the two operands has a 1-bit, and a 0-bit in every other position. For instance, `13 ^ 10` is 00001101 ^ 00001010 in binary, so the answer is 00000111 in binary, which is 7. In summary: Since 13 is 8+4+1 and 10 is 8+2, `13 & 10` is 8, `13 | 10` is 8+4+2+1, and `13 ^ 10` is 4+2+1.

The **left-shift** operator is <<; `x << n` shifts all the bits of x to the left by n places, filling in zeros in the places left empty. So `13 << 1` is 26 (00011010) and `13 << 2` is 52 (00110100). In general, a left-shift by n places multiplies the number by 2^n. The **right-shift** operator works as follows: `x >>> n` shifts all the bits of x to the right by n places, filling in zeros in the places left empty. 26 is 00011010 in binary, so `26 >>> 1` is 13 (00001101) and `26 >>> 2` is 6 (00000110, since the last bit "drops off the end"). In general, a right-shift by n places divides by 2^n, dropping the remainder.

Other stuff you can do in Java

- You may declare a parameter of a method as `final`, which means that the coding in the method cannot change its value.
- You may have one method whose heading is simply `static`. This initializer class method will be executed before any user of the class calls one of its methods.
- You may put two or more separate classes in one file, but at most one can be public. If one is public, the name of the file must match the name of the public class.

Exercise 12.47* Calculate `22 & 12`, `22 | 12`, and `22 ^ 12`.
Exercise 12.48* Calculate `47 << 1`, `47 << 2`, `47 >>> 1` and `47 >>> 2`.

*12.11 Java Bytecode Commands (*Enrichment)*

The `.class` file contains primarily **bytecode**. This is a sequence of byte values to be executed by the runtime system, carrying out the instructions in the various methods of the class. The bytecode sequence for each method consists of a number of commands. Each **command** is a single byte, called the **opcode**, followed by a number of bytes, called the **operands** of the opcode. The **Java Virtual Machine** (**JVM**) executes these commands one at a time.

An **opcode** can be any of the numbers 0 through 255 (since a byte is eight bits, and 2^8 is 256). For instance, the command to add two double values is 99 and the command to subtract one int value from another is 100. Since the numbers are difficult to remember, we use a standard set of verbal clues in their place. The **mnemonic code** for 99 is `dadd` and the mnemonic code for 100 is `isub`. In general, mnemonic codes starting with `i` operate on int values and those starting with `d` operate on double values.

The number of bytes in the command depends on the particular opcode. Some opcodes require no operands at all, others require one or two, each of which might be 1, 2, 4, or 8 bytes long.

The operand stack

Each method has an **operand stack** for doing computations. When a method is called, it becomes the **current method** and its operand stack is created. When the current method finishes executing, the method that called it becomes the current method and takes up where it left off, with its own operand stack.

Some commands add values on the top of the stack, others take values off the top of the stack. The stack is a LIFO structure -- last-in-first-out. So if you add two values to the stack and then remove one, you will get the second one that you added.

The JVM typically numbers the parameters of a method in order starting from 0, then applies the next few numbers to the variables declared inside the method. As an example, consider the following method:

```
public static int diff (int x, int y)  // independent
{  int answer = x - y;
   return 2 * answer;
}  //=====================
```

The body of this method is translated into bytecode as a sequence of only 12 bytes, as you will see shortly. The three variables are given the index numbers 0 for `x`, 1 for `y`, and 2 for `answer`. Then the first statement is translated to bytecode as follows:

- `iload 0` (numerically, 21 0) is the command to load the value of local #0, which is x, onto the operand stack.
- `iload 1` (numerically, 21 1) is the command to load the value of local #1, which is y, onto the operand stack. So far, the stack contains the value of y on top of the value of x.
- `isub` (numerically, 100) is the command to pop off the top two values of the operand stack and push back onto the stack the result of subtracting the top value from the second-from-top value. So now the operand stack contains only one value, `x - y`.
- `istore 2` (numerically, 54 2) is the command to pop off the top value from the operand stack and store it in local #2, which is `answer`. Now the stack contains no values at all, but the variable named `answer` contains the value of `x - y`.

What actually appears in the bytecode to implement the first statement is therefore this sequence of seven bytes: `21 0 21 1 100 54 2`.

Standard bytecode descriptions

A compact but fairly clear description of the bytecode commands mentioned above is as follows. In these descriptions, the phrase "causes..." shows how the stack changes, e.g., for `iload`, an int value is added to the top of the stack without changing what is already there. The part before the ==> is what the stack looks like before the command is executed, the part after the ==> is what the stack looks like after the command is executed, and the ellipsis `...` indicates the part of the stack that remains unchanged.

`iload` `(21) byteVal` causes ... ==> ... intVal
// The current value stored in local variable #byteVal is pushed onto the stack.
`istore` `(54) byteVal` causes ... intVal ==> ...
// The int value on top of the stack is popped and stored into local variable #byteVal.
`isub` `(100)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 - intVal2 is pushed.

Some additional bytecode commands needed for the `diff` method are as follows:

`iconst_2` `(5)` causes ... ==> ... intVal
// The constant int value of 2 is pushed onto the stack.
`imul` `(104)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 * intVal2 is pushed.
`ireturn` `(172)` causes ... intVal ==>
// The int value on top of the stack is popped and returned to the calling method.
// The operand stack of the current method is discarded, and the returned value is
// pushed onto the operand stack of the calling method, which is now the
// current method.

Now the second statement in the `diff` method can be translated as

```
iconst_2;  iload 2;  imul;  ireturn;
```

which numerically is this sequence of five bytes: `5 21 2 104 172`. Other byte codes whose meaning should now be clear are as follows:

`iadd` `(96)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 + intVal2 is pushed.
`idiv` `(108)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 / intVal2 is pushed.
`irem` `(112)` causes ... intVal1 intVal2 ==> ... intVal3
// The two int values on top of the stack are popped, and intVal1 % intVal2 is pushed.

iconst_1 (4) causes ... ==> ... intVal
// The constant int value of 1 is pushed onto the stack.
iconst_0 (3) causes ... ==> ... intVal
// The constant int value of 0 is pushed onto the stack.
iconst_m1 (2) causes ... ==> ... intVal
// The constant int value of -1 is pushed onto the stack.

Branching instructions

You know that the if-statement and while-statement and others can cause execution to jump from one instruction to another, skipping those in between. This is implemented in bytecode by a **branching instruction**. The main one is `goto shortVal`, where shortVal is a two-byte whole number in the range from -32768 to +32767. The effect of `goto +20` is that the next command executed is the one that is 20 bytes forward of the goto byte in the bytecode sequence. The effect of `goto -52` is that the next command executed is the one that is 52 bytes before the goto byte in the bytecode sequence. In both cases, the new position must be the first byte in a command within the same method.

You also have six bytecode commands for a **condition branch**. The mnemonic codes for these six are all the same except for the last two characters, which tell what kind of comparison is to be made. For instance `ifeq +80` means to pop off the top int value on the operand stack and, if it equals zero, branch forward 80 bytes from the `ifeq` byte; and `iflt -40` means to pop off the top int value on the operand stack and, if it is less than zero, branch backward 40 bytes from the `iflt` byte. The standard descriptions of these bytes codes are as follows:

goto (167) shortVal causes ... ==> ...
// The stack is not changed. Branch shortVal bytes away, forward or backward.
ifeq (153) shortVal causes ... intVal ==> ...
// The int value is popped, and if intVal == 0 then branch shortVal bytes away.
iflt (155) shortVal causes ... intVal ==> ...
// The int value is popped, and if intVal < 0 then branch shortVal bytes away.

The bytecode also has `ifne` (154), `ifge` (156), `ifgt` (157), and `ifle` (158). Now consider the following method to find the smaller of two int values:

```
public static int smaller (int x, int y)  // independent
{  int answer;
   if (x >= y)
      answer = y;
   else
      answer = x;
   return answer;
}  //=====================
```

Remember than x is local variable #0, y is local variable #1, and answer is local variable #2. The mnemonic codes for this method would be as follows:

```
iload 0;  iload 1;  isub;  ifge + 10;
iload 1;  istore 2;
goto + 7;
iload 0;  istore 2;
iload 2;  ireturn;
```

You are probably wondering how the +10 and +7 could be calculated before the rest of the bytecode commands were written. The answer is, they could not be. You leave the **offset** amount (+10 or +7 in this case) blank until you have written enough more of the bytecode that you can count the bytes to see how much it should be. Remember when you count that each offset amount takes up two bytes.

This section has presented only 20 of the 255 possible bytecodes. Some perform operations with float, byte, short, and long values; and some perform operations with object and array references. You may look at `http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html`, particularly the section titled "The Java Virtual Machine Instruction Set", to see the other possibilities.

Assembly language

An **assembly language** is basically machine code with mnemonics for operands and other things, as well as opcodes. For instance, an assembly language for Java bytecode could specify these memory aids:

`int x, y, answer` means that `x` represents local #0, `y` represents local #1, `answer` represents local #2, and similarly wherever a byteValue operand is called for. `double u, v, result` means that `u` represents local #0, `v` represents local #2, `result` represents local #4, and similarly wherever a byteValue operand is called for (since each double value requires the space of two int values).

The counter is assumed to continue with further such "declarations"; so if both of those declarations appeared in a single method in that order, then `u` would represent local #3, `v` would represent local #5, and `result` would represent local #7.

It is then not too hard to develop an **assembler** program to translate a source file containing such mnemonics to the actual machine code. You would also save yourself the trouble of counting bytes (and re-counting if you make modifications) by specifying that an "instruction" which is some name followed by a colon denotes the location of the immediately following byte; that name could be called a **target**. The assembly program could allow a target to be mentioned directly after any of the seven branching instructions in place of the byteVal offset; the assembly program will compute the offset for you. Then the `smaller` method using double values could be expressed in assembly language as follows:

```
            double x, y, answer;
            dload x;  dload y;  dcmpg;  ifge xLarger;
            dload y;  dstore answer;
            goto endif;
   xLarger: dload x;  dstore answer;
   endif:   dload answer;  dreturn;
```

Surely you can see how it would be much easier to write a bytecode program in assembly language and have the assembler make the conversion for you. Programmers almost uniformly wrote all programs in an assembly language up until around 1955, when FORTRAN was invented. Clearly they were hardy souls. Of course, it is even easier to write in Java itself and let the Java compiler make the translation for you.

Exercise 12.49 The assembler language for the smaller method could have one or more instructions omitted and still have the same effect. What would the change be?
Exercise 12.50* Write the `diff` method in the assembly language described here.
Exercise 12.51* Write an assembly language method to calculate the greatest common divisor of two positive integer parameters. Hint: Repeatedly subtract the smaller from the larger until they become equal.

*12.12 About Networking Using Sockets (*Sun Library)*

The **java.net.ServerSocket** class has the following useful methods:

- `new ServerSocket(portInt, queueInt)` creates a new socket that can be used as a server for various clients. The `portInt` is the **port number** by which the ServerSocket is identified to its clients; the `portInt` can be any int value from 1024 to 65535. The `queueInt` is the number of clients requesting a connection that it can hold in its backlog queue for later processing while it is already busy with a client. The `queueInt` can be any non-negative int value.
- `someServerSocket.accept()` waits until some client socket asks for a connection (i.e., it **blocks** until asked). At that point the method returns the Socket object that requested the connection.
- `someServerSocket.close()` terminates the existence of that ServerSocket.

The **java.net.Socket** class has the following useful methods:

- `new Socket(hostString, portInt)` creates a new socket that is connected to port number `portInt` on the remote host whose internet name is `hostString`.
- `someSocket.getInputStream()` returns an InputStream object for reading bytes from this client.
- `someSocket.getOutputStream()` returns an OutputStream object for writing bytes to this client.
- `someSocket.close()` terminates the existence of that ServerSocket.

All of these methods can throw a `java.io.IOException`. The constructors and the `accept` method can also throw a `java.lang.SecurityException` (which is a RuntimeException).

Example of using Sockets

The best way to understand how all of these methods interact is to see an example. My friend would like all her relatives to be able to suggest names for the baby she is expecting. So she has an account on which she runs the Telnetting application program given in Listing 12.10 (see next page). It starts by creating a Handler object. Then each time some relative telnets to that account, the Handler's `getMore` method executes.

Note that the `getMore` method is completely independent of sockets. It is simply given an input and an output by which it communicates with the relatives. It prints a list of all names suggested so far for the baby, then gets any additional suggestions the relative has and adds them to the list. When my friend wants to know the list so far, she telnets in to the account to see what it displays.

The main method creates a new Handler for the baby names and a new ServerSocket with an arbitrarily-chosen port number of 12345. It allows up to twenty relatives to be queued up waiting to make suggestions (it is a very large extended family). Then a loop executes over and over again until the server is shut down (e.g., by CTRL/C at the terminal).

The loop executes the `accept` method, which blocks execution until some relative telnets in to the server. Then it creates a BufferedReader object that accepts input from the input stream associated with that client, as well as a PrintWriter object that sends output to the output stream associated with that client. After that, it sents these two communication channels to the Handler object so it can do its job, then closes the connection and waits for another connection.

Listing 12.10 The Telnetting application program

```
import java.net.*;
import java.io.*;

public class Telnetting
{
   public static void main (String[] args)
   {  try
      {  Handler birthNames = new Handler();
         ServerSocket server = new ServerSocket (12345, 20);
         for (;;)
         {  Socket client = server.accept();  // blocks
            BufferedReader input = new BufferedReader
                (new InputStreamReader(client.getInputStream()));
            PrintWriter output = new PrintWriter
                                 (client.getOutputStream());
            birthNames.getMore (input, output);
            client.close();
         }
      }catch (IOException e)
      {  System.out.println ("could not open server or socket");
      }
   }  //======================
}
//#############################################################


public class Handler
{
   private String itsInfo = "";

   public void getMore (BufferedReader input, PrintWriter output)
   {  try
      {  if (itsInfo.length() > 0)
            output.println ("Suggestions so far:" + itsInfo);
         output.println ("What do you suggest for a baby name?");
         output.flush();
         String data = input.readLine();
         while (data != null && data.length() > 0)
         {  itsInfo += " " + data;
            output.println ("Another suggestion? ENTER to quit");
            output.flush();
            data = input.readLine();
         }
         output.close();
      }catch (IOException e)
      {  // no need for any statements here
      }
   }  //======================
}
```

If this application is running in an account named `sam.ccsu.edu`, all a relative has to do to suggest a baby name is to enter the following command in his or her terminal window:

```
telnet sam.ccsu.edu 12345
```

*12.13 About File And JFileChooser (*Sun Library)*

You can make the user's choice of which file to use more comfortable with a standard library class named **JFileChooser**. This is a subclass of JComponent in the `javax.swing` package. You create a JFileChooser object and show either an Open dialog or a Save dialog. If it returns the APPROVE_OPTION, you can access the file chosen using a statements such as the following:

```
File selected = someJFileChooser.getSelectedFile();
FileReader file = new FileReader (selected);
```

- `new JFileChooser()` creates a JFileChooser object.
- `someJFileChooser.showOpenDialog(null)` returns one of the three int values listed below, after opening a window that allows the user to choose a file to open.
- `someJFileChooser.showSaveDialog(null)` returns one of the three int values listed below, after opening a window that allows the user to save a file.
- `JFileChooser.APPROVE_OPTION` int value indicating the user selected a file.
- `JFileChooser.CANCEL_OPTION` int value indicating the user canceled out.
- `JFileChooser.ERROR_OPTION` int value indicating that some error occurred.
- `someJFileChooser.getSelectedFile()` returns the File value chosen.

A **File** object is not a file itself, only the description of a file including the drive and subfolder. You need to open a FileReader to actually access the file itself. The File class is in java.io. It allows the following useful operations:

- `someFile.exists()` tells whether there actually is a file of that name and folder.
- `someFile.canRead()` tells whether you have permission to read from this file.
- `someFile.canWrite()` tells whether you have permission to write to this file.
- `someFile.delete()` removes this file from its folder.

12.14 Review Of Chapter Twelve

Listing 12.1, Listing 12.2, and Listing 12.3 illustrate the main new library facilities discussed in this chapter.

About sequential file input and output (all are in java.io):

- A **sequential input file** is a file from which you can only obtain values in the order in which they were written; you cannot jump directly to another input position in the file.
- A **sequential output file** is a file to which you can only write values in order.
- `new BufferedReader(new FileReader(filenameString))` **opens** a text file for sequential input. The FileReader constructor can throw an IOException.
- `someBufferedReader.readLine()` gets one line of input. It can throw an IOException.
- `new BufferedReader(new InputStreamReader(System.in))` opens the keyboard for input. It can throw an IOException.
- `new PrintWriter(new FileWriter(filenameString))` opens a text file for sequential output. Any pre-existing file of the same name is lost, unless you add a second parameter value of `true` to indicate appending. It can throw an IOException.
- `somePrintWriter.flush()` clears out all characters left in the buffer, if any.
- `somePrintWriter.close()` is executed when you are done with the file.
- `somePrintWriter.print(someString)` writes the characters to the file.
- `somePrintWriter.println(someString)` writes the characters to the file followed by a newline.
- `new PrintWriter(System.out)` opens the terminal window for output.

About the java.util.StringTokenizer and java.io.StreamTokenizer classes:

- `new StringTokenizer(inputString)` creates a tokenized form of the String of characters. A **token** is a sequence of non-whitespace characters with whitespace on each side, counting beginning and end of the String as whitespace (but you can customize that definition). StringTokenizer is in the `java.util` package.
- `someStringTokenizer.hasMoreTokens()` tells whether you can get another token from the `inputString` without throwing a RuntimeException.
- `someStringTokenizer.nextToken()` returns the next available token in the `inputString` and advances in the sequence.
- `new StreamTokenizer(someReader)` creates the file processing object.
- `someStreamTokenizer.nextToken()` returns an int value that tells whether the next token is a word, number, string, end-of-file, etc. It skips Java comments. It can throw an IOException.

About random-access file input and output (from java.io):

- `new RandomAccessFile (filenameString, "rw")` opens a file for random-access. Use `"r"` instead of `"rw"` if you will not be writing to it.
- `someRandomAccessFile.writeInt(someInt)` writes one int value.
- `someRandomAccessFile.writeChar(someChar)` writes one char value.
- `someRandomAccessFile.writeDouble(someDouble)` writes one double value.
- `someRandomAccessFile.writeBoolean (someBoolean)` writes one boolean.
- `someRandomAccessFile.writeChars(someString)` writes without a newline.
- `someRandomAccessFile.readInt()` returns an int value.
- `someRandomAccessFile.readChar()` returns a char value.
- `someRandomAccessFile.readDouble()` returns a double value.
- `someRandomAccessFile.readBoolean()` returns a boolean value.
- `someRandomAccessFile.readLine()` returns a String value. It only handles the basic Unicode values 0 to 255 well, stops reading at a newline or end-of-file, and does not include the newline character in the returned value.
- `someRandomAccessFile.close()` is executed when you are done with the file.
- Read through the documentation for the `java.io` package partially described in this chapter. Look at the API documentation at `http://java.sun.com/docs` or on your hard disk.

Other vocabulary to remember:

- An output file is **buffered** if it saves up in RAM the small chunks of data you give it until it has a big chunk of data that it can write to the hard disk or screen all at once. An input file is **buffered** if it gets one big chunk of data at a time from the hard disk and saves it in RAM until you ask for it, typically in small chunks. If you have a reason to move the saved-up output to its ultimate destination before the buffering algorithm feels like doing so, you **flush the buffer**.
- A **binary file** is one that stores values in the form that they have in RAM, not in the textual form that appears on the screen. **Random-access** means that you can read from or write to any point in the file without first going through all the values at the beginning of the file, which is what you have to do with a sequential file.
- The **file-pointer position** is the byte number at which the next read or write operation will take effect in a file. You use it to specify where you will next read or write in a random-access file.

Answers to Selected Exercises

12.1
```
public static void findCaps (String name) throws IOException
{   BufferedReader fi = new BufferedReader (new FileReader (name));
    for (String si = fi.readLine();  si != null;  si = fi.readLine())
        if (si.length() > 0 && si.charAt (0) >= 'A' && si.charAt (0) <= 'Z')
            System.out.println (si);
}
```
12.2
```
public int readInt ()
{   String si = readLine();
    try
    {   return si == null  ?  0  :  Integer.parseInt (si);
    }catch (NumberFormatException e)
    {   return 0;
    }
}
```
12.3
```
public static int numChars (String name) throws IOException
{   BufferedReader file = new BufferedReader (new FileReader (name));
    int count = 0;
    for (String si = file.readLine();  si != null;  si = file.readLine())
        count += si.length();
    return count;
}
```
12.4
```
public static double averageCharsPerLine (String name) throws IOException
{   BufferedReader file = new BufferedReader (new FileReader (name));
    int lines = 0;
    int count = 0;
    for (String si = file.readLine();  si != null;  si = file.readLine())
    {   lines++;
        count += si.length();
    }
    return lines == 0  ?  0  :  1.0 * count / lines;
}
```
12.5
```
public static boolean descending (String name) throws IOException
{   BufferedReader file = new BufferedReader (new FileReader (name));
    String previous = file.readLine();
    if (previous != null)
        for (String si = file.readLine();  si != null;  si = file.readLine())
        {   if (previous.compareTo (si) < 0)
                return false;
            previous = si;
        }
    return true;
}
```
12.10
```
public static void copy (BufferedReader infile, String outName) throws IOException
{   PrintWriter outfile = new PrintWriter (new FileWriter (outName));
    for (String si = infile.readLine();  si != null;  si = infile.readLine())
        outfile.println (si);
    outfile.close();
}
```
12.11
```
public static void underForty (String inName, String outName) throws IOException
{   BufferedReader infile = new BufferedReader (new FileReader (inName));
    PrintWriter outfile = new PrintWriter (new FileWriter (outName));
    for (String si = infile.readLine();  si != null;  si = infile.readLine())
        if (si.length() < 40)
            outfile.println (si);
    outfile.close();
}
```
12.16 Replace the line "if (line.hasMoreTokens())" by the following:
```
if (! line.hasMoreTokens())  itsSender = null;  else
```
12.17 Replace the body of the while statement by the following:
```
char ch = data.nextToken().charAt (0);
if (ch >= '0' && ch <= '9')
    count++;
```
12.18 That simpler expression will be true when the character is anything between '0' + 9 and '0' - 9, because integer division by 10 of any number from -1 to -9 yields 0.

12.21
```
String [] [] [] friend = new String [5] [25] [3];
friend [2] [12] [1]
```
12.22
```
int[][] age = new int [2][2].  (age[0][0] + age[0][1] + age[1][0] + age[1][1]) / 4.
```
12.23 Replace the line beginning with "if" by the following:
```
if (itsItem[from][to] > cutoff && from != to)
```

12.24
```
public int numPeopleSentBy (int code)  // in EmailSet
{     int count = 0;
      for (int to = 0;  to < itsSize;  to++)
            if (itsItem[code][to] > 0)
                  count++;
      return count;
}
```
12.25
```
public int maxSentTo (int code)  // in EmailSet
{     int max = itsItem[0][code];
      for (int from = 1;  from < itsSize;  from++)
            if (max < itsItem[from][code])
                  max =  itsItem[from][code];
      return max;
}
```
12.26
```
public boolean allLessThan (int cutoff)
{     return ! anyMoreThan (cutoff - 1);
}
```
12.30 Replace the method call in Listing 12.5 by the following:
printStatistics (database, args[1]);
Replace "PrintWriter out" in the heading of printStatistics by "String outName" and insert the following as the first statement in that method:
PrintWriter out = new PrintWriter (new FileOutputstream (outName));

12.31 Replace the body of the for-statement by the following:
```
      int numSent = numSentFrom (k);
      if (min > numSent)
            min = numSent;
```
12.36
```
public int howManyArriveAt (int city)
{     int count = 0;
      for (int dep = 0;  dep < MAX;  dep++)
            if (plane[dep][city] != null)
                  count++;
      return count;
}
```
12.37
```
public int numSeatsAvailableFrom (int city)
{     int count = 0;
      for (int arr = 0;  arr < MAX;  arr++)
            if (plane[city][arr] != null)
                  count += plane[city][arr].numSeats;
      return count;
}
```
12.38
```
public int flightsOverHalfFull ()
{     int count = 0;
      for (int dep = 0;  dep < MAX;  dep++)
            for (int arr = 0;  arr < MAX;  arr++)
                  if (plane[dep][arr] != null
                              && plane[dep][arr].numSeats < plane[dep][arr].numReservations)
                        count++;
      return count;
}
```
12.39
```
public int roundTrips()
{     int count = 0;
      for (int dep = 0;  dep < MAX;  dep++)
            for (int arr = dep + 1;  arr < MAX;  arr++)
                  if (plane[dep][arr] != null && plane[arr][dep] != null)
                        count++;
      return count;
}
```
12.43
```
public void readRecord (Flight flit, long filePosition)
{     raf.seek (filePosition);
      raf.readChars (flit.aircraftType);
      raf.readDouble (flit.ticketPrice);
      raf.readInt (flit.numSeats);
      raf.readInt (flit.numReservations);
      flit.filePosition = filePosition;
}
```
12.49 Remove the dstore answer command in two places and the dload answer command in one place.

13 Sorting and Searching

Overview

This chapter discusses several standard algorithms for sorting, i.e., putting a number of values in order. It also discusses the binary search algorithm for finding a particular value quickly in an array of sorted values. The algorithms described here can be useful in various situations. They should also help you become more comfortable with logic involving arrays. These methods would go in a utilities class of methods for Comparable objects, such as the CompOp class of Listing 7.2. For this chapter you need a solid understanding of arrays (Chapter Seven).

- Sections 13.1-13.2 discuss two basic elementary algorithms for sorting, the SelectionSort and the InsertionSort.
- Section 13.3 presents the binary search algorithm and big-oh analysis, which provides a way of comparing the speed of two algorithms.
- Sections 13.4-13.5 introduce two recursive algorithms for sorting, the QuickSort and the MergeSort, which execute much faster than the elementary algorithms when you have more than a few hundred values to sort.
- Sections 13.6-13.7 go further with big-oh analysis and introduce the HeapSort algorithm.
- Section 13.8 introduces Data Flow Diagrams as a tool for software design and applies it to a company that maintains a database of orders from customers. This company may need reports on its database in order of customer ID, other reports in order of date, others in order of product ID, etc. The sorting algorithms presented in this chapter can help with such reports.

13.1 The SelectionSort Algorithm For Comparable Objects

When you have hundreds of Comparable values stored in an array, you will often find it useful to keep them in sorted order from lowest to highest, which is **ascending order**. To be precise, ascending order means that there is no case in which one element is larger than the one after it. Sometimes you prefer to store them from highest to lowest, which is **descending order** (no element is smaller than the one after it). The problem is to write an independent method that puts an array into ascending order.

Finding the smallest in a partially-filled array

As a warmup to a sorting algorithm, look at a simpler problem for an array of Comparable objects: How would you find the smallest value within a given range of values in an array?

Say the parameters of the method are the array `item`, the first index `start`, and the ending index `end`, where the values to search are from `item[start]` up through `item[end]`; assume `start <= end`. You look at the first value: `smallestSoFar = item[start]`. Then you go through the rest of the values in the array one at a time. Whenever the current value is smaller than `smallestSoFar`, you store it in `smallestSoFar`. So an independent class method to do this would be as follows:

```
public static Comparable findMinimum (Comparable[ ] item,
                                      int start, int end)
{  Comparable smallestSoFar = item[start];
   for (int k = start + 1;  k <= end;  k++)
      if (item[k].compareTo (smallestSoFar) < 0)
         smallestSoFar = item[k];
   return smallestSoFar;
}  //======================
```

Say you apply this logic to the array in Figure 13.1 with `start` being 2 and `end` being 5. The figure indicates the values by decimal numbers to make this example clearer. You begin by noting the smallest so far is 5.0, which is the value at index 2. Next you go through each index value from index 3 on up, stopping after processing index value 5:

- You compare `item[3]` to 5.0 but make no change, because `item[3]` is not smaller than 5.0.
- You compare `item[4]` to 5.0 and assign that value 3.0 to the smallest so far, because `item[4]` is smaller than 5.0.
- You compare `item[5]` to 3.0 but make no change, because `item[5]` is not smaller than 3.0. The end result is that you return the value 3.0 from the method.

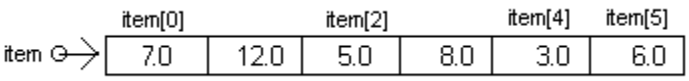


Figure 13.1 An array with six decimal values

Naturally this logic only works if the array has non-null Comparable values in components 2 through 5. Now try a mild modification of this logic: How would you find the smallest value and swap it with the value at index `start`? It is not enough to find the smallest; you have to find the index where the smallest is stored:

```
int indexSmallest = start;
for (int k = start + 1;  k <= end;  k++)
   if (item[k].compareTo (item[indexSmallest]) < 0)
      indexSmallest = k;
```

Then you can swap the value at that index with the value at index `start`. For instance, for the array in Figure 13.1, you would find that the index of the smallest is 4, because that is where the 3.0 is. So you swap the value at index 4 with the value at index `start`:

```
Comparable saved = item[start];
item[start] = item[indexSmallest];
item[indexSmallest] = saved;
```

The SelectionSort Algorithm

With this warmup, you can look at a standard method of putting all array values in ascending order. This algorithm is the **SelectionSort Algorithm**. The plan is to select the smallest of all the values and swap it into component 0. Then select the smallest of all the values from index 1 on up and swap it into component 1. At this point you have the two smallest values of all, in ascending order in the first two components of the array.

You next select the smallest of all the values from index 2 on up and swap it into component 2. Now you have the three smallest values of all, in ascending order in the first three components of the array. Next you select the smallest of all the values from index 3 on up and swap it into component 3. This continues until you come to the end of the values in the array. Then all of the values are in ascending order.

Figure 13.2 illustrates the sequence of steps on the array with six values, at indexes 0 through 5, from Figure 13.1. The process only needs five steps, because once the first five are at the right index, the last one must be (do you see why?). On Step 5a, the smallest of the two values is already at the right spot, so swapping it has no effect.

The indexes of the six components are:	0	1	2	3	4	5
And the values stored in the array are initially:	7	12	5	8	3	6
Step 1a: Find the location of the smallest at index 0...5:	?	?	?	?		?
Step 1b: Swap it with the value at index 0:	**3**	12	5	8	**7**	6
Step 2a: Find the location of the smallest at index 1...5:	"	?		?	?	?
Step 2b: Swap it with the value at index 1:	"	**5**	**12**	8	7	6
Step 3a: Find the location of the smallest at index 2...5:	"	"	?	?	?	
Step 3b: Swap it with the value at index 2:	"	"	**6**	8	7	**12**
Step 4a: Find the location of the smallest at index 3...5:	"	"	"	?		?
Step 4b: Swap it with the value at index 3:	"	"	"	**7**	**8**	12
Step 5a: Find the location of the smallest at index 4...5:	"	"	"	"		?
Step 5b: Swap it with the value at index 4:	"	"	"	"	**8**	12

Figure 13.2 Sequence of operations for the SelectionSort

This algorithm can be expressed in Java in just a few statements, using the coding previously developed. To sort a partially-filled array of Comparable values, call `swapMinToFront` for `start` having the values 0, 1, 2, 3, etc., stopping when `start` is `size-1`. The logic is in Listing 13.1. This method might be called from the NumericList class (whose instance variables are a partially-filled array `itsItem` of Numeric objects and an int value `itsSize` saying how many values in `itsItem` are useable) using this statement:

```
CompOp.selectionSort (itsItem, itsSize);
```

Listing 13.1 The selectionSort method for a partially-filled array, in CompOp

```
/** Precondition:  size >= 0 and item[0]...item[size-1] are
 *  non-null Comparable values, all comparable to each other.
 *  Postcondition: The array contains the values it initially
 *  had but with item[0]...item[size-1] in ascending order. */

public static void selectionSort (Comparable[] item, int size)
{  for (int k = 0;  k < size - 1;  k++)
      swapMinToFront (item, k, size - 1);
}  //======================

private static void swapMinToFront (Comparable[] item,
                                    int start, int end)
{  int indexSmallest = start;
   for (int k = start + 1;  k <= end;  k++)
      if (item[k].compareTo (item[indexSmallest]) < 0)
         indexSmallest = k;
   Comparable saved = item[start];
   item[start] = item[indexSmallest];
   item[indexSmallest] = saved;
}  //======================
```

An example of an independent class method that calls on the `selectionSort` method to sort 3000 randomly-chosen decimal numbers and then reports the median value is the following. It uses the Double methods described in Chapter Eleven: `new Double(x)` produces an object with a single instance variable storing the double value x, and the Double class implements the Comparable interface.

```
public static double median()        // independent
{  final int numToSort = 3000;
   Double[] item = new Double [numToSort];
   for (int k = 0;  k < numToSort;  k++)
      item[k] = new Double (Math.random());
   selectionSort (item, numToSort);
   return item[numToSort / 2].doubleValue();
}  //======================
```

Exercise 13.1 Revise Listing 13.1 to perform a SelectionSort on a null-terminated array (such as described for NumericArray in Listing 9.8): You have only the `item` array parameter, no `size`. You are to sort all values up to the first one that is `null`.
Exercise 13.2 Revise Listing 13.1 to put the values in descending order.
Exercise 13.3* Rewrite Listing 13.1 to find the largest value each time and swap it to the rear of the array.
Exercise 13.4* Rewrite Listing 13.1 to omit swapping when the smallest is already at the front. Does this speed up or slow down the execution?

13.2 The InsertionSort Algorithm For Comparable Objects

The InsertionSort is another standard sorting algorithm. As a warmup, start with this problem: If you know that the first `m` values in an array are already in ascending order, but the next one (at `item[m]`) is probably out of order, how would you get them all in order? The accompanying design block is a plan for the solution.

DESIGN to re-order values
1. Set the value from `item[m]` aside, thereby leaving an empty spot in the array.
2. Compare the value before the empty spot with the value you set aside.
   If that value before is larger, then...
   - 2a. Move that value into the empty spot, so where it came from is now empty.
   - 2b. Repeat from Step 2.
3. Put the value you set aside in the empty spot.

This logic is defective: What if all of the values are larger than the one you set aside? You have to stop when the empty spot is at the very front of the array, i.e., `item[0]`. Making this adjustment, the following method solves the problem.

```
private static void insertInOrder (Comparable[ ] item, int m)
{  Comparable save = item[m];
   for (;  m > 0 && item[m - 1].compareTo (save) > 0;  m--)
      item[m] = item[m - 1];
   item[m] = save;
}  //======================
```

The InsertionSort Algorithm

Now that you have had a warmup, you can look at the second standard algorithm to put the array values in ascending order. This algorithm is the **InsertionSort Algorithm**. The plan is as follows (see the illustration in Figure 13.3):

1. Put the first two values in order.
2. Insert the third value in its proper place in the sequence formed by the first two.
3. Insert the fourth value in its proper place in the sequence formed by the first three.
4. Insert the fifth value in its proper place in the sequence formed by the first four.
5. Keep this up until you have them all in order.

Suppose the values stored in the array are initially:	12	7	8	5	13	6	9
Step 1: Put the first two values in order:	7	12	"	"	"	"	"
Step 2: Insert the third in order among the first two:	7	**8**	12	"	"	"	"
Step 3: Insert the fourth in order among the first three:	**5**	7	8	12	"	"	"
Step 4: Insert the fifth in order among the first four:	5	7	8	12	**13**	"	"
Step 5: Insert the sixth in order among the first five:	5	**6**	7	8	12	13	"
Step 6: Insert the seventh in order among the first six:	5	6	7	8	**9**	12	13

Figure 13.3 Sequence of operations for the InsertionSort

You will have noticed that the logic needed for each of Steps 2, 3, 4, etc. is in the `insertInOrder` method just developed. In fact, it can be used for Step 1 as well (calling the method with `m == 1`). The complete InsertionSort algorithm is coded in Listing 13.2. The precondition and postcondition are the same as for Listing 13.1.

Listing 13.2 The insertionSort method for a partially-filled array, in CompOp

```
/** Precondition:  size >= 0 and item[0]...item[size-1] are
 *  non-null Comparable values, all comparable to each other.
 *  Postcondition: The array contains the values it initially
 *  had but with item[0]...item[size-1] in ascending order. */

public static void insertionSort (Comparable[] item, int size)
{  for (int k = 1;  k < size;  k++)
      insertInOrder (item, k);
}  //======================

private static void insertInOrder (Comparable[] item, int m)
{  Comparable save = item[m];
   for (;  m > 0 && item[m - 1].compareTo (save) > 0;  m--)
      item[m] = item[m - 1];
   item[m] = save;
}  //======================
```

Contrasting the two sorting algorithms

At each call of `insertInOrder`, the list of values is actually two lists: The "bad list", which is all the unsorted values indexed `k` and higher, versus the "good list" of sorted values at indexes `0..k-1`. Each call of `insertInOrder` increases by 1 the number of items in the good list and consequently decreases by 1 the number of items in the bad list. When the InsertionSort finishes, the entire list is the good list, i.e., it is sorted.

For the SelectionSort, the bad list is also the unsorted values indexed `k` and higher, and the good list of sorted values is at indexes `0..k-1`. Each call of `swapMinToFront` increases by 1 the number of items in the good list and consequently decreases by 1 the number of items in the bad list. When the SelectionSort finishes, the entire list is the good list, i.e., it is sorted.

Both algorithms gradually transform the bad into the good, one value at a time. The way they transform differs, but the overall effect is the same: What was originally all bad (unsorted) is at the end all good (sorted).

The difference between them can be described this way: The SelectionSort logic slowly selects a value (the smallest) from the bad list so it can quickly put it on the end of the

good list. The InsertionSort logic quickly selects a value (the first one available) from the bad list so it must slowly put it where it goes within the good list.

Loop invariants verify that the algorithms work right

A **loop invariant** is a condition that is true every time the continuation condition of a looping statement is evaluated. For these two sorting algorithms, a loop invariant tells what it means to be a "good" list:

Loop invariant for the main loop of insertionSort (any time at which the loop condition k < size is evaluated): The values in item[0]...item[k-1] are in ascending order and are the same values that were originally in item[0]...item[k-1].

How do we know this condition is in fact true every time `k < size` is evaluated? It depends on two facts that you can easily check out:

1. The first time `k < size` is evaluated, `k` is 1, so item[0]...item[k-1] contain only one value, so they are by definition in ascending order.
2. If at any point when `k < size` is evaluated, item[0]...item[k-1] are in ascending order and are the values that were originally there, then one iteration of the loop shifts the last few values up by one component and inserts `item[k]` into the place that was vacated by the shift, immediately after the rightmost value that is less than or equal to `item[k]`. So item[0]..item[k] are now in ascending order and are the values that were originally in components 0 through `k`. Then, just before testing `k < size` again, k is incremented, which means that the invariant condition is again true: item[0]...item[k-1] are the original first `k` values in ascending order.

Once you see that both of those assertions are true, you should be able to see that together they imply that the loop invariant is true when the loop terminates: item[0]...item[k-1] are in ascending order and are the original first `k` values. But at that point, `k` is equal to `size`, which means that item[0]...item[size-1] are in ascending order and are the original `size` values. Conclusion: The `insertionSort` coding sorts the values in the partially-filled array.

The same pattern of reasoning can be applied to verify that the `selectionSort` coding in the earlier Listing 13.1 actually sorts the values given it:

Loop invariant for the main loop of selectionSort (any time at which the loop condition k < size-1 is evaluated): The values in item[0]...item[k-1] are in ascending order and are the k smallest values that were originally in item[0]...item[size-1].

How do we know that condition is in fact true every time `k < size-1` is evaluated? It depends on two facts that you can easily check out:

1. The first time `k < size-1` is evaluated, `k` is zero, so item[0]...item[k-1] contain no values at all, so they are the 0 smallest values in ascending order (vacuously).
2. If at any point when `k < size-1` is evaluated, item[0]...item[k-1] are the `k` smallest of the original values in ascending order, then one iteration of the loop finds the (k+1)th smallest of the original values and puts it after the k smallest values. So item[0]...item[k] are now in ascending order and are the `k+1` smallest values that were originally in the array. Then, just before testing `k < size-1` again, k is incremented, which means that the invariant condition is again true: item[0]...item[k-1] are the k smallest of the original values in ascending order.

Once you see that both of those assertions are true, you should be able to see that together they imply that the loop invariant is true when the loop terminates: item[0]...item[k-1] are the `k` smallest of the original values in ascending order. But at

that point, `k` is equal to `size-1`, which means that item[0]...item[size-2] are in ascending order and are all but the largest of the original values. Conclusion: The `selectionSort` algorithm sorts the values in the partially-filled array.

As a general principle, when you have a complex looping algorithm and you want to verify that it produces a given result under any and all conditions, proceed as follows: Find a condition that is trivially true the first time the loop condition is evaluated, and is "incrementally true" for the loop (i.e., if true at the time of one evaluation, it is true at the time of the next evaluation), so you can deduce that it will be true when the loop terminates. If you have chosen the condition well, its truth when the loop terminates will be an obvious proof that the loop produces the required result.

Exercise 13.5 Rewrite `insertInOrder` in Listing 13.2 to check whether the value to be set aside is in fact smaller than the one in front of it; if not, do not set it aside. Does this speed up or slow down the algorithm?
Exercise 13.6 Revise Listing 13.2 to perform an InsertionSort on a null-terminated array: You have only the `item` parameter, no `size`. You are to sort all values up to the first one that is `null`.
Exercise 13.7 Revise Listing 13.2 to perform an InsertionSort on an array of double values.
Exercise 13.8 Revise Listing 13.2 to allow for `null` values in the array. A `null` value is to be considered larger than any non-null value.
Exercise 13.9* The loop condition in `insertInOrder` makes two tests each time through. Rewrite the method to execute faster by making only one test on each iteration, as follows: First see if `item[0]` is larger than `save`. If not, have a faster loop to insert `save` where it goes, otherwise have a completely separate loop to insert `save` at the front of the array.

13.3 Big-oh And Binary Search

Why have two or more different sorting methods? Because one may be better for one purpose and another may be better for a different purpose. We will look at the execution time of the two elementary algorithms you have seen so far.

Counting the comparisons made

When you study the logic in Listing 13.2, you should be able to see that a call of `insertInOrder` with `k` as the second parameter could make anywhere from 1 to `k` comparisons. So if `N` denotes the total number of items to be sorted, the InsertionSort could make as many as `1 + 2 + 3 + ... + (N-1)` comparisons of data altogether before it gets them all in order. That adds up to `(N-1)*N/2`, using basic counting methods. In effect, each of the `N` items could be compared with each of the other `(N-1)` items before the sorting is done.

If you look back at the logic in the earlier Listing 13.1, you will see that the first time you call `swapMinToFront`, when `k` is zero, you will make `N-1` comparisons. The second time you will make `N-2` comparisons. This continues until `k` is the index of the next-to-last value, which requires only one comparison. So the total number of comparisons is `1 + 2 + 3 +...+(N-1)`, i.e., `(N-1)*N/2`.

The `(N-1)*N/2` comparisons that the InsertionSort could make is the **worst case**; the average should be somewhere around one-quarter of N-squared. The SelectionSort always takes almost one-half of N-squared comparisons, but it usually has much less movement of data than the InsertionSort. Which is faster depends on the time for a comparison versus the time for a movement of data.

Both the InsertionSort and the SelectionSort are called **elementary sorting algorithms**. "Elementary" means that the number of comparisons made in the process of sorting N items is on average a constant multiple of N-squared. The multiple for InsertionSort is ¼ and the multiple for SelectionSort is ½. There are other elementary sorting algorithms; the first one invented for computer programs was probably the one called the BubbleSort. But it executes more slowly than the two described here.

The technical way of saying this is that **the big-oh behavior of elementary sorts is N-squared**, where N is the number of items to be sorted. It means that, if it takes X amount of time to sort a certain number of items, then it takes roughly 4X amount of time to sort twice as many items, and it takes roughly 100X amount of time to sort ten times as many items, at least if X is fairly large. In general, the amount of time to do the job is roughly proportional to the square of the number of items to be processed, for large values of N.

To see this, suppose you have a processor that takes 1 second to sort 1000 items. That 1 second is what you need for roughly M times 1 million comparison operations, where M is the multiplier for the elementary sort (M == ¼ for the InsertionSort, M == ½ for the SelectionSort) and 1 million is the square of 1000. To sort 2000 items requires M times 4 million comparisons, which is four times as long as for sorting 1000 items. To sort 10,000 items requires M times 100 million comparisons, which is 100 times as long as for sorting 1000 items.

In particular, it would take a thousand-squared seconds to sort one million items, which is over ten days. This is not practical. The next section discusses a faster algorithm for sorting. But first we look at an algorithm for searching an array and its execution time.

Binary Search

A key algorithm for searching an array for a value when the array is already sorted in ascending order is called Binary Search. The logic of the standard **Binary Search Algorithm** is as follows: The target value, if it is in the array, must be in the range from the `lowerBound` of 0 to the `upperBound` of `size-1`, inclusive. First look at the value at the middle component `item[midPoint]`. If that is less than the target value, the target can only be above that component, because the values are sorted in ascending order; so the revised `lowerBound` is `midPoint+1`. On the other hand, if `item[midPoint]` is not less than the target value, the target must be in the range from `lowerBound` to `midPoint`, inclusive; so the revised `upperBound` is `midPoint`.

Whichever of the two cases apply, you have cut the number of possibilities about in half. This process can be repeated until `lowerBound` is equal to `upperBound`. At that point, the target value must be in the only component in that range, if it is in the array at all.

Figure 13.4 illustrates a search for the value 40 in an ordered array of 16 values. Initially `lowerBound` is 0 and `upperBound` is 15 (listed in the first two columns), so `midPoint` is 7 (in the third column). 40 is greater than `item[7]` (boldfaced in the figure) so we change `lowerBound` to 8. Now `midPoint` is 11 and 40 is not greater than `item[11]`. This continues until we isolate the value in `item[10]`.

This logic expressed in Java is in Listing 13.3 (see next page). It includes an extra test to make sure `size` is positive before beginning the subdivision process.

lB	uB	m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	15	7	20	22	24	26	28	30	32	**34**	36	38	40	42	44	46	48	50
8	15	11									36	38	40	**42**	44	46	48	50
8	11	9									36	**38**	40	42				
10	11	10											**40**	42				
10	10												40					

Figure 13.4 Finding the value 40 in an array of 16 values

Listing 13.3 The binarySearch method for an ordered partially-filled array

```
/** Precondition:  size >= 0 and item[0]...item[size-1] are
 *  non-null Comparable values, all comparable to target.
 *  Returns: whether target is one of those size values.  */

public static boolean binarySearch (Comparable[] item,
                                    int size, Comparable target)
{  if (size <= 0)                                              //1
      return false;                                            //2
   int lowerBound = 0;                                         //3
   int upperBound = size - 1;                                  //4
   while (lowerBound < upperBound)                             //5
   {  int midPoint = (lowerBound + upperBound) / 2;            //6
      if (item[midPoint].compareTo (target) < 0)               //7
         lowerBound = midPoint + 1;                            //8
      else                                                     //9
         upperBound = midPoint;                                //10
   }                                                           //11
   return item[lowerBound].equals (target);                    //12
}  //======================
```

Counting the comparisons made

If the number of items to be searched is 1000, then you only need eleven comparisons to find out whether the target value is in the array. This is because, when you start from 1000 and repeatedly divide by 2, you reduce it to just one possibility in only 10 iterations; the eleventh comparison is to find out whether that one possibility is the target value.

The number of times you divide a number by 2 to get it down to 1 is its **logarithm base 2** (more precisely, it is the logarithm after rounding up). Call this rounded-up number `log2(N)`. `log2(1000)` is 10, so binary search requires 11 comparisons. Similarly, `log2(1,000,000)` is 20, so binary search only requires 21 comparisons. And `log2(1,000,000,000)` is 30, so binary search of a billion items only requires 31 comparisons.

The big-oh

In general, if N is the number of items to be searched, then the Binary Search Algorithm requires `log2(N)+1` comparisons to find out whether a particular target value is there. The technical way of saying this is that **the big-oh behavior of binary search is log(N)**, where N is the number of items to be searched. It means that the amount of time to do the job is roughly proportional to the logarithm of the number of items to be processed, for large values of N. A useful loop invariant for the binarySearch method is the following:

Loop invariant for the one loop in binarySearch (any time at which the loop condition lowerBound < upperBound is evaluated): The target value is in item[lowerBound]...item[upperBound] or else is not in the array at all.

By contrast, the **sequential search** algorithm that you have seen many times before looks at potentially every element in the list. The **big-oh behavior of sequential search is N**: the amount of time to do the job is roughly proportional to the number of items to be processed, for large values of N.

Exercise 13.10 The `binarySearch` logic divides the possibilities up into two parts, but they are not always the same size. When is one part larger by one item? Which part of the array is larger in that case, the front or the rear half?
Exercise 13.11 Explain what Exceptions could be thrown, and when, if the statement testing `size` were omitted from the coding of `binarySearch`.

Exercise 13.12 In the `binarySearch` method, what would be the consequence of rounding off `midPoint` in the other direction from what is done in Listing 13.3 (i.e., what Exceptions could be thrown or other problems occur)?
Exercise 13.13* Rewrite the `binarySearch` method to check to see whether `item[midPoint]` actually equals the target value each time and, if so, to stop the process early. Then explain why this slows execution on average.
Exercise 13.14* Write a `binarySearch` method to return an int value: Return the index where the target was found, except return a negative int `-n-1` if the target is not in the array (n is the index where target should be inserted to keep all values in order).
Exercise 13.15** Write a `binarySearch` method for a null-terminated array with execution time big-oh of log(L) where L is the length of the array.
Exercise 13.16** Write out logical reasoning to verify that the loop invariant for `binarySearch` is trivially true at the first test and incrementally true for each iteration.

13.4 The Recursive QuickSort Algorithm For Comparable Objects

People have made attempts to improve on the speed of the elementary sorting algorithms of selection sort and insertion sort. One idea is the **QuickSort** logic, which the accompanying design block describes, assuming you create a QuickSorter object whose job it is to sort the elements of an array.

DESIGN for the sorting logic of a QuickSorter object
If you have at least two elements to sort in ascending order, then...

1. Choose one of the elements of the array; call it the `pivot`.
2. Move everything smaller than the `pivot` towards the front of the array.
3. Move everything larger than the `pivot` towards the rear of the array.
4. Put the `pivot` in the component between those two groups of elements.
5. Have some other QuickSorter object sort the values before the pivot.
6. Have some other QuickSorter object sort the values after the pivot.

Why this is usually faster

Let us compare this logic with that of the insertion sort. Say you find that it takes 800 milliseconds to sort 2000 items with the insertion sort logic. Since it makes 1999 passes through the data (calls of `insertInOrder`), the average call of `insertInOrder` takes 0.4 milliseconds. The average call of `insertInOrder` inserts a value into about 1000 items and makes about 500 comparisons to do so. Steps 1 through 4 of the QuickSort logic go through all 2000 items, so that must take under 2 milliseconds.

Say steps 5 and 6 called `insertionSort` instead of using another QuickSort logic, once for the half that are smaller than the pivot and once for the half that is larger. Then each call would take about 200 milliseconds, one quarter of the 800 for sorting 2000, since the execution time of `insertionSort` is roughly proportional to the square of the number of items sorted. So altogether the QuickSort logic would take 2+200+200 = 402 milliseconds instead of 800 milliseconds. And that is assuming it switches to the `insertionSort` logic for steps 5 and 6. If it continued with the QuickSort logic, the improvement in speed would be far greater.

You probably noticed a big IF in that logic: If the pivot turns out to divide the values roughly in half, it almost doubles the speed. But the pivot is chosen at random, so it may be very close to one end or the other of the range of values. That would make the execution time about the same as the insertion sort, possibly a bit worse.

This is the drawback to the QuickSort: If the pivot is one of the five or so smallest values, or one of the five or so largest values, the vast majority of the time the pivot is chosen,

then the QuickSort can take a little longer than the insertion sort. But that is extremely rare for randomly-ordered data. The QuickSort will almost always drastically speed up the sorting process relative to the insertion sort. An exercise has you implement a "tune-up: in which you select the **middle of three** values taken from the bottom, the top, and the middle of the array.

Recursion

A single method may be called many times at different points during the execution of the program. Each of these calls is a different **activation** of the method. The runtime system records information about each activation in separate "method objects" that it creates and discards during execution of the program. You must keep this point firmly in mind when thinking about **recursion**: A method is allowed to contain a call of itself. The QuickSort logic requires the use of recursion.

As an example, the `insertionSort` method could have been written recursively as follows (though we normally do not do this where a simple for-statement accomplishes the same thing):

```
public static void insertionSort (Comparable[] item, int size)
{  if (size > 1)
   {  insertionSort (item, size - 1);               //recur
      insertInOrder (item, size - 1);
   }
}  //======================
```

For that matter, the `selectionSort` method could have been written recursively if we had chosen to have a method that swaps the largest value to the rear of the array instead of a method that swaps the smallest value to the front of the array:

```
public static void selectionSort (Comparable[] item, int size)
{  if (size > 1)
   {  swapMaxToRear (item, 0, size - 1);
      selectionSort (item, size - 1);               //recur
   }
}  //======================
```

Say you want to print out all the 6-digit binary numbers (all 64 of them). Then a call of `printBinary ("", 6)` does the job if you have the following method. Study it for a while to see that it prints out every binary number that has `numDigits` digits, each one prefixed by the given String `prefix`:

```
public static void printBinary (String prefix, int numDigits)
{  if (numDigits <= 1)
      System.out.println (prefix + "0\n" + prefix + "1");
   else
   {  printBinary (prefix + "0", numDigits - 1);      //recur
      printBinary (prefix + "1", numDigits - 1);      //recur
   }
}  //======================
```

If you try to do that with a while-statement, it will not be nearly as compact and clear. In the three methods just given, each expression that makes a recursive call has this hallmark property:

(a) it is guarded by a test that a certain variable, called the **recursion-control variable**, is greater than a certain **cutoff** amount, and
(b) it passes to the new activation a value of the recursion-control variable that is at least 1 smaller than the existing activation has.

In the three examples just given, the expression that makes the recursive call is marked `//recur` out at the side:

- For `insertionSort`, the recursion-control variable is `size` and its cutoff is 1.
- For `selectionSort`, the recursion-control variable is `size` and its cutoff is 1.
- For `printBinary`, the recursion-control variable is `numDigits` and its cutoff is 2.

In some applications of recursion, the coding itself does not declare a variable specifically as the recursion-control variable. But any coding that uses recursion properly will involve a quantity that could be assigned to a recursion-control variable if need be.

Of course, it is possible to write recursive coding that falls into an "infinite loop" because it does not have a recursion-control variable. But then again, it is also possible to write a while-statement that falls into an "infinite loop" because it does not have an "iteration-control variable" that eventually reaches a certain cutoff and terminates the loop.

It is the recursion-control variable and the cutoff value that guarantee that recursion does not go on forever. Think of each activation of the method as being a certain height above the floor. The recursion-control variable measures the height above the floor. Each time you step from one activation to another, the height drops. When you spiral down to the floor (the cutoff value), the recursion stops. In short, a correctly coded recursion process does not go in circles, it goes in spirals.

Object design

A reasonable approach is to create a QuickSorter object by giving it the array of values that contain the values to be sorted. Then you can ask it to sort one part or another part of the array. So the primary message sent to a QuickSorter object should be as follows, where `start` and `end` are the first and last indexes of the part of the array to be sorted:

```
someQuickSorter.sort (start, end);
```

The object responds to the `sort` message by first making sure that it has at least two values to sort, i.e., that `start < end`. If so, it rearranges the values around the pivot, then has some other QuickSorter object sort each half. Since the rearranging is rather complex, we could put it off to some other method, which will report the index where the pivot was put. That leads to the following basic coding for the `sort` method (assuming that the QuickSorter's instance variable is named `itsItem`, as usual):

```
int mid = this.split (start, end);
QuickSorter helper = new QuickSorter (itsItem);
helper.sort (start, mid - 1);
helper.sort (mid + 1, end);
```

The `split` method is to split the sortable portion of the array into two parts, those below index `mid` that are smaller than the pivot, and those above index `mid` that are larger than the pivot. The `helper` object then sorts each part separately. This overall logic is illustrated in Figure 13.5.

Development of the split method

Splitting the portion of the array in parts smaller than and larger than a pivot value is tricky. Call the lowest and highest indexes for the range `lo` and `hi`. They are initialized to the `start` and `end` values, but `lo` and `hi` will change during the logic whereas `start` and `end` will stay their original values. So it will be less confusing to have different names.

The initial array	7	12	5	10	3	8	6
Remove the pivot 7	**X**	12	5	10	3	8	6
Split, smaller below, larger above	6	3	5	**X**	10	8	12
Put the pivot back in the middle	6	3	5	**7**	10	8	12
Sort the ones smaller than 7	3	5	6	**7**	10	8	12
Sort the ones larger than 7	3	5	6	**7**	8	10	12

Figure 13.5 The overall logic of the QuickSort

Take the pivot value from `itsItem[lo]`. That leaves the component empty. What should go there at the lower part of the array? An element that is smaller than the pivot. And where should you take it from? From where it should not be, namely, the higher part of the array. So look at `itsItem[hi]` to see if that is smaller than the pivot. Say it is not. Then `itsItem[hi]` is in the right place, so decrement `hi` and look at `itsItem[hi]`, which is the one below the original `hi` one. Say that one is smaller than the pivot. So you move the contents of `itsItem[hi]` down to the empty spot `itsItem[lo]`, which leaves `itsItem[hi]` empty.

What should go into `itsItem[hi]`, in the higher part of the array? An element that is larger than the pivot. And where should you take it from? From where it should not be, namely, the lower part of the array. You start looking at `itsItem[lo+1]`. So you increment `lo` to keep track of the position in the lower part of the array. Keep incrementing `lo` until you get to a component that has a larger value than the pivot. Move that value up into the empty spot at `itsItem[hi]`. Then go back to looking in the higher part of the array for a value to move into the now-empty spot `itsItem[lo]`.

The coding for this logic is in Listing 13.4 (see next page). Keep track of whether you are looking in the higher part of the array (versus the lower part) with a boolean variable `lookHigh` (line 8). When it is true, you are decrementing `hi` in the higher part of the array until you find a smaller value than the pivot, which you then move into the empty `itsItem[lo]` (line 14). When `lookHigh` is false, you are incrementing `lo` in the lower part of the array until you find a larger value than the pivot, which you then move into the empty `itsItem[hi]` (line 21).

This coding uses a language feature not previously mentioned: The phrase `itsItem[lo++]=` in line 14 assigns a value to `itsItem[lo]` and thereafter increments `lo`. In general, the value of the expressions `x++` and `x--` is the value that `x` had before it incremented or decremented. Thus `itsItem[hi--]` in line 22 assigns a value to `itsItem[hi]` and thereafter decrements `hi`. If you use this shorthand feature, follow this safety rule: Never use it in a statement that mentions the variable (`x` or whatever) more than once. The reason is that its value is very tricky to figure out in such a case.

Note that most QuickSorter objects call the `sort` method twice, and on each such call create a new helper. A worthwhile improvement in the logic is to have it create a helper only on the first call and remember that helper for the second call. This is an exercise.

The loop invariant

When `lo` and `hi` coincide, you have found the empty spot where the pivot should go. You can verify that the loop in the `split` method does what it should do by checking that the following statement is a loop invariant:

Listing 13.4 The QuickSort Algorithm for a partially-filled array

```
// a method in the CompOp class; same conditions as Listing 13.1
   public static void quickSort (Comparable[] item, int size)
   {  new QuickSorter (item).sort (0, size - 1);
   }  //======================

public class QuickSorter extends Object
{
   private Comparable[] itsItem;

   public QuickSorter (Comparable[] item)
   {  super();
      itsItem = item;
   }  //======================

   /** Precondition: start <= end; itsItem[start]...itsItem[end]
    *  are the Comparable values to be sorted. */

   public void sort (int start, int end)
   {  if (start < end)                                          //1
      {  int mid = split (start, end);                          //2
         QuickSorter helper = new QuickSorter (itsItem);        //3
         helper.sort (start, mid - 1);                          //4
         helper.sort (mid + 1, end);                            //5
      }                                                         //6
   }  //======================

   private int split (int lo, int hi)
   {  Comparable pivot = itsItem[lo];                           //7
      boolean lookHigh = true;                                  //8
      while (lo < hi)                                           //9
         if (lookHigh)                                          //10
            if (itsItem[hi].compareTo (pivot) >= 0)             //11
               hi--;                                            //12
            else                                                //13
            {  itsItem[lo++] = itsItem[hi];                     //14
               lookHigh = false;                                //15
            }                                                   //16
         else                                                   //17
            if (itsItem[lo].compareTo (pivot) <= 0)             //18
               lo++;                                            //19
            else                                                //20
            {  itsItem[hi--] = itsItem[lo];                     //21
               lookHigh = true;                                 //22
            }                                                   //23
      itsItem[lo] = pivot;                                      //24
      return lo;                                                //25
   }  //======================
}
```

Loop invariant for the while loop in split (any time when the loop condition lo < hi is evaluated): No value from start through lo-1 is larger than the pivot and no value from hi+1 through end is smaller than the pivot. Moreover, itsItem[lo] is "empty" if lookHigh is true, but itsItem[hi] is "empty" if lookHigh is false.

By "empty" we mean that you may overwrite that value without losing any value that was in the original portion of the array to be sorted. In Figure 13.6, the X marks the

component that is "empty". The positions of `lo` and `hi` after the action described at the left of Figure 13.6 are boldfaced so you can track their movements. On each iteration of the loop, one or the other moves one step closer to the middle.

Initially lo=20 and hi = 26:	20	21	22	23	24	25	26
The array values are (boldfaced at lo and at hi):	**7**	12	5	10	3	8	**6**
First: Take the pivot from index lo, so pivot is 7:	**X**	12	5	10	3	8	**6**
Iteration 1: lookHigh is true, itsItem[hi]<pivot so move:	6	**12**	5	10	3	8	**X**
Iteration 2: lookHigh is false, itsItem[lo]>pivot so move:	6	**X**	5	10	3	**8**	12
Iteration 3: lookHigh is true but itsItem[hi]>pivot so:	6	**X**	5	10	**3**	8	12
Iteration 4: lookHigh is true, itsItem[hi]<pivot so move:	6	3	**5**	10	**X**	8	12
Iteration 5: lookHigh is false, itsItem[hi]<pivot so:	6	3	5	**10**	**X**	8	12
Iteration 6: lookHigh is false, itsItem[hi]>pivot so move:	6	3	5	**X**	10	8	12
Now lo == hi, so X marks the spot where pivot goes:	6	3	5	**7**	10	8	12

Figure 13.6 Sequence of operations for the split method

Language elements
For any int variable k, k++ has the value that k had before it was incremented. Similarly, k-- evaluates as the value k had before it was decremented. So if k is 4, then k++ is 4 and k-- is 4. On the other hand, ++k yields the value k had after incrementing and --k yields the value that k had after decrementing. Examples: If k is 4, then ++k is 5 and --k is 3.
This compactness comes with an increased risk of getting the logic wrong.
It is most dangerous if you mention k anywhere else in the same statement.

Exercise 13.17 Trace the action of the split method on the sequence {7, 2, 9, 1, 5, 8, 4} in the way shown in Figure 13.6.
Exercise 13.18 Rewrite the `split` method to take the pivot from index `hi` instead of index `lo`.
Exercise 13.19 Let us add the instance variable `private QuickSorter itsHelper` to the QuickSorter class, initially `null`. Revise the coding of the `sort` method so that a QuickSorter object creates `itsHelper` if it does not have one, otherwise it uses the one it has.
Exercise 13.20 Explain why the loop invariant is trivially true at the first test of the loop condition.
Exercise 13.21* Explain why the loop invariant is incrementally true for each iteration.
Exercise 13.22* Rewrite the `split` method to find a pivot value that is much more likely to be close to the middle, as follows: Compare the three values at indexes `lo`, `hi`, and `(lo+hi)/2`. Whichever of them is between the other two is to be the pivot.
Exercise 13.23* What difference would it make if the >= and <= operators were replaced by > and < respectively in the `split` method?
Exercise 13.24* Write an InsertionSorter class of objects analogous to QuickSorter. This would allow people to use the InsertionSort logic for any subsequence of an array, not just starting from zero.

13.5 The Recursive MergeSort Algorithm For Comparable Objects

The irritating thing about the QuickSort algorithm is that it is unreliable. The vast majority of the time, it executes faster than the InsertionSort for a large number of values. But occasionally it is not significantly faster, and rarely it is slower. The key to the problem is that the pivot value does not always divide the group of values to be sorted in two equal parts. The MergeSort algorithm presented in this section avoids this problem.

Putting a mostly sorted group in order

For a warmup, consider this problem: Somewhere in the `item` array is a sequence of values. The first half of them are in ascending order and the second half of them are also in ascending order. Your job is to rearrange this sequence so that they are all in ascending order.

This could be very messy, swapping values all around, except for one thing: You get to have another array to put the values in as you sort them. That makes it so much easier. Look at a picture to see what is going on: The upper array in Figure 13.7 has two subsequences of four values, each sequence in ascending order. The lower array shows what the result of the process should be.

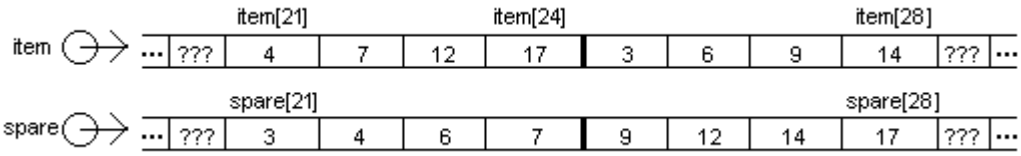


Figure 13.7 Problem: Move the 8 values into spare in ascending order

There is really only one decent way to go about this solving this problem, described in the accompanying design block. The key point is that the smallest of all of the values must be either the first (smallest) value in the lower half or the first (smallest) value in the higher half. Compare these two values. Whichever is smaller, put that one in the first available spot in the spare array and move on to the one after it in its subsequence.

STRUCTURED DESIGN for merging two ascending sequences

1. Set `lo` to be the first index of the lower group of ordered values.
2. Set `hi` to be the first index of the higher group of ordered values.
3. Compare the values at `lo` and `hi` in the `item` array to see which is smaller.
4. If the one at `lo` is smaller, then..
   2a. Move the one at `lo` to the next available spot in the `spare` array and increment `lo`.
5. Otherwise...
   3a. Move the one at `hi` to the next available spot in the `spare` array and increment `hi`.
6. Repeat from Step 3 until you have put all the values into the `spare` array.

Suppose we name the highest index value that `hi` can have `end` (as usual) and we name the highest index value that `lo` can have `mid` (because it is in the middle of the portion to be sorted). In Figure 13.7, for instance, `mid` is 24 and `end` is 28. Then the following coding is a translation of the design. The conditional test is complex because you cannot compare `item[lo]` with `item[hi]` unless you make sure that `lo` has not yet gone past `mid` and that `hi` has not yet gone past `end`. Study the condition so you can see why it works.

```
for (int spot = lo;  spot <= end;  spot++)
   if (lo > mid || (hi <= end
                    && item[lo].compareTo (item[hi]) > 0))
      spare[spot] = item[hi++];
   else
      spare[spot] = item[lo++];
```

For Figure 13.7, `spot` and `lo` are initially 21 and `hi` is initially 25. Compare 4 and 3. Since 3 is smaller, the 3 from `item[25]` moves into `spare[21]` and `hi` is increased to 26. Next compare 4 and 6. Since the 4 is smaller, the 4 from `item[21]` moves into `spare[22]` and `lo` is increased to 22. The rest of this example is left as an exercise.

Loop invariant for the for-loop in the merging process (any time when the loop condition `spot <= end` is evaluated): If `start` is the original value of `lo`, then all values in the `spare` array from index `start` through index `spot-1` are the `spot-start` smallest of the values that are in `item[start]` through `item[end]`, in ascending order.

The MergeSort logic

The idea behind the MergeSort is that you divide the range of values to be sorted into two exactly equal parts (or with a difference of 1 if you have an odd number of values). First you sort the first half, next you sort the second half, and finally you merge the two halves together as one long sequence in ascending order.

Why is this so much faster than the InsertionSort logic? Say the InsertionSort takes 800 milliseconds to sort 2000 items and you decide to do a MergeSort but sort the two halves with the InsertionSort logic. Then, as explained earlier, the InsertionSort will take about 200 milliseconds to sort the first 1000 items, and about 200 milliseconds to sort the second 1000 items, and the MergeSort will take about 2 milliseconds to merge them into another array in order. Total execution time is 402 milliseconds, barely more than half as long as using the InsertionSort for all the sorting. And if the MergeSort logic is used to sort each of the two halves, that speeds it up far far more.

This logic does not have the disadvantage that the QuickSort had of sometimes degenerating into a rather slow InsertionSort. But it has a different disadvantage: It requires a second array to get the job done.

The object design

The obvious object design for the MergeSort algorithm is about the same as for the QuickSort except you have to make allowance for the second (spare) array. You need the following method in CompOp, to be consistent with the other sorting algorithms:

```
public static void mergeSort (Comparable[] item, int size)
{  Comparable[] spare = new Comparable [item.length];
   new MergeSorter (item, spare).sort (0, size - 1);
}  //======================
```

How does the MergeSorter object sort the values? It first checks that it has at least two values to sort, since otherwise it does not need to take any action. If it has at least two values, it creates a MergeSorter helper to sort each half of the array. This helper should work with the same two arrays `itsItem` and `itsSpare`. Then the executor calls the merge logic discussed previously to merge the two sorted halves together. The logic apparently goes something like this:

```
int mid = (start + end) / 2;
MergeSorter helper = new MergeSorter (itsItem, itsSpare);
helper.sort (start, mid);
helper.sort (mid + 1, end);
merge (start, mid, mid + 1, end);
```

But now you have a problem. Since the executor is to leave the values in `itsItem` array, it should merge them from `itsSpare` array into `itsItem` array. That means that its helper should leave each of its sorted halves in `itsSpare` array. So some

MergeSorter objects leave the values they sort in `itsItem` and others leave them in `itsSpare`. You need some way of telling a MergeSorter object which one it is to do.

The solution, given in Listing 13.5 (see next page), is to have two sort methods, one for sorting into `itsItem` array (lines 1-8) and one for sorting into `itsSpare` array (lines 9-18). A MergeSorter object that is to sort into `itsItem` array tells its helper to sort into `itsSpare` array so it can merge the two halves back into `itsItem`. That helper will tell its own helper to sort into `itsItem` array so it can merge the two halves into `itsSpare`.

When a MergeSorter object gets back the two sorted halves, it calls the `merge` method (lines 19-24). That method's first parameter is the array the values are coming from and its second parameter is the array the values are going into. Some MergeSorter objects will pass `itsItem` to the first parameter and others will pass `itsSpare` to the first parameter.

You should study the complete coding in Listing 13.5 until you understand everything about it. It incorporates the time-saving trick discussed for the QuickSort, in which each MergeSorter object has an instance variable `itsHelper` where it stores the helper it creates. That way it does not have to create two helpers.

Faster sorting with the MergeSort

Every iteration of the loop in the `merge` method makes four tests, until one of the two halves runs out of values. It would execute faster if it could make just two tests. A nice way to do this is to sort the first half in increasing order and the second half in decreasing order. Then to merge the two halves of this "rise-then-fall" sequence, `lo` increments from `start` towards the middle and `hi` decrements from `end` towards the middle, which is just what happens in the QuickSort logic. When they meet, the loop stops. The following is the resulting `merge` method, making only two tests on each iteration:

```
private void merge (Comparable[] from, Comparable[] into,
                                  int lo, int hi)
{  int spot = lo;
   while (lo < hi)
      into[spot++] = from[lo].compareTo (from[hi]) > 0
                  ?  from[hi--]  :  from[lo++];
   into[spot] = from[lo];
}  //======================
```

Of course, now you have to also write a `mergeDown` method for merging the same kind of "rise-then-fall" sequence in descending order. And you need two additional methods `sortDown` and `sortToSpareDown` that call on the `mergeDown` method, unless you add a boolean parameter to each of `sort` and `sortToSpare` to choose the right merging method. But still, it will execute moderately faster than the simple MergeSort algorithm. This is left as a major programming project.

Another speed-up is for the `merge` method to see whether the highest value in the lower group is less than the lowest value in the upper group and, if so, skip the merging. This executes much faster for nearly-sorted lists. It is left as a major programming project.

Exercise 13.25 Write out a complete trace of the merge action for Figure 13.7.
Exercise 13.26 Revise Listing 13.5 to omit the `sortToSpare` method. Instead, pass an extra boolean parameter to `sort` that tells whether the data should end up in `itsItem` or in `itsSpare`. Is this an improvement? Why or why not?
Exercise 13.27 The MergeSort logic works faster if you handle the sorting of just two values separately and straightforwardly. Rewrite Listing 13.5 accordingly.

Listing 13.5 The MergeSort Algorithm for a partially-filled array

```
public class MergeSorter extends Object
{
   private Comparable[] itsItem;
   private Comparable[] itsSpare;
   private MergeSorter itsHelper = null;


   public MergeSorter (Comparable[] item, Comparable[] spare)
   {  super();
      itsItem = item;
      itsSpare = spare;
   }  //======================

   public void sort (int start, int end)
   {  if (start < end)                                          //1
      {  int mid = (start + end) / 2;                           //2
         if (itsHelper == null)                                 //3
            itsHelper = new MergeSorter (itsItem, itsSpare);    //4
         itsHelper.sortToSpare (start, mid);                    //5
         itsHelper.sortToSpare (mid + 1, end);                  //6
         merge(itsSpare, itsItem, start, mid, mid + 1, end);    //7
      }                                                         //8
   }  //======================

   private void sortToSpare (int start, int end)
   {  if (start >= end)                                         //9
         itsSpare[start] = itsItem[start];                      //10
      else                                                      //11
      {  int mid = (start + end) / 2;                           //12
         if (itsHelper == null)                                 //13
            itsHelper = new MergeSorter (itsItem, itsSpare);    //14
         itsHelper.sort (start, mid);                           //15
         itsHelper.sort (mid + 1, end);                         //16
         merge(itsItem, itsSpare, start, mid, mid + 1, end);    //17
      }                                                         //18
   }  //======================

   private void merge (Comparable[] from, Comparable[] into,
                       int lo, int mid, int hi, int end)
   {  for (int spot = lo;  spot <= end;  spot++)                //19
         if (lo > mid || (hi <= end                             //20
                  && from[lo].compareTo (from[hi]) > 0))        //21
            into[spot] = from[hi++];                            //22
         else                                                   //23
            into[spot] = from[lo++];                            //24
   }  //======================
}
```

Exercise 13.28* Rewrite Listing 13.5 to omit the `sortToSpare` method. Instead, have each MergeSorter object merge from `itsItem` into `itsSpare` and then copy all the sorted values back into `itsItem`.

Exercise 13.29* Essay: How much does the modification described in the previous exercise slow the execution of the algorithm? Work it out as a formula.

Exercise 13.30* Rewrite the `merge` method in Listing 13.5 to have the loop execute only as long as `lo <= mid && hi <= end`. Then clean up whatever is left after that loop.

Exercise 13.31* Rewrite Listing 13.5 to have two kinds of MergeSorter objects, one merging into `itsItem` and the other merging into `itsSpare`.
Exercise 13.32* Say that `sort` is initially called from outside the class with an array of 1024 items to sort. How many different MergeSorter objects will be created? How many would it be if the `sort` method created a local helper (as was done in the QuickSorter class) rather than having each MergeSorter object keep track of and reuse its helper?
Exercise 13.33** Rewrite the MergeSort logic to work without using recursion.
Exercise 13.34** Rewrite the QuickSort logic to work without using recursion.

13.6 More On Big-Oh

Question: How many comparisons does the MergeSort Algorithm make when there are `N` values to sort? Answer: First note that the body of the for-loop of `merge` executes exactly once for each value to be sorted. The MergeSorter object created by the `mergeSort` method gets `N` values to sort, so the if-condition in its `merge` method executes `N` times. The last time, the `compareTo` method is not evaluated, and perhaps some other times as well, so less than `N` comparisons are made.

The MergeSorter object created by the `mergeSort` method also has its helper sort each half separately, so the helper gets `N/2` values to sort the first time and the rest of the `N` values the second time, so the if-condition in its `merge` method executes less than `N` times also (though it is called twice rather than once). The helper's helper calls `merge` four times but still makes a total of less than `N` comparisons.

If say there are 32 values to sort altogether, the helper's helper compares less than 8 times on each of four calls, and the helper's helper's helper compares less than 4 times on each of eight calls, and the helper's helper's helper's helper compares 1 time on each of sixteen calls, and that is all the comparisons that are made. The total is less than `5 * 32` comparisons. In general, the total is less than `N * log2(N)` comparisons made to get `N` values sorted in ascending order, where `log2(N)` is the number of times you divide `N` by 2 to reduce it to 1.

The technical way of saying this is that **the big-oh behavior of MergeSort is N*log(N)**, where N is the number of items to be sorted. It means that the amount of time to do the job is roughly proportional to the number of items processed multiplied by the logarithm of the number of items processed, for large values of N.

If we define comps(N) to be the maximum number of comparisons the MergeSort requires to sort N values, then comps(N) can be calculated as N-1 plus the number required to sort N/2 values (one half) plus the number required to sort N-N/2 values (the other half). Since no comparisons are required to sort just 1 value, the comps function is defined by the following **recurrence equations**:

comps(N) = (N-1) + comps(N/2) + comps(N - N/2)
comps(1) = 0

A concrete example

If a particular processor requires 0.01 seconds to sort 100 items using an InsertionSort, it will take about $10{,}000^2$ times as long to sort a million items, because 1 million is 10,000 times as much and the InsertionSort algorithm is big-oh of N^2. That is 1 million seconds, which is over 11 days. Suppose it takes that same processor 0.02 seconds to sort 100 items using the MergeSort (perhaps the MergeSort is slower for so few items). Then it will still take only 600 seconds to sort a million items using the MergeSort, i.e., 10 minutes. This is calculated from 0.02 seconds = `M * 100 * log2(100)`, where the

multiplier M depends on the processor. So the time for a million items is `M * 1,000,000 * log2(1,000,000)` which is only 30,000 times as long as 0.02 seconds, to sort 10,000 times as many items. Figure 13.8 shows comparative values for N being various powers of 10.

N	N * log2(N)	N-squared
10	40	100
100	700	10,000
1,000	10,000	1 million
10,000	140,000	100 million
100,000	1.7 million	10,000 million
1 million	20 million	1 million million

Figure 13.8 The relation of N to N*log2(N) and N-squared

Clearly the MergeSort executes faster than any elementary sort for very large amounts of data. Does that mean it is more efficient? Efficiency is not just a matter of execution time. **Efficiency depends on three factors -- space, time, and programmer effort.** The MergeSort takes twice as much space (the `itsSpare` array) and a lot more programmer effort to develop. So it is better to use an elementary sort unless the amount of data is so large as to make the extra space and effort worth the savings in time.

Execution time for QuickSort

The QuickSort algorithm makes slightly more comparisons than the MergeSort even in the best case, i.e., when the pivot turns out to be right in the middle each time. Specifically, one execution of the QuickSort algorithm for N values takes N-1 comparisons to get the pivot in the right spot, then it sorts N/2 values in one half and N - N/2 - 1 values in the other half. But since the QuickSort on average only moves about half of its values around in the array, the best-case performance of QuickSort is somewhat better than MergeSort.

Statistical analysis has shown that the average-case performance of QuickSort is 1.386 times the best-case performance time (that number is twice the natural log of 2, in case you are interested in where it came from). That is 39% slower than the best-case time, so it is on average about the same as the MergeSort algorithm. However the worst-case performance of QuickSort is big-oh of N^2 and somewhat worse than InsertionSort.

Stability

Overall, the MergeSort seems best, if only because it is reliably big-oh of N*log(N). It has another advantage in that it is a stable sort. A sort is **stable** if two values that are equal end up in the same relative position that they had in the unsorted sequence. For example, if `A.compareTo(B)` is zero, and A was initially at index 22 and B was initially at index 37, then A should be at a lower index than B in the sorted array. This is always true for the MergeSort and the InsertionSort, but it is not always true for the QuickSort and the SelectionSort.

Exercise 13.35 For the processor described in this section (where sorting 100 values takes 0.01 second for InsertionSort and 0.02 seconds for MergeSort), how long would it take to sort ten million values with each method?
Exercise 13.36 Give an example to show that the SelectionSort is not stable.
Exercise 13.37* Give an example to show that the QuickSort is not stable.
Exercise 13.38* Explain why the InsertionSort in Listing 13.2 is stable.
Exercise 13.39* Explain why the MergeSort in Listing 13.5 is stable.

13.7 The HeapSort Algorithm For Comparable Objects

The HeapSort logic is to put the first `size` elements of an array named `item` in ascending order. The logic requires that you think of each component in the array as having two components as its "children". Specifically, the first component, `item[0]`, has the next two at indexes 1 and 2 as its children. Those two have the next four, at indexes 3 through 6, as their children -- 3 and 4 are the children of 1, and 5 and 6 are the children of 2. Those four have the next eight components as their children -- 7 and 8 are the children of 3, 9 and 10 are the children of 4, 11 and 12 are the children of 5, etc.

Figure 13.9 shows this relationship as a sort of genealogy tree. You start with one "node" of the tree at the top, then draw two children below it, then two children below each of those, repeating for as many elements as you have to store. Then you number the top node 0, number the next level with the next two integers, number the third level with the 4 integers after those, etc. When you look at which numbers are "children" of which other numbers, you see that it can be expressed as a formula: The **children** of index n are indexes `2*n+1` and `2*n+2`. We also say that the elements at indexes `2*n+1` and `2*n+2` are the children of the element at index n.

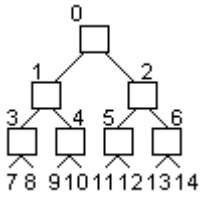


Figure 13.9 Tree

Making a max-heap

A **max-heap** is a relationship where no element is smaller than either of its two children. The first step in the HeapSort logic is to make a sufficient number of changes in the array to get the elements into a max-heap relationship. A reasonable way to do this is to go through the index values from 1 on up to `size-1`, making a max-heap out of all elements from 0 up to the index value currently under consideration. That is, you make items 0 through 1 a max-heap. Then you make items 0 through 2 a max-heap. Then you make items 0 through 3 a max-heap. Then you make items 0 through 4 a max-heap, etc.

Figure 13.10 illustrates this process for the top nine components. The values inside the components are integers, to simplify the description, though of course objects are normally used. On each iteration (reading left to right on each level), one additional component is brought into compliance by swapping it up the tree until it is greater than all of its children. (Note: A **min-heap** would have no element larger than its children).

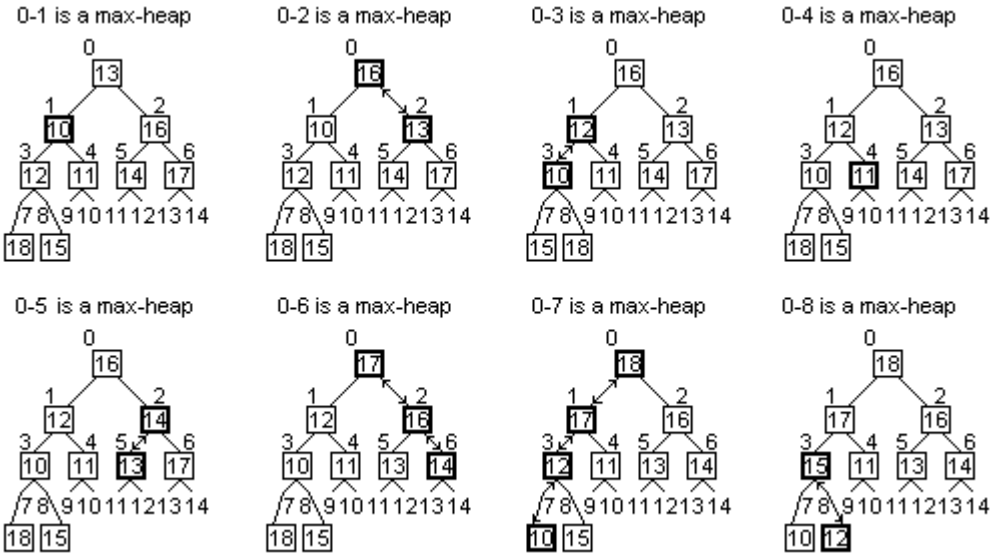


Figure 13.10 Incremental conversion to a max-heap

The coding for the `heapSort` method starts by making the entire array into a max-heap using this process. So it begins as follows:

```
public static void heapSort (Comparable[ ] item, int size)
{  for (int k = 1;  k < size;  k++)
      putInHeap (item, k);
```

The `putInHeap` auxiliary method brings items 0 through k into compliance, given that items 0 through `k-1` have previously been brought into compliance. It does this by swapping (if needed) `item[k]` with its "parent" `item[(k-1)/2]`, then swapping (if needed) that one's "parent" with it, etc. To perform this task more efficiently, it first saves `item[k]` into a Comparable variable named `save`, which leaves component k empty (actually, the value is still there; we just pretend it is only in `save`). The `empty` index is the child of index `(empty-1)/2`, so we make the assignment `parent = (empty-1)/2`. Then a loop that repeatedly copies `item[parent]` into `item[empty]`, then re-assigns `empty = parent` and `parent = (empty-1)/2`, shifts all the elements to their right places. This coding for the `putInHeap` method is in the middle part of Listing 13.6 (see next page); it is called from lines 1-2 of the public `heapSort` method.

Note the similarity with the InsertionSort logic: Each value is inserted in a sequence of values running from its current index towards zero, keeping the sequence in order. But the sequence here skips most values. For instance, when `putInHeap` is called for the element at index 127, it is inserted into the sequence of values at indexes 63, 31, 15, 7, 3, 1, and 0. Execution time for each insertion is big-oh of log(N) rather than big-oh of N. So this first stage takes a total time of big-oh of N*log(N) rather than big-oh of N^2.

Getting the values in ascending order

At this point, all of the values form a max-heap, from which it follows that `item[0]` must be the largest value of all. We want the largest value to end up in `item[size-1]`. So we swap `item[0]` with `item[size-1]`. But then what remains is not a max-heap any more, since `item[0]` is a relatively small value. We then adjust the elements in `item[0]` through `item[size-2]` to be a max-heap. To do this, swap `item[0]` with whichever of its two children is larger, then swap it with the larger of that one's two children, etc., until it ends up where it should be. Figure 13.11 illustrates this process.

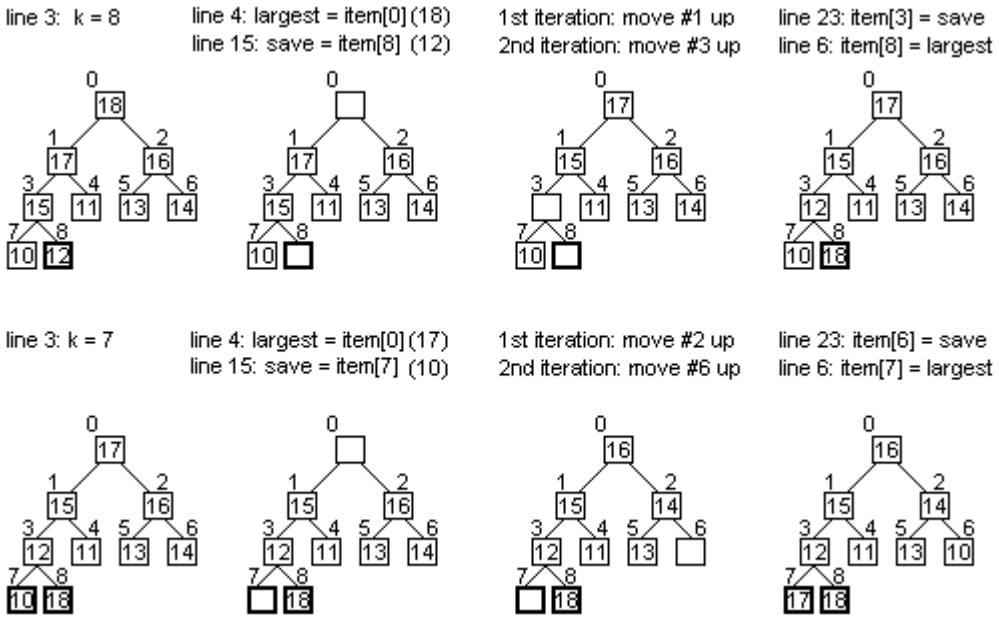


Figure 13.11 First two calls of the adjust method

Listing 13.6 The HeapSort Algorithm for a partially-filled array, in CompOp

```
/** Precondition:  size >= 0 and item[0]...item[size-1] are
 *  non-null Comparable values, all comparable to each other.
 *  Postcondition: The array contains the values it initially
 *  had but with item[0]...item[size-1] in ascending order. */

public static void heapSort (Comparable[] item, int size)
{  for (int k = 1;  k < size;  k++)                          //1
      putInHeap (item, k);                                   //2

   // now item[0]...item[size-1] form a max-heap
   for (int k = size - 1;  k > 0;  k--)                      //3
   {  Comparable largest = item[0];                          //4
      adjust (item, k); // make 0..k-1 a max-heap            //5
      item[k] = largest;                                     //6
   }
}  //======================

private static void putInHeap (Comparable[] item, int empty)
{  Comparable save = item[empty];                            //7
   int parent = (empty - 1) / 2;                             //8

   while (empty > 0 && item[parent].compareTo (save) < 0) //9
   {  item[empty] = item[parent];                            //10
      empty = parent;                                        //11
      parent = (empty - 1) / 2;                              //12
   }                                                         //13
   item[empty] = save;                                       //14
}  //======================

private static void adjust (Comparable[] item, int end)
{  Comparable save = item[end];                              //15
   int empty = 0;                                            //16
   int child = bigKid (item, 1, end);                        //17

   while (child < end && item[child].compareTo (save) > 0)
   {  item[empty] = item[child];                             //19
      empty = child;                                         //20
      child = bigKid (item, empty * 2 + 1, end);             //21
   }                                                         //22
   item[empty] = save;                                       //23
}  //======================

private static int bigKid (Comparable[] item, int n, int end)
{  return n + 1 < end                                        //24
             && item[n + 1].compareTo (item[n]) > 0          //25
          ?  n + 1  :  n;                                    //26
}  //======================
```

At this point `item[0]` through `item[size-2]` is a max-heap, so we can repeat the whole process: swap the second-largest element of all (now at `item[0]`) with the element at `item[size-2]`, so you have the two largest elements in the places they should be. Then adjust the rest of the array to be a max-heap again. Repeat as needed.

Lines 3-6 of the `heapSort` method repeatedly take the largest of the as-yet-unsorted elements from `item[0]`, adjust the rest of the array (indexes 0 through `k`) to make room for it at the end, then put that largest element in component `k`. this is done for each value of `k` from `size-1` down to 1. The `adjust` method names the index parameter `end`, because it is the `end` component that is to receive the largest value. The `adjust` method puts the item at the end into a Comparable variable named `save` and notes that the component at index 0 is now `empty` (lines 15-16). It computes the larger child of the empty component (lines 24-26, allowing for the possibility that the component only has one child). Then as long as the `child` value is within the bounds 1 to `end-1`, and as long as the element at index `child` is larger than the `save` value, it shifts the `child` value up to the `empty` component and calculates the `child` of that newly-emptied component. This coding is in lines 18-23 of Listing 13.6.

Note the similarity with the SelectionSort logic: We select the largest of the remaining values and swap it with the element in the component where that largest value goes. But the process of selecting the largest executes in big-oh of log(N) time rather than in big-oh of N time. The reason is that the adjusting of the array to restore the max-heap condition jumps through the index values, doubling the size of the jump at each iteration. By contrast, the SelectionSort has to go through every single value.

Comparison with other sorting methods

As the preceding logic shows, the HeapSort requires big-oh of N*log(N) time, even in the worst case. This is far better worst-case performance than the QuickSort logic, which can sometimes degenerate to big-oh of N^2 execution time. Of course, the MergeSort also has big-oh of N*log(N) worst-case execution time, but it requires an extra array for storage, which doubles the storage requirements. The HeapSort does not require any significant extra storage (two or three variables for `save`, `empty`, etc.).

On the other hand, the HeapSort is the most complex of these three sorting methods, and it is the slowest in terms of average execution time.

Exercise 13.40 The `heapSort` logic will execute faster if it only calls the two private methods when the second parameter is 2 or more. Make the modifications to do this.
Exercise 13.41 Count the maximum number of possible comparisons of elements that the HeapSort makes when `size` is 3, then count them when `size` is 7.
Exercise 13.42* Show that, if you make a max-heap and then perform an InsertionSort on it, you still have big-oh of N^2 execution time for the worst case. Hint: What is the worst case for the order of the elements in the lower half of the array?
Exercise 13.43* An alternate version of the `heapSort` has the first stage make `item[k]` through `item[size-1]` a max-heap for values of `k` decreasing from `size/2` to 0. Make the modifications in Listing 13.6 to do this.

13.8 Data Flow Diagrams

A small manufacturing company wants to hire you to create software to automate some of their operations. A scenario of part of what this company does could go as follows:

A customer places an order. You verify that the information about the customer (address, references, preferences, etc.) is already in your files. You check with your accounting department that the customer's credit is good. You record the order, manufacture the product wanted, and ship it to the customer along with an invoice. The customer sends you a check in payment, which you use to pay investors' dividends.

The preceding paragraph is called a **use case**; it is a scenario that describes a possible sequence of events using the system you are to design. If you are to model the entire business in software, then the business is the system and everything outside the system is its **environment**. Figure 13.12 is a Data Flow Diagram that graphically shows the elements mentioned in this particular use case.

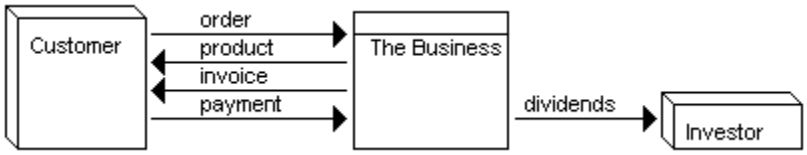


Figure 13.12 Data Flow Diagram for one use case

A **Data Flow Diagram** (**DFD**) has four basic kinds of elements, the first three called the **nodes** of the diagram:

1. A **process**, which is part of the system to be modeled, shown as a rectangle divided horizontally. You put the name and/or description of the system part in the lower part.
2. An **environment entity** that interacts with the system, shown as a box-like figure. You put the name of the entity on the front of the box.
3. A **data store**, which is part of the system to be modeled, shown as a rectangle divided vertically and with the right side containing its name: It stores a collection of information of a particular kind. No data stores appear in Figure 13.12.
4. A **data flow**, shown as an arrow between two nodes, of which one is almost always a process. A data flow indicates information flowing from one node to another. Its name is above or beside the arrow.

Some other use cases for the manufacturing company are as follows:

You check the amount you have on hand of various parts you use to manufacture your product. You see that you are short on one kind of part. You place an order with a supplier. The supplier sends you boxes of parts and an invoice. You send a check to pay the invoice.

Investors contribute more capital to the operation, for which you issue stock. Every three months, you calculate your profits. Based on these calculations, you send a tax payment to the IRS and also dividend checks and a quarterly report to your investors.

When you look over these and other use cases, including entering a description of the customer (address, etc.) in your files, you might come up with the Data Flow Diagram in Figure 13.13 to describe the major interactions of the company with its environment.

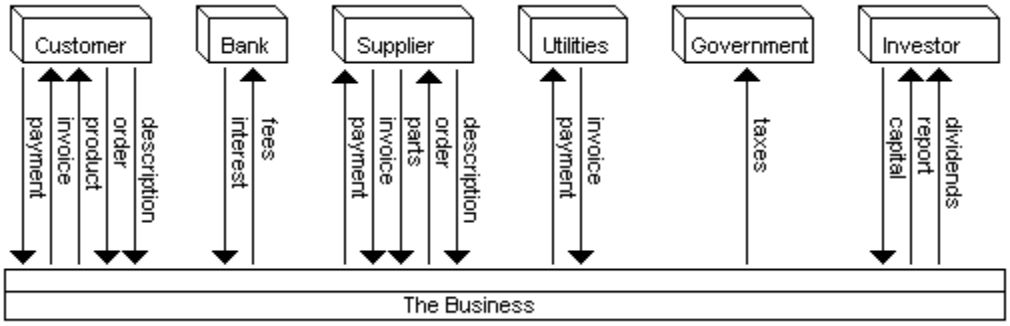


Figure 13.13 Data Flow Diagram for the entire business

The OrderHandling system

Your client wants you to implement in software only the part of the operation that handles orders from customers and orders to suppliers. So you do not need all of the data flows described so far, such as payment of utility bills and taxes. Start with some examples of use cases for the OrderHandling system:

Use Case 1 A person wants to order products from your company. You look up the name in your records and see that the customer is not recorded there. So you get the full description of the customer (name, address, references, etc.) and store it in your data store of customers. You then take the order and store it in your data store of orders. You check your inventory and see that the product is there. So you ship the product to the customer along with an invoice for payment, and at the same time notify the accounting department to expect payment since you have sent an invoice.

Use Case 2 A person places an order. You look up the name in your records and retrieve the customer's information from your customer data store (since it was previously recorded). You then take the order and store it in your data store of orders. You check inventory and see that the product is not there. So you order it from the appropriate supplier (which in many cases will be a manufacturing department in the rest of the business, but treat it as a supplier for now).

Use Case 3 A supplier ships parts to you along with an invoice. You store it in your inventory and notify the accounting department to send payment to the supplier.

Use Case 4 A manufacturing department sends you finished product. You store it in your inventory.

Use Case 5 You check your data store of orders not yet filled and find one for which you now have the required product. So you ship the product to the customer along with an invoice for payment, and at the same time notify the accounting department to expect payment since you have sent an invoice.

Further thought about the software and discussion with the client revealed that the customer will sometimes ask for the status of all of the customer's orders currently being processed, to verify its own records. In addition, the rest of the business operation may query the system from time to time for a list of all orders pending, of all orders filled in the past year, of inventory changes during the year, etc.

A very effective first step in the development of a software system is to describe its interactions with its environment, that is, with everything that is not the system to be developed. From this point of view, the rest of the business is part of the environment. These interactions usually take the form of input to the system from external entities and output from the system to external entities, both of which can be expressed as data flows.

Figure 13.14 is the only part of the overall operation of the company that concerns you. Note that the system is split off from the rest of the business in this Data Flow Diagram. The names of the data flows have been made more precise: CustDesc is a description of one customer, SuppOrder is one order sent to a supplier, etc. The reporting functions have been added to the Data Flow Diagram. Minor "backflows" along certain arrows have been omitted, such as a confirmation message to the customer that the description or order has been received and a query for a report.

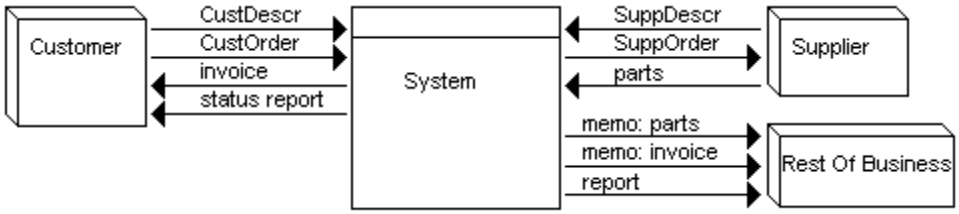


Figure 13.14 Data Flow Diagram for the OrderHandling software

Detailed Analysis

The next step in the analysis is to specify the exact form of the inputs and the outputs, the conditions under which they will occur, the sequence in which they occur, etc. This information fleshes out the DFD by saying what happens. Avoid saying how it happens; that is almost always a matter of design. Treat the system as a "black box" whose contents and workings are a mystery.

Analysis for this problem requires a full and precise description of the number and types of values entered, e.g., first and last name are strings of characters, the ZIP code is entered separately from the city and state, and the number of items ordered is a positive integer. It also requires a full and precise description of the form of the reports. This **Requirements Specification Document** should include examples and specifications for these things along with the Data Flow Diagram. Test data should also be developed at this point. We will not try to do here for this problem.

Design

Now you start to divide up the proposed system into components. Design is a matter of deciding which components you want to have and how they will interact. For this OrderHandling software, some obvious choices are classes of CustomerOrder objects, SupplierOrder objects, Invoice objects, CustomerDescription objects, and SupplierDescription objects. You will also need a data structure for long-term storage of the list of CustomerDescription objects for all current customers, the list of CustomerOrder objects still pending, and the analogous lists of SupplierDescription and SupplierOrder objects. Reports are presumably simply long Strings of characters, and you already have a String object class.

The two main processes of the system are managing the customer relations and managing the supplier relations. These considerations lead to the Data Flow Diagram shown in Figure 13.15. The abbreviations CustOrder, SuppOrder, CustDescr, and SuppDescr make the diagram more compact. Process nodes are numbered 1, 2, 3, on up in no particular order. Data store nodes are numbered D1, D2, D3, on up in no particular order; we often append DS to the name to suggest a data store (also known as a data structure). These numbers are used for cross-referencing with more detailed Data Flow Diagrams and with narrative specifications.

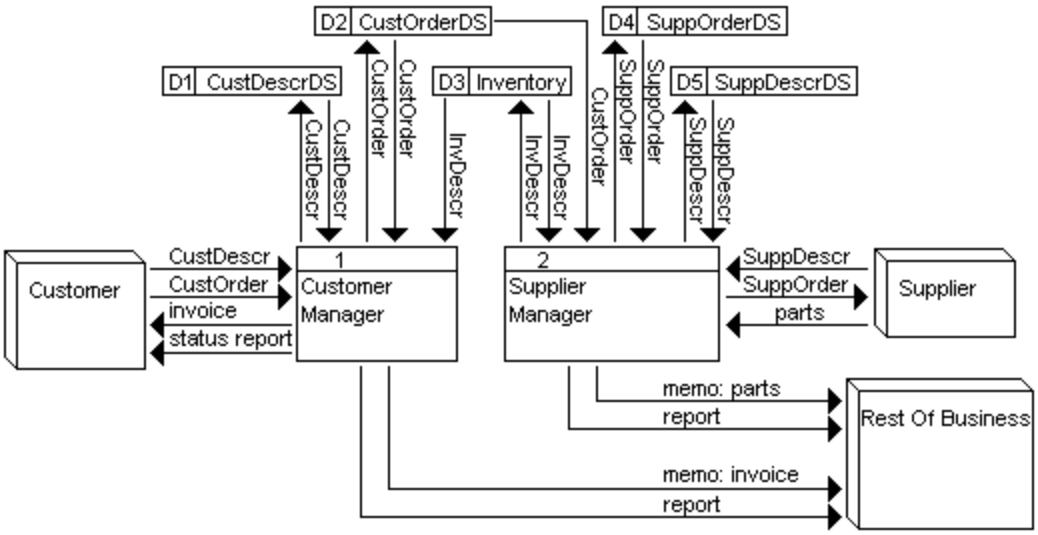


Figure 13.15 Data Flow Diagram for second level of OrderHandling

The middle and upper parts of Figure 13.15 are an **explosion** of the System node in Figure 13.14. You need to verify that all the data flows between the System node and its environment in Figure 13.14 are in the exploded view, and no others have been added. A close check shows that the ten data flows of the earlier figure are present in the later one, although "report" occurs twice in the later one (since the two separate processes have separate kinds of reports).

CRC cards

A useful device for a preliminary specification is a Class/Responsibilities/Collaborations card, a **CRC** card for short. You use an index card (no bigger than 4-by-6) on which you write the name of the class of objects at the top. Then you list on the left side of the card the major responsibilities of that class -- what messages it can be sent and what action it should take in response. You list on the right side of the card the collaborations of that class -- which other classes it send messages to or otherwise uses.

The CustomerManager kind of object (process #1) has four major responsibilities:

1. Given a CustomerOrder object containing a CustomerID, see whether the CustomerDescription object is in the CustDescrDS. If not, create one from from data the customer provides and store it in the CustDescrDS. Then store the CustomerOrder object in the CustOrderDS.
2. Search the CustOrderDS for orders not yet filled. For each one, check whether all products ordered are currently in the Inventory. If so, create an Invoice object, send it to the customer along with the product ordered, and send a memo of the Invoice to the accounts-receivable department.
3. Given a request from the customer for a status report, create and return it.
4. Given a request from the rest of the business for any of several kinds of summary reports, create and return it.

This process #1 has therefore collaborations with three data stores (CustDescrDS, CustOrderDS, and Inventory), with three passive data flows (CustomerOrder, CustomerDescription, and Inventory objects) and with the rest of the business. Now stubbed documentation for the CustomerManager class of objects can be developed as shown in Listing 13.7.

Listing 13.7 CustomerManager class of objects, stubbed documentation

```
public class CustomerManager extends Object          // stubbed
{
   /** Verify that the customer's description is on file.
    *  If not, obtain and file it.  Then file the order.  */

   public void storeOrder (CustomerOrder order)              { }

   /** Find all orders that are currently unfilled.
    *  Fill any such order for which all product is available. */

   public void fillAllPossibleOrders()                       { }

   /** Return a description of all orders from this customer
    *  that are currently unfilled. */

   public String getStatus (String customerID)   { return null; }

   /** Return a description of all orders currently unfilled,
    *  in order of customer ID. */

   public String getOrdersByID()                 { return null; }

   /** Return a description of all items currently unfilled,
    *  in order of dates entered. */

   public String getOrdersByDate()               { return null; }

   // many other report functions would go here
}
```

You develop CRC cards by walking through the scenarios given by use cases and seeing which classes and operations a software system should have. A large number of people find that developing CRC cards is a far better way to develop an object design than data flow diagrams, since they have a concrete object (the 4-by-6 card) to help them anchor their thoughts. Other people prefer developing data flow diagrams.

*13.9 About The Arrays Utilities Class (*Sun Library)*

The **Arrays** class in the `java.util` package has 55 methods that can be useful at times. They are named `equals`, `sort`, `fill`, `binarySearch`, and `asList`.

- `Arrays.equals(someArray, anotherArray)` returns a boolean that tells whether the two arrays have the same values in the same order. There is one such method for each of the 8 primitive types (e.g., two arrays of ints) and one for two arrays of Objects.
- `Arrays.sort(someArray)` sorts in ascending numeric order an array of byte, char, double, float, int, short, or long values (thus 7 such methods). It uses a modification of the QuickSort algorithm that maintains N*log(N) performance on many sequences of data that would cause the simple QuickSort to degenerate into N^2 performance.
- `Arrays.sort(someArray, startInt, toInt)` is the same except it only sorts the values in the indexes from `startInt` up to but not including `toInt`. As usual, it requires `0 <= startInt <= toInt <= someArray.length`.
- `Arrays.sort(someArray)` sorts in ascending order an array of Comparable objects with `compareTo`. It uses the MergeSort algorithm, thus it is a stable sort.
- `Arrays.sort(someArrayOfObjects, startInt, toInt)` is the same for an array of Comparable objects except it only sorts the values in the indexes from `startInt` up to but not including `toInt`.
- `Arrays.fill(someArray, someValue)` assigns `someValue` to each component of the array. There is one such method for each of the 8 primitive types (e.g., an array of longs with a long `someValue`) and one for Objects.
- `Arrays.fill(someArray, startInt, toInt, someValue)` is the same except it only assigns components from `startInt` up to but not including `toInt`.
- `Arrays.binarySearch(someArray, someValue)` returns the int index of `someValue` in the array using binary search. The array should be sorted. There is one such method for each of the 7 numeric types (e.g., an array of doubles with a double `someValue`) and one for Objects (using `compareTo`). If `someValue` is not found, then the value returned is negative: `-(n+1)` where n is the index where you would insert `someValue` to keep all the values in order.
- `Arrays.asList(someArrayOfObjects)` returns a List object "backed by" the array. This List cannot be modified in size, and each change to the List is reflected immediately in the array itself (`set` is supported but `add` and `remove` are not). The List interface is described in Chapter Fifteen.
- Three additional Arrays methods are analogous to the three Object methods above that use `compareTo`, but they have an additional Comparator parameter.

The **Comparator** interface in the `java.util` package specifies two methods that can be used for Objects instead of the "natural" `compareTo` and `equals` methods:

- `someComparator.compare(someObject, anotherObject)` returns an int value with the same meaning as `compareTo`, but the comparison is normally based on some other criterion than what `compareTo` uses.
- `someComparator.equals(someObject, anotherObject)` returns a boolean telling whether the Comparator's `compare` method returns zero, or both objects are `null`.

13.10 Review Of Chapter Thirteen

About the Java language:

- For any int variable k, if you use the expression k++ or k-- in a statement, it has the value that k had before it was incremented or decremented. When the variable is after the ++ or -- operator, it has the final value after incrementing or decrementing. For instance, if k is 4, then `item[k++]=2` assigns 2 to `item[4]` and changes k to 5, but `item[++k]=2` assigns 2 to `item[5]` and changes k to 5. If a variable appears twice in the same statement, and one appearance has the ++ or -- operator on it, the effect is too hard to keep straight, so do not do that.
- The source code for a method X can contain a call of X or a call of a method that (eventually) calls X. This is **recursion**. At run time, each execution of a method call generates an **activation** of that method. During execution of that activation, additional method calls generate complete different activations. Recursion means that two different activations can be executing the same logic sequence.
- A recursive method cannot loop forever if it has a **recursion-control variable**, either explicitly declared or implicit in the logic. A recursion-control variable for method X is a variable with a **cutoff** amount which satisfies (a) each call of X from within X requires that the recursion-control variable for the current activation is greater than the cutoff, (b) each call of X from within X passes a value of the recursion-control variable to the new activation that is at least 1 smaller than the existing activation has.

About some classic algorithms:

- The **SelectionSort** puts a list of two or more values in order by selecting the smallest, putting it first, then applying the SelectionSort process to the rest of the list.
- The **InsertionSort** puts a list of two or more values in order by applying the InsertionSort process to the sublist consisting of all but the last value, then inserting that last value where it goes.
- The **BinarySearch** algorithm finds a target value in an ordered list of two or more values by comparing the target value with the middle value of the list to decide which half could contain the target value, then applying the BinarySearch process to find the target value in that half of the list.
- The **QuickSort** puts a list of two or more two or more values in order by choosing one element, dividing the list in the two sublists of those larger and those smaller than the chosen element, then applying the QuickSort process to each of the two sublists.
- The **MergeSort** puts a list of two or more values in order by dividing it in half, applying the MergeSort process to each half, and then merging the two sorted halves together in order.
- The **HeapSort** is a more complex kind of sorting algorithm discussed in this chapter.
- Average execution time for the InsertionSort and SelectionSort is roughly proportional to N^2 where `N` is the number of values to sort. Sorting algorithms with this property are called **elementary sorts**.
- Average execution time for the BinarySearch algorithm is roughly proportional to log2(N). In this context, **log2(N)** is the lowest power you put on 2 to get a number not smaller than N. In particular, log2(N) is 6 for any number in the range from 33 to 64.
- Average execution time for the MergeSort, QuickSort, and HeapSort is roughly proportional to `N*log2(N)`. This describes the **big-oh behavior** of these algorithms. Figure 13.16 compares the sorting algorithms for efficiency.
- A sort is **stable** if, of two values that are equal, the one that came earlier in the unsorted list ends up earlier in the sorted list. The InsertionSort and MergeSort are stable sorts; the other three discussed in this chapter are not.

	worst-case time	average-case time	extra space	programmer effort
SelectionSort	N-squared	N-squared	1	low
InsertionSort	N-squared	N-squared	1	low
QuickSort	N-squared	N * log(N) fast	1	medium
MergeSort	N * log(N)	N * log(N) fast	N	medium
HeapSort	N * log(N)	N * log(N) slow	1	high

Figure 13.16 Efficiency of five sorting algorithms

Answers to Selected Exercises

13.1 Replace the loop condition k < size - 1 by item[k + 1] != null and replace the loop condition k <= end by item[k] != null.

13.2 Replace the phrase "< 0" by the phrase "> 0" and the word "Min" by the word "Max".

13.5
```
private static void insertInOrder (Comparable[ ] item, int n)
{    if (item[n].compareTo (item[n - 1]) < 0)
     {    Comparable save = item[n];
          item[n] = item[n - 1];
          int k = n - 1;
          for (;  ...  // all the rest as shown in Listing 13.2
     }
}
```
This executes faster only because item[n - 1] is moved up before the loop begins, so that the loop has one less iteration than in the original logic. Otherwise, if you made the comparison of item[n] with item[n - 1] twice, it would usually execute more slowly (the probability is quite low that item[n] is larger than the one before, once n becomes fairly large).

13.6 Replace the loop condition k < size by item[k] != null.

13.7 Replace "Comparable" by "double" in three places. Replace the second for-loop condition by: k > 0 && item[k - 1] > save

13.8 Insert the following as the first statement of insertInOrder:
```
if (item[m] == null)
     return;
```
Change the condition in the for-loop to be the following:
m > 0 && (item[m - 1] == null || item[m - 1].compareTo (save) > 0)

13.10 If the range lowerBound...upperBound has an odd number of values, say 13, then upperBound is lowerBound+12, so midPoint is lowerBound + 6, which divides the range into the 6 values above midPoint and the 7 values up through midPoint. In general, the front half is 1 larger when there are an odd number of values to search (upperBound - lowerBound is an even number).

13.11 If size is in fact 0, then the body of the loop will not execute. item[lowerBound] will be item[0], which might throw a NullPointerException or an ArrayIndexOutOfBoundsException

13.12 If there are an even number of values to search, at least 4, then rounding the other way would make the lower part 2 larger than the upper part; the imbalance would only slow the convergence somewhat. If however upperBound is lowerBound + 1, thus 2 values to search, then rounding up would make midPoint equal to upperBound. If item[midPoint] is then smaller than target, lowerBound could be greater than size-1, which might throw a NullPointerException or an ArrayIndexOutOfBoundsException. If item[midPoint] is not smaller than target, you would have an infinite loop.

13.17 pivot is 7, leaving **X**, 2, 9, 1, 5, 8, **4**. 4<7 moves the 4 to the X, leaving 4, **2**, 9, 1, 5, 8, **X**. 2<7 stays where it is. 9>7 moves the 9 to the X, leaving 4, 2, **X**, 1, 5, **8**, 9. 8>7 stays where it is. 5<7 moves the 5 to the X, leaving 4, 2, 5, **1**, **X**, 8, 9. 1<7 stays where it is. 7 replaces the X.

13.18 Replace the first two statements by Comparable pivot = itsItem[hi]; boolean lookHigh = false;

13.19 Replace the last three statements in the sort method by the following:
```
if (itsHelper == null)
     itsHelper = new QuickSorter (itsItem);
itsHelper.sort (start, mid - 1);
itsHelper.sort (mid + 1, end);
```

13.20 At the first evaluation of the while-condition lo < hi, lo equals start (since start was passed in as the initial value of the parameter lo, so there are NO values from start through lo-1, so all of them are (vacuously) less than the pivot. similarly, hi equals end, so there are no values from hi+1 through end, so all of those values are (vacuously) greater than the pivot. Finally, lookHigh is true and itsItem[lo] is where the pivot came from, so itsItem[lo] is "empty".

13.25 Initially we have (lo)4, 7, 12, 17, (hi)3, 6, 9, 14. Since 4 is larger than 3, move 3 down, so we now have (lo)4, 7, 12, 17, X, (hi)6, 9, 14. Since 4 is smaller than 6, move 4 down, so we now have X, (lo)7, 12, 17, X, (hi)6, 9, 14. Since 7 is larger than 6, move 6 down so we now have X, (lo)7, 12, 17, X, X, (hi)9, 14. Since 7 is smaller than 9, move 7 down so we now have X, X, (lo)12, 17, X, X, (hi)9, 14. Since 12 is larger than 9, move 9 down so we now have X, X, (lo)12, 17, X, X, X, (hi)14. Since 12 is smaller than 14, move 12 down so we now have X, X, X, (lo)17, X, X, X, (hi)14. Since 17 is larger than 14, move 14 down, then 17.

13.26 The sort method is revised as follows:

```
public void sort (int start, int end, boolean toSpare)
{   if (start >= end)
    {   if (toSpare)
            itsSpare[start] = itsItem[start];
    }
    else
    {   int mid = (start + end) / 2;
        MergeSorter helper = new MergeSorter (this);
        helper.sort (start, mid, ! toSpare);
        helper.sort (mid + 1, end, ! toSpare);
        if (toSpare)
            merge (itsItem, itsSpare, start, mid, mid + 1, end);
        else
            merge (itsSpare, itsItem, start, mid, mid + 1, end);
    }
}
```

This is worse. The extra tests make this execute slower, and it seems harder to understand.

13.27 Add the following at the beginning of the body of the sort method:

```
if (start + 1 == end)
{   if (itsItem[start].compareTo (itsItem[end]) < 0)
    {   Comparable save = itsItem[start];
        itsItem[start] = itsItem[end];
        itsItem[end] = save;
    }
}else
```

Add the following at the beginning of the sortToSpare method:

```
if (start + 1 == end)
{   if (itsItem[start].compareTo (itsItem[end]) < 0)
    {   itsSpare[end] = itsItem[start];
        itsSpare[start] = itsItem[end];
    }
    else
    {   itsSpare[start] = itsItem[start];
        itsSpare[end] = itsItem[end];
    }
}else
```

13.35 For the InsertionSort, multiply the 11 days by the square of 10, thus 1100 days. For the MergeSort, the time is M * 10million * log2(10million). The second factor is 10 times what it was for a million items, and the third factor is 24/20 times what it was for a million items, so it takes 12 times 10 minutes, thus 2 hours.

13.36 For the sequence 5, 5, 3, 8, the first pass swaps the 3 with the first 5, so those two 5's are not in their original relative position. Other passes do not change anything.

13.40 In the heapSort method, have the first for-loop initialize int k = 2 and have the second for-loop perform the test k > 1. Put the following statements before the first for-loop:

```
if (size >= 2 && item[1].compareTo (item[0]) < 0)
{   Comparable save = item[1];
    item[1] = item[0];
    item[0] = save;
}
```

Also put those last three swapping statements at the end of the heapSort coding.

13.41 3 when size is 3: 2 for putInHeap, 1 for adjust. 24 when size is 7: 2*1+4*2 for the two levels of putInHeap, then (4*4-3)+1 for adjust.

14 Stacks And Queues

Overview

This chapter requires that you have a solid understanding of arrays (Chapter Seven) and have studied Exceptions (Sections 10.1-10.3) and interfaces (Sections 11.1-11.3).

- Section 14.1 discusses situations in which stack and queues can be useful.
- Section 14.2 presents implementations of stacks and queues using arrays.
- Section 14.3 introduces the concept of linked lists and use them to implement stacks and queues.
- Sections 14.4-14.7 define priority queues and give array and linked list implementations of them.
- Sections 14.8-14.9 relate priority queues to the discussion of sorting and big-oh performance begun in Chapter Thirteen and present an algorithm for sorting sequential files. Postpone these sections if you skipped Chapter Thirteen.

This chapter kills three birds with one stone: You develop a strong understanding of the stacks and queues, you strengthen your abilities in working with arrays, and you develop a moderate facility with linked lists.

14.1 Applications Of Stacks And Queues

Your in-basket on your desk at work contains various jobs for you to do in the near future. Each time another job comes in, you put it on top of the pile. When you finish one job, you take another job from the pile to work on. Which one do you take?

You can model this situation in software with a **data structure**, an object that contains several elements, allows you to add elements to it, and allows you to remove an element from it when you wish.

- If you always take the job on top, you are following a **Last-In-First-Out** algorithm (LIFO for short): Always take the job that has been in the pile for the shortest period of time. A data structure that implements this algorithm is called a **stack**. However, this algorithm may not be appropriate for a job-pile; some jobs may sit on the bottom of the stack for days.
- If you always take the job on the bottom, you are following a **First-In-First-Out** algorithm (FIFO for short): Always take the job that has been in the pile for the longest period of time. A data structure that implements this algorithm is called a **queue**. This algorithm guarantees that jobs do not sit overly long on the job-pile, but you may end up postponing very important jobs while working on trivial jobs that came in earlier.
- If you always take the job that has the highest priority, you are following a **Highest-Priority** algorithm: Always take the job in the pile that has the highest priority rating (according to whatever priority criterion you feel gives you the best chance of not getting grief from your boss). A data structure that implements this algorithm is called a **priority queue**.

Another example where a queue would be important is in storing information about people who are waiting in line to buy tickets. Each new arrival goes to the rear of the queue (which we call the `enqueue` operation), and the next ticket sold goes to the person at the front of the queue (produced by the `dequeue` operation).

If you are selling movie tickets, an ordinary queue is appropriate. But if they are tickets to fly on an airline, a priority queue might be better: You would want to serve people whose plane will be taking off within the hour before people who have longer to wait, and you would want to serve first-class customers before customers buying the cheap tickets.

Stock market application of stacks

An investor buys and sells shares of stock in various companies listed in the stock market. What if the person sells 120 shares of a particular company for $90 each at a time when the person owns 150 shares, of which 50 shares were bought in January at $60 each, 50 more in February at $70 each, and 50 more in March at $80 each? Which of those particular shares are being sold?

The federal tax laws require this investor to compute the profit and loss on a LIFO basis, i.e., the most recently purchased shares are sold first. So the profit on the sale is $500 on the 50 purchased in March, $1000 on the 50 purchased in February, and another $600 (20 shares at $30 each) on some shares purchased in January. Therefore, an appropriate data structure for this situation is one stack of stock transactions for each of the companies that the person owns stock in.

For this particular example of selling 120 shares, the software would remove from the top of the stack the first three transactions (which we call the `pop` operation) and then add back the net transaction of 30 shares remaining at $60 each (which we call the `push` operation).

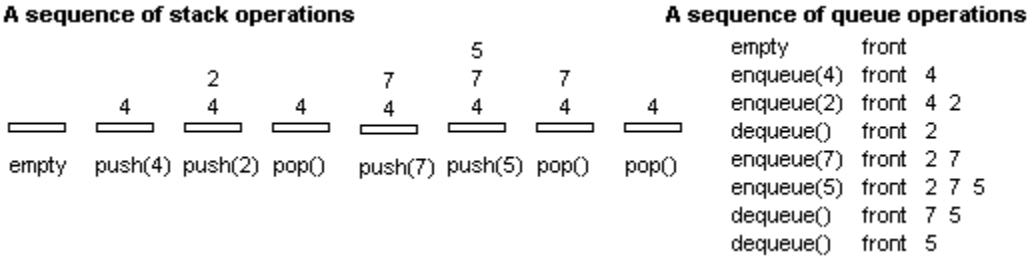


Figure 14.1 Effect of method calls for stacks and queues

Software to update the current stock holdings for the investor could be based on Transaction objects, each storing information about one stock transaction: company name, number of shares, price per share, and date of purchase. A text file could hold several lines of characters, each line giving the description of one transaction (a purchase of stock). The investor could start the program (which reads in the text file), enter several new buy-and-sell transactions, and then exit the program (which writes the updated information to a text file). A more detailed design is given in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for stock transactions

1. Read the "transact.dat" file into an array of stacks, one stack per company.
2. Repeat as long as the user has more transactions to process...
   - 2a. Get the next transaction from the user (buy or sell).
   - 2b. Modify the information in the array of stacks accordingly.
3. Write the updated information to a file.

Some other applications of stacks

You may need a program that reads in several lines of input, each containing what is supposed to be a properly-formed expression in algebra, and tells whether it is in fact legal. The expression could have several sets of grouping symbols of various kinds, `( )` and `[ ]` and `{ }`. Part of the parsing process is to make sure that these grouping symbols match up properly. The accompanying design block could be part of the logic for processing the algebraic expression.

STRUCTURED NATURAL LANGUAGE DESIGN for grouping symbols

1. Get the next character on the input line; call it the `currentChar`.
2. If the `currentChar` is one of '(', '[', or '{', then...
   Push the `currentChar` on the stack.
3. Otherwise, if the `currentChar` is one of ')', ']', or '}', then...
   3a. If the stack is empty then...
   Conclude that this is not a properly-formed expression.
   3b. Pop the top character from the stack.
   3c. If that is not the other half of the `currentChar`, then...
   Conclude that this is not a properly-formed expression.
4. Otherwise [additional logic for other characters]...

A Java compiler uses a stack to handle method calls, which works roughly as follows (very roughly), assuming `calls` is the stack of method calls:

- `x = doStuff(y)` causes `calls.push (new Activation (arguments));`
- `z = 3` for a local variable `z` causes `calls.peekTop().setVariable(z, 3);`
- `return 3` causes `calls.pop(); x = 3.`

Stack and queue interfaces

We will develop several different implementations of stacks and queues in this chapter. We specify what is common to all by defining an interface for each. The interface describes what operations the object can perform but does not give the coding. So it describes an **abstract data type**. So we call these two interfaces **StackADT** and **QueueADT**. They are shown in Listing 14.1 (see next page). Each has an operation to add an element to the data structure (`push` and `enqueue`), an operation to remove an element (`pop` and `dequeue`, which return the element removed), and two query methods: an operation to see what would be removed (`peekTop` and `peekFront`), and an operation to tell whether the data structure has any more elements to remove (`isEmpty`).

As an example of how these methods can be used, the following is an independent method that reverses the order of the elements on a given stack, using a queue for temporary storage. Suppose that ArrayQueue is an implementation of QueueADT. The coding removes each value from the stack and puts it on the queue, with what was on top of the stack going to the front of the queue. Then it removes each value from the queue and puts it on the stack. The value that was originally on top of the stack therefore ends up on the bottom of the stack:

```
public static void reverse (StackADT stack)
{  QueueADT queue = new ArrayQueue();
   while ( ! stack.isEmpty())
      queue.enqueue (stack.pop());
   while ( ! queue.isEmpty())
      stack.push (queue.dequeue());
}  //=====================
```

Listing 14.1 The StackADT and QueueADT interfaces

```
public interface StackADT            // not in the Sun library
{
   /** Tell whether the stack has no more elements. */

   public boolean isEmpty();

   /** Return the value that pop would give, without modifying
    *  the stack.  Throw an Exception if the stack is empty. */

   public Object peekTop();

   /** Remove and return the value that has been in the stack the
    *  least time.  Throw an Exception if the stack is empty. */

   public Object pop();

   /** Add the given value to the stack. */

   public void push (Object ob);
}
//#############################################################


public interface QueueADT            // not in the Sun library
{
   /** Tell whether the queue has no more elements. */

   public boolean isEmpty();

   /** Return the value that dequeue would give without modifying
    *  the queue.  Throw an Exception if the queue is empty. */

   public Object peekFront();

   /** Remove and return the value that has been in the queue the
    *  most time.  Throw an Exception if the queue is empty. */

   public Object dequeue();

   /** Add the given value to the queue. */

   public void enqueue (Object ob);
}
```

Anagrams

An interesting problem is to print out all the anagrams of a given word. That is, given an N-letter word with all letters different, print all the "words" you can form using each letter once, regardless of whether they are actual words in some language. The number of such rearrangements is N-factorial where N is the number of letters. The problem can be solved in several different ways, one of which uses one queue and one stack. We illustrate the process here and leave the coding as a major programming project.

Say the word is abcdefgh and you have just printed ehcgfdba. The very next word in alphabetical order that you can form with those eight letters is ehdabcfg (compare the two

to see why). The way one iteration of the main loop of the process goes from a stack containing **ehcgfdba** to that same stack containing **ehdabcfg** is as follows:

Initially you have the letters on a stack in reverse order (a on top): **stack= abdfgche**
1. Move one value from the stack to a variable x: **stack= bdfgche x= a**
2. Repeatedly move values from x to a queue and then from the stack to x until the one you enqueue comes before x. Here we enqueue a and pop b into x, then enqueue b and pop d into x,..., enqueue g and pop c into x: **stack= he x= c queue= abdfg**
3. Repeatedly move one value from the front of the queue to the rear of the queue until the front of the queue comes after x: **stack= he x= c queue= dfgab**
4. Move one value from the queue to the stack: **stack= dhe x= c queue= fgab**
5. Move one value from x to the queue: **stack= dhe x= c queue= fgabc**
6. Repeatedly move values from the front of the queue to the rear of the queue until the front of the queue is smaller than x: **stack= dhe x= c queue= abcfg**
7. Move all values from the queue to the stack: **stack= gfcbadhe**

Exercise 14.1 Write an independent method `public static Object removeSecond (StackADT stack)`: It removes and returns the element just below the top element if the stack has at least two elements, otherwise it simply returns `null` without modifying the stack.
Exercise 14.2 Write an independent method `public static void removeDownTo (StackADT stack, Object ob)`: It pops all values off the stack down to but not including the first element it sees that is equal to the second parameter. If none are equal, leave the stack empty.
Exercise 14.3 Write an independent method `public static Object removeSecond (QueueADT queue)`: It removes and returns the element just below the top element if the queue has at least two elements, otherwise it simply returns `null` without modifying the queue. Hint: Create a new object totally different from any that could possibly be on the queue, add it to the end of the queue, then move elements from the front of the queue to the back of the queue until the new object comes off the front of the queue.
Exercise 14.4* Write an independent method `public static void transfer (StackADT one, StackADT two)`: It transfers all elements in the first parameter onto the top of the second parameter, keeping the same order. So what was initially on top of `one` ends up on top of `two`. Hint: Use a third stack temporarily.
Exercise 14.5* Write an independent method `public static void reverse (QueueADT queue)`: It reverses the order of the elements on the queue.
Exercise 14.6* Write an independent method `public static void removeBelow (QueueADT queue, Object ob)`: It removes all values from the queue that come after the first element it sees that is equal to the second parameter. If none are equal, leave the queue as it was originally.
Exercise 14.7** Write an independent method named `interchange`: You have two StackADT parameters. All the items on each stack are to end up on the other stack in the same order, so that each stack has the items that the other stack had at the beginning. Use only one additional temporary stack.

14.2 Implementing Stacks And Queues With Arrays

The simplest implementation of a StackADT is with a partially-filled array: We use an array of Objects (call it `itsItem`) and an int value `itsSize` that tells how many elements the stack has. We store the elements in indexes 0 through `itsSize-1`. So the stack is empty if `itsSize` is zero. To pop an element off the stack, we return the value at index `itsSize-1` and decrement `itsSize`. To push an element onto the stack, we increment `itsSize` and then put the new element at index `itsSize-1`. This logic for a class of objects named ArrayStack is in Listing 14.2.

Listing 14.2 The ArrayStack class of objects

```
public class ArrayStack extends Object implements StackADT
{
   private Object[] itsItem = new Object [10];
   private int itsSize = 0;


   public boolean isEmpty()
   {  return itsSize == 0;
   }  //=====================

   public Object peekTop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      return itsItem[itsSize - 1];
   }  //=====================

   public Object pop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      itsSize--;
      return itsItem[itsSize];
   }  //=====================

   public void push (Object ob)
   {  if (itsSize == itsItem.length)
      {  Object[] toDiscard = itsItem;
         itsItem = new Object [2 * itsSize];
         for (int k = 0;  k < itsSize;  k++)
            itsItem[k] = toDiscard[k];
      }
      itsItem[itsSize] = ob;
      itsSize++;
   }  //=====================
}
```

Removing an element or peeking at an element requires that there be an element in the data structure, so each of these methods first makes sure that the stack is not empty. If it is, a runtime Exception is to be thrown. The obvious choice is an **IllegalStateException** (from `java.lang`).

When the ArrayStack object is first created, we start with an array of size 10 (an arbitrary choice). Adding an element requires that there be room in the array. If not, we have to make the array bigger to hold the added element. Since we cannot simply add more components to an existing array, we create a new array that is twice as large and transfer the data to that other array. It becomes the new `itsItem` array. Figure 14.2 is the UML class diagram for the ArrayStack class.

The **internal invariant** of a class of objects is the condition that (a) each method ensures is true when the method exits, so that (b) each method can rely on it being true when the method starts execution, because (c) no outside method can change things so that the condition becomes false (due to encapsulation). The purpose of an internal invariant is to describe the relationship of the abstract concept to the state of the instance variables. The ArrayStack class implements the abstract concept of a stack as follows:

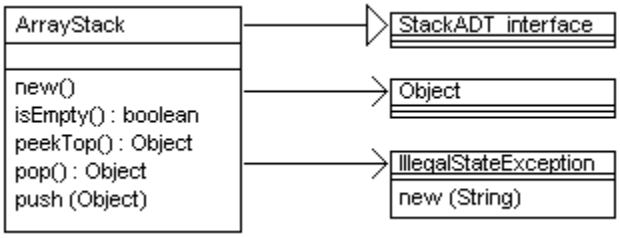


Figure 14.2 UML class diagram for the ArrayStack class

Internal invariant for ArrayStacks

- The int value `itsSize` is the number of elements in the abstract stack. These elements are stored in the array of Objects `itsItem` at the components indexed 0 up to but not including `itsSize`.
- If the stack is not empty, the top element of the stack is at index `itsSize-1`.
- If `k` is any positive integer less than `itsSize`, then the element at index `k-1` is the element that will be on top of the stack immediately after popping off the element at index `k`.

Simplistic implementation of queues

We could implement QueueADT the same way, except that elements are removed from the other end of the array from that on which elements are added. So we could code `enqueue` the same as `push` but code `dequeue` as follows:

```
public Object dequeue()  // in a simplistic implementation
{  if (isEmpty())
      throw new IllegalStateException ("queue is empty");
   Object valueToReturn = itsItem[0];
   for (int k = 1;  k < itsSize;  k++)
      itsItem[k - 1] = itsItem[k];
   itsSize--;
   return valueToReturn;
}  //======================
```

Note that the for-statement copies the contents of `itsItem[1]` into the variable `itsItem[0]`, then the contents of `itsItem[2]` into `itsItem[1]`, etc. Copying in the opposite order would lose information. That is why the value to return is saved in a local variable before moving values down.

We leave the coding of `isEmpty` unchanged. The coding of `peekFront` is the same as `peekTop` except we return `itsItem[0]`. That completes the rather inefficient "move-them-all" implementation of QueueADT.

A better implementation of queues

The implementation of queues just described is very wasteful of execution time. If for instance the queue normally has around 100 elements in it, then each call of `dequeue` requires shifting 100 values around.

A better way is to keep track of two places in the array, the index of the front of the queue and the index of the rear of the queue. Call these instance variables `itsFront` and `itsRear`. Forget about `itsSize`. You add values at the rear and remove them at the front of the queue (except of course you cannot remove anything from an empty queue). So `peekFront` returns `itsItem[itsFront]`. The implementation of `dequeue` is as follows. Note that it corresponds line-for-line with ArrayStack's `pop`:

```
public Object dequeue()           // in ArrayQueue
{  if (isEmpty())
      throw new IllegalStateException ("queue is empty");
   itsFront++;
   return itsItem[itsFront - 1];
}  //======================
```

Say `itsFront` is 10. Then if `itsRear` is 12, the queue has three values in it (at components 10, 11, and 12). If `itsRear` is 10, the queue has one value in it (at component 10). In general, the number of values in the queue is `itsRear - itsFront + 1`. So how do you tell when the queue is empty? When this expression has the value zero, namely, when `itsRear` is 1 less than `itsFront`. Since you should add the first element at index 0 for a newly created queue, `itsFront` should initially be 0, which means that `itsRear` must initially be -1 (1 less than `itsFront` because the queue is initially empty). This coding is in the upper part of Listing 14.3 (see next page).

The enqueue method

The basic logic of the `enqueue` method is to increment `itsRear` and put the given value at that component:

```
itsRear++;
itsItem[itsRear] = ob;
```

But you have a problem: What if `itsRear` is already up to `itsItem.length - 1` and so there is no room to add the new value? This will happen once the number of `enqueue` calls is `itsItem.length`. There are two possibilities when in this case: Either the array has plenty of room at the lower indexes or it does not. In the latter case, you have a simple solution: Just move all the values back down to the front of the array en masse. We choose to do this as long as the array is no more than three-quarters full:

```
if (itsFront > itsRear / 4)
   for (int k = 0;  k <= itsRear - itsFront;  k++)
      itsItem[k] = itsItem[k + itsFront];
```

Check that these values are correct: Clearly the first iteration moves the front value in `itsItem[itsFront]` down to `itsItem[0]`. And the last iteration moves `itsItem[itsRear - itsFront + itsFront]`, which is `itsItem[itsRear]`, which of course should be the last one moved. And it is moved down to `itsItem[itsRear - itsFront]`. This illustrates a key technique for verifying that a for-statement is correct: Calculate what happens on the first and last iteration; otherwise it is easy to be "off by one".

Since the position of the values in the queue has changed, you also have to update two instance variables:

```
itsFront = 0;
itsRear -= itsFront;
```

But what if the array is nearly full when you are called upon to add another value? You simply do what was done in the earlier Listing 14.2 for ArrayStack: Transfer all the elements to another array that is twice as large. This coding is in the lower part of Listing 14.3. Handling the case when `itsRear` reaches the rear of the array is in a separate private method because it would otherwise obscure the basic logic of `enqueue`.

Listing 14.3 The ArrayQueue class of objects

```
public class ArrayQueue extends Object implements QueueADT
{
   private Object[] itsItem = new Object [10];
   private int itsFront = 0;  // location of front element
   private int itsRear = -1;  // location of rear element


   public boolean isEmpty()
   {  return itsRear == itsFront - 1;
   }  //=====================

   public Object peekFront()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      return itsItem[itsFront];
   }  //=====================

   public Object dequeue()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      itsFront++;
      return itsItem[itsFront - 1];
   }  //=====================

   public void enqueue (Object ob)
   {  if (itsRear == itsItem.length - 1)
         adjustTheArray();
      itsRear++;
      itsItem[itsRear] = ob;
   }  //=====================

   private void adjustTheArray()
   {  if (itsFront > itsRear / 4)
      {  itsRear -= itsFront;
         for (int k = 0;  k <= itsRear;  k++)
            itsItem[k] = itsItem[k + itsFront];
         itsFront = 0;
      }
      else
      {  Object[] toDiscard = itsItem;
         itsItem = new Object [2 * itsRear];
         for (int k = 0;  k <= itsRear;  k++)
            itsItem[k] = toDiscard[k];
      }  // automatic garbage collection gets rid of toDiscard
   }  //=====================
}
```

Another popular way of implementing a queue avoids all movement of data except when the queue is full: When `itsRear == itsItem.length - 1` but `itsFront` is at least 2, add the next value at index 0. That is, the rear starts over from zero, leaving the front where it was. Double the size of the array when the element you are adding would fill the array (do not wait until after it is full). This "avoid-all-moves" approach is harder to understand and code than the "move-when-forced" approach used for Listing 14.3, but it executes somewhat faster.

Exercise 14.8 Add a method `public int size()` to the ArrayQueue class: The executor tells the number of values currently in the queue.
Exercise 14.9 Add a method `public String toString()` to the ArrayQueue class: The executor returns the concatenation of the string representation of each element currently in the queue with a tab character between any two elements, in order from front to rear. This is very useful for debugging purposes.
Exercise 14.10 How would the coding change in Listing 14.2 if you decided to have an instance variable `itsTop` keep track of the index of the top element, instead of `itsSize`?
Exercise 14.11 Write a method `public void removeBelow (Object ob)` that could be added to ArrayStack: The executor removes all values from the stack that are below the element closest to the top that is equal to the parameter. If none are equal, leave the stack as it was originally. Handle `ob == null` correctly.
Exercise 14.12 Add a method `public boolean equals (Object ob)` to the ArrayStack class: The executor tells whether `ob` is an ArrayStack whose elements are equal to its own in the same order. Do not throw any Exceptions.
Exercise 14.13* Add a method `public boolean equals (Object ob)` to the ArrayQueue class: The executor tells whether `ob` is an ArrayQueue with whose elements are equal to its own in the same order. Do not throw any Exceptions.
Exercise 14.14* Add a method `public void clear()` to the ArrayQueue class: The executor deletes all the elements it contains, thereby becoming empty.
Exercise 14.15* Add a method `public int search (Object ob)` to the ArrayStack class: The executor tells how many elements would have to be popped to have `ob` removed from the stack; it returns -1 if `ob` is not in the stack.
Exercise 14.16* Write out the internal invariant for ArrayQueues.
Exercise 14.17** Rewrite Listing 14.3 to "flip the array over": Store the front value initially at index `itsItem.length-1` and have `itsRear` move towards index 0.
Exercise 14.18** Write the method `public Object dequeue()` for the "avoid-all-moves" method of implementing QueueADT.

14.3 Implementing Stacks And Queues With Linked Lists

We next implement a StackADT object as a **linked list of Nodes**. A Node object stores two pieces of information: a reference to a single piece of data of type Object and a reference to another Node object. If for instance you want to represent a sequence of three data values as a linked list of Nodes, you put the three data values in three different Node objects and have the first Node refer to the second Node, the second Node refer to the third, and the third not refer to any Node at all.

The Node class

The Node class is defined in Listing 14.4 (see next page). A Node object's data is referenced by `itsData` and the Node that comes next after it in the linked list is referenced by `itsNext`. The following statements create a linked list of Nodes having the words "the", "linked", "list" in that order. The first statement creates a Node object in `nodeA` whose data value is "list" and which is linked up to no node (indicated by having `itsNext` be `null`). The second statement creates another Node object in `nodeB` which is linked up to `nodeA`. The third statement creates another Node object in `nodeC` which is then linked up to the second node, `nodeB`. Figure 14.3 shows how the list will look when it is done:

```
Node nodeA = new Node ("list", null);
Node nodeB = new Node ("linked", nodeA);
Node nodeC = new Node ("the", nodeB);
```

Listing 14.4 The Node class

```
public class Node extends Object
{
   private Object itsData;
   private Node itsNext;

   public Node (Object data, Node next)
   {  super();
      itsData = data;
      itsNext = next;
   }  //=====================

   /** Return the data attribute of the Node. */

   public Object getData()
   {  return itsData;
   }  //=====================

   /** Return the next attribute of the Node. */

   public Node getNext()
   {  return itsNext;
   }  //=====================

   /** Replace the next attribute. */

   public void setNext (Node next)
   {  itsNext = next;
   }  //=====================

   /** Replace both the data and the next attributes. */

   public void setNode (Object data, Node next)
   {  itsData = data;
      itsNext = next;
   }  //=====================
}
```

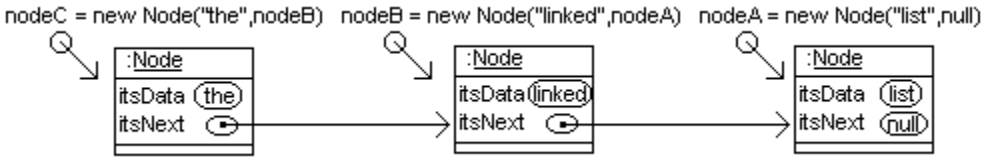


Figure 14.3 UML object diagram of three nodes in a linked list

Nodes in a NodeStack object

A NodeStack object will have an instance variable `itsTop` for the first Node object on its list. Initially `itsTop` is `null`, which is a signal that the stack is empty (after all, you cannot have any data without a Node to put it in). Suppose a particular NodeStack object has a linked list of two or more Nodes. Coding to swap the two data values in the first two Nodes could be as follows (the last two statements change the data attribute):

```
Object saved = itsTop.getData();
Node second = itsTop.getNext();
itsTop.setNode (second.getData(), second);
second.setNode (saved, second.getNext());
```

Yes, this is rather clumsy without a separate `setData` method. But it works, and in practice we usually change both values or else just the `itsNext` attribute.

Coding to add a new data value "sam" at the beginning of a list could be as follows. Actually, this coding will work even when the linked list is empty (since then `itsTop` has the value `null`):

```
Node newNode = new Node ("sam", itsTop);
itsTop = newNode;
```

It follows that the `push` method for a linked list implementation of StackADT only needs one statement to add `ob` to the front of the list (the effect is illustrated in Figure 14.4):

```
itsTop = new Node (ob, itsTop);
```

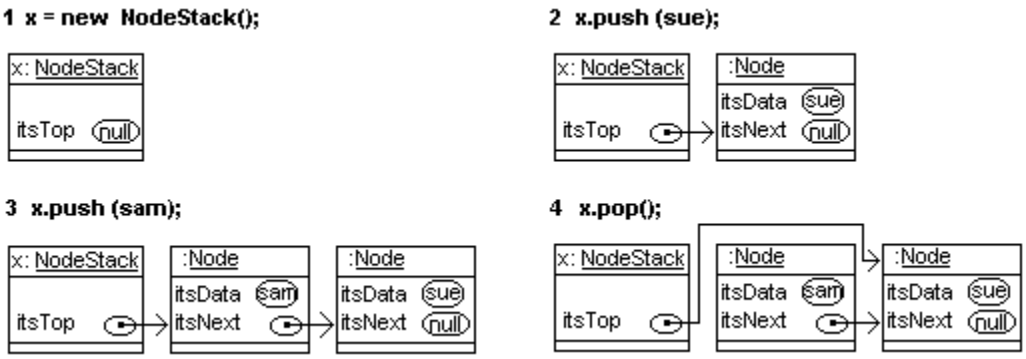


Figure 14.4 UML object diagrams for stack operations in NodeStack

The `peekTop` method is quite straightforward: First you check to make sure the stack is not empty (otherwise you throw an Exception). If the stack has at least one Node, the first Node is the top of the stack, so you simply return `itsTop.getData()`. This coding is in the upper part of Listing 14.5 (see next page).

The `pop` method is the most complex of the NodeStack methods: Once you check that the stack is not empty, you discard the first Node in the linked list, since it contains the data on top of the stack that you are to remove. This can be done simply with the following statement:

```
itsTop = itsTop.getNext();
```

That stores a reference to the second Node on the list in the `itsTop` instance variable. If there was only one Node on the list in the first place, `itsTop.getNext()` has the value `null`, so `itsTop` is now `null`, which signals that the stack is empty. The coding for `pop` is in the lower part of Listing 14.5.

Internal invariant for NodeStacks

- If the abstract stack is empty, the Node value `itsTop` is `null`.
- If the abstract stack is not empty, `itsTop` refers to the first Node in a linked list of Nodes and `itsTop.getData()` is the top element of the stack.
- If `p` is any Node in that linked list for which `p.getNext()` is not `null`, then the element `p.getNext().getData()` is the element that will be on top of the stack immediately after popping off the element `p.getData()`.
- The Nodes of one NodeStack are all different objects from those in any other.

Listing 14.5 The NodeStack class of objects

```
public class NodeStack extends Object implements StackADT
{
   private Node itsTop = null;


   public boolean isEmpty()
   {  return itsTop == null;
   }  //====================

   public Object peekTop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      return itsTop.getData();
   }  //====================

   public Object pop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      Node toDiscard = itsTop;
      itsTop = itsTop.getNext();
      return toDiscard.getData();
   }  //====================

   public void push (Object ob)
   {  itsTop = new Node (ob, itsTop);
   }  //====================
}
```

Postfix algebraic notation

The programming language Forth is based on postfix notation, in which you enter the operator after you enter the two values it is to operate on: The sum of 2 and 3 is written as `2 3 +` and the difference of two squares is written as `x x * y y * -`. Some calculators also use postfix notation (a major programming project is to develop a graphical user interface for such a calculator).

The runtime system for Forth and the internal circuitry for the postfix calculators use a stack to store the values yet to be processed. For instance, `x x * y y * -` is processed as follows when `x` is 5 and `y` is 2: Push 5 on the stack; push 5 again; pop off the two 5's and push the product 25; push 2 on the stack; push 2 again; pop off the two 2's and push the product 4; finally, pop off the 25 and 4 and push the difference 21.

Implementing queues with linked lists

A linked list is an excellent way to implement QueueADT, as long as you keep track of both the front and the rear of the list. That way, you can quickly add an element or remove an element. For this NodeQueue class, begin by declaring two instance variables `itsFront` and `itsRear`, each referring to a Node. In general, the queue's `itsFront` will always refer to the Node containing the first data value (if any) and the queue's `itsRear` will refer to the Node containing the last data value (if any).

An empty queue has `null` for `itsFront`, since there are no data values and so no Nodes at all. The `isEmpty` , `dequeue`, and `peekFront` methods are precisely the same as `isEmpty`, `pop`, and `peekTop` are for NodeStack except of course `itsFront` is used in place of `itsTop`. The implementation so far is in the top part of Listing 14.6.

Listing 14.6 The NodeQueue class of objects

```
public class NodeQueue extends Object implements QueueADT
{
   private Node itsFront = null;
   private Node itsRear;


   public boolean isEmpty()
   {  return itsFront == null;
   }  //=====================

   public Object peekFront()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      return itsFront.getData();
   }  //=====================

   public Object dequeue()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      Node toDiscard = itsFront;
      itsFront = itsFront.getNext();
      return toDiscard.getData();
   }  //=====================

   public void enqueue (Object ob)
   {  Node toBeAdded = new Node (ob, null);
      if (isEmpty())
         itsFront = toBeAdded;
      else
         itsRear.setNext (toBeAdded);
      itsRear = toBeAdded;
   }  //=====================
}
```

The enqueue method

To enqueue a new data value to the end of the queue, you normally create a new Node attached to the Node referred to by `itsRear`. Since that new Node is now the last Node, you change the value of `itsRear` to refer to the new Node.

However, if the queue is empty, you cannot attach a new Node to the last Node, because there is no last Node. In that case, you need to have the queue's `itsFront` refer to the new Node, because the new Node is the first Node. But you also need to have the queue's `itsRear` refer to the new Node, because the new Node is also the last Node. This coding is in the lower part of Listing 14.6.

Exercise 14.19 Write a method `public void dup()` that could be in NodeStack: The executor pushes the element that is already on top of the stack (so it now occurs twice). Throw an Exception if the stack is empty.
Exercise 14.20 Write a method `public void swap2and3()` that could be in NodeStack: The executor swaps the second element with the third element in the stack. It has no effect if the stack has less than three elements.
Exercise 14.21 Rewrite the `enqueue` method to execute faster by not assigning the newly-constructed Node to a local variable, but instead assigning it directly where it goes.

Exercise 14.22 Write a method `public void add (NodeQueue queue)` that could be in NodeQueue: The executor appends `queue` 's elements in the same order and sets `queue` to be empty. Precondition: The executor is not empty.
Exercise 14.23 Compute the values of these two postfix expressions:
(a) `12 10 3 5 + - /` (b) `5 4 - 3 2 - 1 - -`
Exercise 14.24* Write the two postfix expressions in the previous exercise in ordinary algebraic notation without evaluating any of the parts.
Exercise 14.25* Write the obvious `setData` method for the Node class.
Exercise 14.26* Add an instance method `public int size()` to the NodeStack class that tells the number of elements currently on the stack. Also add an instance variable `itsSize` to keep track of the current number so `size()` can use it.
Exercise 14.27* Add a method `public Object firstAdded()` to NodeQueue: The executor tells what was the first element ever added to it, even if it was removed later. It returns `null` if the queue has always been empty.
Exercise 14.28* Write out the internal invariant for NodeQueues.

14.4 Interface For Priority Queues

Say you have a number of jobs to do, each with its own priority. As time becomes available to do a job, you select the one with the highest priority. Whenever you receive another job to do, you add it to the list of jobs on hand. An example of this situation is a printer queue: A high-speed printer that serves a large number of users is continually receiving new print jobs to do and printing them as fast as it can. It prints those with highest priority first.

We shall assume that the priority is determined by a boolean method named `prior` belonging to a **Prioritizer** object: For a given Prioritizer `test`, `test.prior(x,y)` is true when `x` has a higher priority than `y`. So each time we start a new job, it should be the one that is earliest as determined by the appropriate `prior` method. We use classes of objects that implement the Prioritizer interface, which is defined as shown in the upper part of Listing 14.7 (see next page).

In general, you may define an interface by specifying the headings of the public instance methods its implementors are to have, with a semicolon in place of the body of each such method. If you compile this interface definition, the compiler will allow you to put `implements Prioritizer` in the heading of any class that contains a method with the specified heading.

Priority queues

You can store tasks to be prioritized in a **priority queue**. This is a data structure that has available the methods described in the PriQue interface in the lower part of Listing 14.7. Note the similarity with the stack and queue methods described in the earlier Listing 14.1:

- `add` corresponds to `push` and `enqueue`;
- `get` corresponds to `peekTop` and `peekFront`;
- `remove` corresponds to `pop` and `dequeue`.

One difference is that `null` values are not allowed in a priority queue (what would the priority of `null` be?) though they are allowed in stacks and queues. The only other difference is in the effect of `remove` and `pop` and `dequeue`: `pop` delivers the object that has been in the data structure for the shortest period of time, `dequeue` delivers the object that has been in the data structure for the longest period of time, and `remove` delivers the object that has the highest priority (in case of ties in priority, the element that has been in the data structure the longest is removed).

Listing 14.7 The Prioritizer and PriQue interfaces

```
public interface Prioritizer         // not in the Sun library
{
   /** Tell whether one has higher priority than two.  Throw
    *  a RuntimeException if either is null or they are not
    *  comparable with each other.  */
   public boolean prior (Object one, Object two);
}
//#############################################################


public interface PriQue              // not in the Sun library
{
   /** Tell whether the priority queue has no more elements. */
   public boolean isEmpty();

   /** Return the object remove would return, without modifying
    *  anything.  Throw an Exception if the queue is empty. */
   public Object get();

   /** Delete the object of highest priority and return it.  In
    *  case of a tie in priorities, use first-in-first-out logic.
    *  Priority is determined by a Prioritizer assigned by the
    *  constructor. Throw an Exception if the queue is empty. */
   public Object remove();

   /** Add the given element ob to the priority queue.
    *  This method has no effect if ob is null. */
   public void add (Object ob);

   // Recommended form for the constructor:
   // public WhateverPriQue (Prioritizer givenTest);
}
```

Creating Prioritizer classes

In many cases, it is appropriate to define `test.prior(x, y)` to mean that `x` is smaller than `y` according to the `compareTo` method. That is, you often would use the following:

```
public class Comparer implements Prioritizer
{
   public boolean prior (Object one, Object two)
   {  return ((Comparable) one).compareTo (two) < 0;
   }  //======================
}
```

We will shortly discuss an implementation of PriQue named ArrayOutPriQue. Suppose you want to have a priority queue of String objects where priority is alphabetical order (or the equivalent provided by the String `compareTo` method). You could create the appropriate data structure with the following statement:

```
PriQue queue = new ArrayOutPriQue (new Comparer());
```

Suppose however that you have several classes with a `getCost()` method, all declared to `implement Priceable`. Then you could create the priority queue with a parameter of `new ByCost()` if you have the following definitions:

```
public interface Priceable
{     public double getCost();
}

public class ByCost implements Prioritizer
{
   public boolean prior (Object one, Object two)
   {  return ((Priceable) s1).getCost()
              < ((Priceable) s2).getCost();
   }  //======================
}
```

14.5 Implementing Priority Queue With Arrays

One obvious way to manage the values in a priority queue is to store them in a partially-filled array in the order in which they will be taken out, so that the next value to remove is always at `itsItem[itsSize-1]` (analogous to ArrayStack in Listing 14.2). This plan almost completely determines the coding for the four methods. Three of them are shown for the ArrayOutPriQue class in Listing 14.8 (see next page); `isEmpty`, `get`, and `remove` are coded exactly the same as `isEmpty`, `peekTop`, and `pop` for ArrayStack. It has two constructors: one for when you want to specify your own Prioritizer method and the other (without a parameter) for when you just want to use `compareTo`.

If the `add` method is called when `itsSize` is `itsItem.length`, you need to increase the size of the array. In any case, when the `add` method is called, you move values up until you open up a spot where the new value goes to keep all values in order; it generally goes at an index `k` such that `itsTest.prior (ob, itsItem[k - 1])` is `true`. The coding of the `add` method is left as an exercise. Figure 14.5 illustrates the process (small numbers indicate high priority): Adding an element of priority 5 requires shifting the three elements of higher priorities 4, 2, and 1 further towards the rear of the array.

indexes of components:	0	1	2	3	4	5	6
current status of the array:	8	6	4	2	1	no data	no data
call add(5):	8	6	**5**	4	2	1	no data
call remove():	8	6	5	4	2	no data	no data

Figure 14.5 For ArrayOutPriQue: Call add(5), then call remove()

Internal invariant for ArrayOutPriQues

- The int value `itsSize` is the number of elements in the abstract priority queue. These elements are stored in the array of Object objects `itsItem` at the components indexed 0 up to but not including `itsSize`.
- If k is any positive integer less than `itsSize`, then the element at index `k` has either equal or higher priority than the element at index `k-1`; and if their priorities are equal, the element at index `k` has been in the queue longer. In particular, the element at index `itsSize-1` has the highest priority of all elements in the priority queue.

Listing 14.8 An array implementation of PriQue, one method left as an exercise

```
public class ArrayOutPriQue extends Object implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private Prioritizer itsTest;


   public ArrayOutPriQue (Prioritizer givenTest)
   {  super();
      itsTest = givenTest;
   }  //======================

   public ArrayOutPriQue()
   {  super();
      itsTest = new Comparer();
   }  //======================

   public boolean isEmpty()
   {  return itsSize == 0;
   }  //======================

   public Object get()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      return itsItem[itsSize - 1];
   }  //======================

   public Object remove()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      itsSize--;
      return itsItem[itsSize];
   }  //======================

   public void add (Object ob)
   {  // left as an exercise
   }  //======================
}
```

Storing the elements in the order they come in

An alternative implementation for a priority queue is to just put the next value added in the array at index `itsSize`, so they remain in the order they are put in the structure. So `isEmpty` and `add` are coded about the same as `isEmpty` and `push` for ArrayStack.

When the `remove` method is called, you then have to search through the array to find the value of highest priority and return it. Coding for the `remove` method of this ArrayInPriQue class of objects could be as shown in the lower part of Listing 14.9 (see next page).

If two elements are tied for the highest priority, this `remove` method does not necessarily return the one that has been on the queue for the longest time, as required by the PriQue specifications. Correcting this defect is left as an exercise. Notation: "ArrayIn" reminds you they are stored in an array in the order of input to the data structure; "ArrayOut" reminds you they are stored in an array in the order of output from the data structure.

Listing 14.9 The ArrayInPriQue class of objects

```
public class ArrayInPriQue extends Object implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private Prioritizer itsTest;


   public ArrayInPriQue (Prioritizer givenTest)
   {  super();
      itsTest = givenTest;
   }  //======================

   public boolean isEmpty()
   {  return itsSize == 0;
   }  //======================

   public Object get()
   {  return null;  // stubbed and left as an exercise
   }  //======================

   public Object remove()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      int best = 0;
      for (int k = 1;  k < itsSize;  k++)
         if (itsTest.prior (itsItem[k], itsItem[best]))
            best = k;
      Object valueToReturn = itsItem[best];
      itsSize--;
      itsItem[best] = itsItem[itsSize];
      return valueToReturn;
   }  //======================

   public void add (Object ob)
   {  if (ob != null)
      { }// same coding goes here as for push in ArrayStack
   }  //======================
}
```

Note that the call of `itsTest.prior` can invoke the `prior` method that uses a String's `compareTo` method or the `prior` method that uses a Priceable object's `getCost` method or any of a number of other possible codings. It is therefore a polymorphic method call.

Exercise 14.29 Write the `add` method for ArrayOutPriQue. Make sure that all elements that are stored closer to index 0 in the array than the inserted value have lower priority.
Exercise 14.30 Write the `get` method for ArrayInPriQue.
Exercise 14.31 Revise the `remove` method for ArrayInPriQue so it always returns elements of equal priority in first-in-first-out order. Do so by replacing the next-to-last statement by coding that moves elements down while keeping their original order.
Exercise 14.32* Write a complete implementation of PriQue for which the element of highest priority is always kept in `itsItem[itsSize-1]` and the rest are in the order in which they were entered. Note that this makes `get` execute very quickly.

14.6 Implementing Priority Queues With Linked Lists

You can store the elements of a priority queue in a linked list instead of in an array. This section discusses two different ways to do this. In the process, you learn how to progress through a linked list of Nodes.

Implementing PriQue with a linked list in the order elements go out

You could keep the data in order of priority, with the highest-priority data in the first Node of the linked list. Then when you want to remove it or just get it, you have it immediately available. This NodeOutPriQue class needs an instance variable to record the first Node on the linked list; call it `itsFirst`. So the `remove` and `get` methods would be coded the same as `dequeue` and `peekFront` for NodeQueue (in Listing 14.5), except that "itsFirst" replaces "itsFront". The only tricky part is the `add` method, since it has to put the given element into the linked list after every value of equal or higher priority.

It will turn out that the coding of `add` is much easier if you always keep an extra Node at the end of the linked list with `null` data in that **trailer node**. So a NodeOutPriQue object is constructed with `itsFirst` being a reference to a trailer node that has `null` for `itsData` and `null` for `itsNext`. The priority queue is empty whenever `itsFirst.getData()` is `null`, not when `itsFirst` is `null`. The coding so far, for `isEmpty`, `get`, and `remove`, is in the upper part of Listing 14.10 (see next page).

For the `add` method, you need to search through the linked list for the Node that contains the value that should come after the given element in the list. Have a variable `p` refer to that Node. The statement `p = p.getNext()` moves p from whichever Node it is on to the next Node. The loop stops at the trailer node if not before, i.e., when `p.getData() == null`. Then you can insert the given element `ob` in that Node `p` and make a new Node after `p` to contain the data value that was in `p`, as follows:

```
p.setNode (ob, new Node (p.getData(), p.getNext()));
```

This would not work if `ob` is to be added at the end of a simple linked list, since then `p` would be `null`. But with the trailer node at the end, if `ob` is to be added at the end of the list, the statement above copies the information from the trailer node into a new trailer node and has `ob` replace the `null` data in the old trailer node. The coding is in the lower part of Listing 14.10. Figure 14.6 shows stages in the execution of the `add` method.

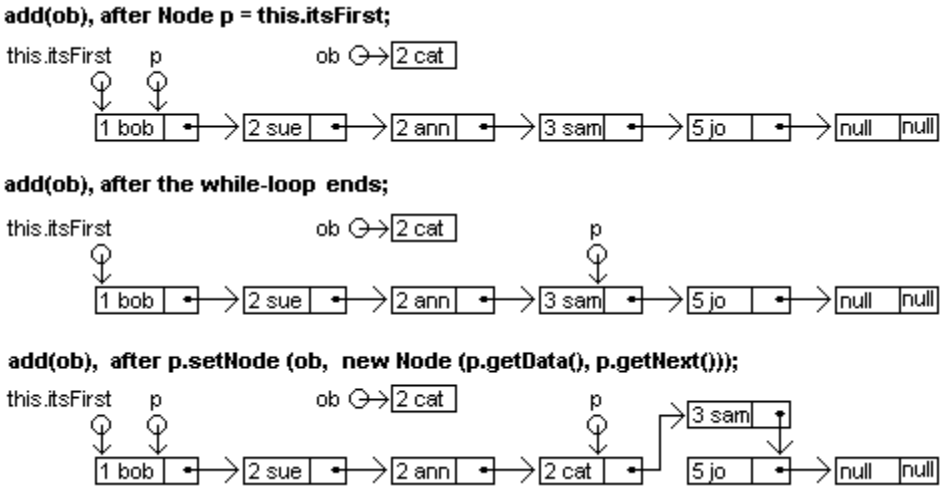


Figure 14.6 Adding a new element of priority 2 in NodeOutPriQue

Listing 14.10 The NodeOutPriQue class of objects, using trailer nodes

```
public class NodeOutPriQue extends Object implements PriQue
{
   private Node itsFirst = new Node (null, null);
   private Prioritizer itsTest;


   public NodeOutPriQue (Prioritizer givenTest)
   {  super();
      itsTest = givenTest;
   }  //======================

   public boolean isEmpty()
   {  return itsFirst.getData() == null;
   }  //======================

   public Object get()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      return itsFirst.getData();
   }  //======================

   public Object remove()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      Object valueToReturn = itsFirst.getData();
      itsFirst = itsFirst.getNext();
      return valueToReturn;
   }  //======================

   public void add (Object ob)
   {  if (ob != null)
      {  Node p = this.itsFirst;
         while (p.getData() != null
                           && ! itsTest.prior (ob, p.getData()))
         {  p = p.getNext();
         }
         p.setNode (ob, new Node (p.getData(), p.getNext()));
      }
   }  //======================
}
```

Implementing PriQue with a linked list in the order elements come in

The NodeInPriQue class implements a priority queue as follows: Add each element to the front of the linked list as it comes in; when you need to produce the element of highest priority, search through the list to find it (just as for ArrayInPriQue in the preceding section). Each NodeInPriQue object is to have an instance variable `itsFirst` to note the first Node on its linked list, as shown in Listing 14.11 (see next page).

Again, the coding is easier if you have an extra Node at the end of the linked list with `null` data in it. So a NodeInPriQue object is constructed with `itsFirst` being a reference to a trailer node with `null` for `itsData` and `null` for `itsNext`. The coding for `add` is as follows, after making sure the element to be added is not `null`:

```
   itsFirst = new Node (ob, itsFirst);
```

Listing 14.11 The NodeInPriQue class of objects, using trailer nodes

```
public class NodeInPriQue extends Object implements PriQue
{
   private Node itsFirst = new Node (null, null);
   private Prioritizer itsTest;


   public NodeInPriQue (Prioritizer givenTest)
   {  super();
      itsTest = givenTest;
   }  //======================

   public boolean isEmpty()
   {  return itsFirst.getData() == null;
   }  //======================

   public Object get()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      return best().getData();
   }  //======================

   private Node best()
   {  Node best = itsFirst;
      for (Node p = itsFirst.getNext();  p.getData() != null;
                                         p = p.getNext())
      {  if (! itsTest.prior (best.getData(), p.getData()))
            best = p;
      }
      return best;
   }  //======================

   public Object remove()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      Node holdsBest = best();
      Object valueToReturn = holdsBest.getData();
      Node toDelete = holdsBest.getNext();
      holdsBest.setNode (toDelete.getData(), toDelete.getNext());
      return valueToReturn;
   }  //======================

   public void add (Object ob)
   {  if (ob != null)
         itsFirst = new Node (ob, itsFirst);
   }  //======================
}
```

When you want to find the element of highest priority, you need to search through the linked list for the Node that contains it. The search process, once you have made sure the linked list is not empty, starts by assigning to a variable `best` a reference to the first Node on the linked list. Then it goes through the rest of the Nodes one at a time. Whenever it finds a Node that contains data that is of equal or higher priority than the data in the `best` Node, that Node it found becomes the new `best` Node. The search process stops when it reaches the trailer node. The coding is in a private method in the middle part of Listing 14.11.

This coding has `p` refer to the second Node, then the third Node, then the fourth Node, until you run out of elements to look at. Now `get` simply returns the data stored in the Node found by that search process.

The `remove` method does the same as `get` except that it also deletes the value to be returned. You have to avoid changing the order of the remaining elements. To delete the element in a Node referenced by `holdsBest`, it is sufficient to replace it by the element in the Node following `holdsBest` and then delete the Node following `holdsBest`. This is where the existence of the trailer node comes in handy. It guarantees that there will always be a Node following `holdsBest`, the trailer node if nothing else. The coding for the `remove` method is in the lower part of Listing 14.11.

An alternative to having a trailer node is to search through the list for the Node before the one that contains the highest-priority element and set that Node's `itsNext` value to the one after the one containing the highest-priority element. This is more complicated, so we do not do that here. You could try it if you want, however, so you will understand why we prefer to use a trailer node.

Exercise 14.33 Rewrite the `remove` method for NodeInPriQue to not use a local variable for the Node to be deleted.
Exercise 14.34 How would you revise the `add` method for NodeOutPriQue if you were not to allow two elements with equal priority?
Exercise 14.35 Add a method `public int size()` to the NodeInPriQue class: The executor counts and returns the number of values currently in the priority queue.
Exercise 14.36 Add a method `public String toString()` to the NodeOutPriQue class: The executor returns the concatenation of the string representation of each element currently in the queue with a tab character between any two elements, in order front to rear. This is very useful for debugging purposes.
Exercise 14.37 Write a method `public void removeBelow (Object ob)` that could be added to NodeStack: The executor removes all values from the stack that come below the element closest to the top that is equal to the parameter. If none are equal, leave the stack as it was originally. Precondition: `ob` is not `null`.
Exercise 14.38 Write a method `public void removeAbove (Object ob)` that could be added to NodeOutPriQue: The executor removes all values from the stack that have higher priority than the parameter. Precondition: `ob` is not `null`.
Exercise 14.39 Rewrite the `add` method for NodeInPriQue to add the given element in the second position of the linked list if the list is not empty and the element does not have higher priority than the data in the first Node.
Exercise 14.40* Rewrite the `remove` method for NodeInPriQue so that, in conjunction with the `add` method of the preceding exercise, the element that is to be removed next is always `itsFirst.getData()` and the rest are in the order they were added. Note that this makes `get` execute much faster.
Exercise 14.41* Explain why the condition in the private `best` method of Listing 14.11 cannot be written as `itsTest.prior (p.getData(), best.getData())` instead.
Exercise 14.42* Write out the internal invariant for NodeOutPriQues.
Exercise 14.43* Write a method `public void removeBelow (Object ob)` that could be added to NodeOutPriQue: The executor removes all values from the priority queue that have lower priority than the given element.
Exercise 14.44* Same as the preceding exercise, but for NodeInPriQue.
Exercise 14.45* Revise the `add` and `isEmpty` methods of NodeOutPriQue to allow `null` values on the priority queue, with lower priority than any non-null value.
Exercise 14.46* Rewrite the full implementation of NodeInPriQue so that each element is added to the end of a linked list but without looping (i.e., the reverse ordering).
Exercise 14.47** Rewrite the full implementation of NodeOutPriQue without a trailer node, so that the number of Nodes equals the number of elements.
Exercise 14.48** Rewrite the full implementation of NodeInPriQue without a trailer node, so that the number of Nodes equals the number of elements.

Exercise 14.49** Write a method `public void add (NodeOutPriQue queue)` that could be in NodeOutPriQue: The executor interweaves `queue` 's elements in priority order and sets `queue` to be empty.

14.7 Implementing Priority Queues With Linked Lists Of Queues

Some situations have only a few priority levels. For instance, you might have only five different priority levels with hundreds of elements at each level. If you add an element with low priority, you may have to go through many hundreds of elements to find the place where your given element is to be inserted.

In such a situation, the logic would execute substantially faster if you had just five regular queues, one for each priority level. Then you could go to the correct queue in at most five steps and quickly append the given element at the end of that queue. You could have a linked list of queues instead of a linked list of elements. You keep these queues in order of the priorities of the elements on them, with the highest priorities first in the linked list.

This NodeGroupPriQue implementation uses a Node class where `itsData` is a QueueADT value instead of Object. This different Node class is declared inside the NodeGroupPriQue class with the heading `private static Node`. Such a declaration means that the word "Node" inside the NodeGroupPriQue class is a reference to the nested class, not the outside class of Listing 14.4 (the nested class **shadows** the other). Other than that, nesting only affects visibility: Methods inside NodeGroupPriQue can access the public variables of the Node class but outside methods cannot. So the principle of encapsulation is not violated by making Node's instance variables public.

Each NodeGroupPriQue object has an instance variable `itsFirst` that keeps track of the linked lists of queues, all of which are to be non-empty. Again it is highly convenient to have a trailer node at the end of this linked list.

The isEmpty, get, and remove methods

The `isEmpty` method only needs to check that `itsData` for the first Node on the linked list is `null`. The element to be returned by the `get` method is the front element of the queue with the highest priority. Since that is the first queue in the linked list, you verify that the linked list is not empty, then return the `peekFront` value for that queue (since the queue is known to be non-empty). You may refer directly to `itsFirst.itsData` rather than `itsFirst.getData()`. This coding is in the upper part of Listing 14.12 (see next page).

The `remove` method is about the same as the `get` method except it returns `itsFirst.itsData.dequeue()`, which removes the front element on the queue of elements of highest priority. The call of `remove` might leave that first queue empty. Since the internal invariant of this implementation requires that you only store non-empty queues, you must discard the first queue if it has become empty as a consequence of dequeueing an element from it. This coding is in the middle part of Listing 14.12.

You should compare the coding throughout this listing with the coding for NodeOutPriQue in Listing 14.10 to see how similar they are. In fact, if no two elements can have the same priority, the NodeGroupPriQue implementation becomes the same as NodeOutPriQue except for an awful lot of extraneous creating and discarding of queue objects. Figure 14.7 is the UML class diagram for the NodeGroupPriQue class.

Listing 14.12 The NodeGroupPriQue class of objects, add method postponed

```
public class NodeGroupPriQue extends Object implements PriQue
{
   private Node itsFirst = new Node (null, null);
   private Prioritizer itsTest;


   public NodeGroupPriQue (Prioritizer givenTest)
   {  super();
      itsTest = givenTest;
   }  //======================

   public boolean isEmpty()
   {  return itsFirst.itsData == null;
   }  //======================

   public Object get()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      return itsFirst.itsData.peekFront();
   }  //======================

   public Object remove()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      Object valueToReturn = itsFirst.itsData.dequeue();
      if (itsFirst.itsData.isEmpty())
         itsFirst = itsFirst.itsNext;  // lose the queue
      return valueToReturn;
   }  //======================


   private static class Node extends Object
   {  private QueueADT itsData;
      private Node itsNext;

      public Node (QueueADT data, Node next)
      {  super();
         itsData = data;
         itsNext = next;
      }
   }  //======================
}
```

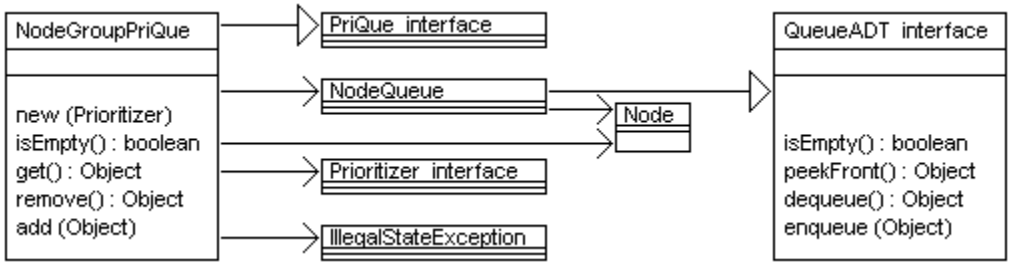


Figure 14.7 UML class diagram of the NodeGroupPriQue class

The add method

To add a non-null element, you need to search for the first Node that contains a queue whose elements do not have higher priority than the given element. Of course, you do not go beyond the last Node in the linked list, which is the trailer node. So the condition for the while-statement is the following:

```
p.itsData != null && itsTest.prior (p.itsData.peekFront(), ob)
```

By contrast, the while-condition for NodeOutPriQue was as follows, because you needed to go past not only elements of higher priority but also those of the same priority that had been on the linked list longer:

```
p.getData() != null && ! itsTest.prior (ob, p.getData())
```

Once you find the right Node p, see if it has a queue of elements with the same priority as the element you are supposed to add. If not (because the Node is empty or it has a queue of elements with lower priority), insert a new empty queue just before the queue in the Node you have found. In either case, you should now `enqueue` the given element onto that queue. This coding is in Listing 14.13.

Listing 14.13 The add method for NodeGroupPriQue

```
public void add (Object ob)
{  if (ob != null)
   {  Node p = this.itsFirst;
      while (p.itsData != null
             && itsTest.prior (p.itsData.peekFront(), ob))
      {  p = p.itsNext;
      }
      if (p.itsData == null
             || itsTest.prior (ob, p.itsData.peekFront()))
      {  p.itsNext = new Node (p.itsData, p.itsNext);
         p.itsData = new NodeQueue();
      }
      p.itsData.enqueue (ob);
   }
}  //======================
```

Exercise 14.50 List the changes that Listing 14.12 would require if the `itsData` field of the Node class were declared as Object rather than as QueueADT.
Exercise 14.51* Write out the internal invariant for NodeGroupPriQues.
Exercise 14.52* Revise the Node class for Listing 14.12 to have a third instance variable `itsKey`, which is any element that is or has been on the queue stored in `itsData`. Use this revision to rewrite the add method to execute faster.

14.8 Sorting And Big-oh Performance

Postpone this section if you did not yet cover Chapter Thirteen on sorting. The performance of the linked list implementations of stacks and queues (NodeStack and NodeQueue) is big-oh of 1 for both adding and removing an element, since neither loops nor recursion is involved. The performance of the standard array implementation of stacks (ArrayStack) is also big-oh of 1 for pushing and popping. However, the array implementations for queues are worse, because of the loops involved, except for the "avoid-all-moves" implementation described (but not coded) at the end of Section 14.2.

Big-oh for priority queue operations

Insertion of one element into an ArrayOutPriQue or NodeOutPriQue object has a worst-case execution time that is big-oh of N, where N is the number of items already in the data structure. Removal is quite fast at big-oh of 1. Insertion of one element into an ArrayInPriQue or NodeInPriQue object has a worst-case execution time that is big-oh of 1, but worst-case execution time for removal is big-oh of N.

A priority queue can be used to sort a large number of values. If you add them one at a time to the priority queue without ever calling `remove`, and then remove them one at a time until the priority queue is empty, they come out in the order determined by `prior`. If `prior` is implemented to mean the same as a negative `compareTo` value, they come out in ascending order. For the ArrayOutPriQue and NodeOutPriQue implementations, this is basically the InsertionSort logic, whose execution time is big-oh of N^2. For the ArrayInPriQue and NodeInPriQue implementations, this is basically the SelectionSort logic, whose execution time is also big-oh of N^2.

Implementing a priority queue as a heap

You could use the HeapSort (Section 13.7) as the basis for a priority queue that executes much faster. Inserting into the heap should have an execution time that is big-oh of log(N), where N is the number of elements currently in the priority queue. And removal from the heap should also have an execution time that is big-oh of log(N). This is a great improvement over the InsertionSort and SelectionSort implementations.

An array implementation HeapPriQue typically uses a partially-filled array (instance variables `itsItem` and `itsSize`). You organize the data structure as a max-heap, so that the value with highest priority is in `itsItem[0]`. Whenever you add another element to the priority queue, you start at `itsItem[itsSize]` and have it "sift up" towards index 0 (similar to the `putInHeap` method of Listing 13.6). Whenever you remove an element from the priority queue, you remove it from `itsItem[0]`, take `itsItem[itsSize-1]` from the rear part of the array, and have that element "sift down" starting at `itsItem[0]` (similar to the `adjust` method of Listing 13.6).

MergeSort of a linked list

Most sorting algorithms for arrays can be coded for linked lists as well. Clearly, the logic in NodeOutPriQue can easily be rewritten to provide a linked list InsertionSort and the logic in NodeInPriQue can easily be rewritten to provide a linked list SelectionSort. We next develop the MergeSort for linked lists.

A method that accepts a linked list as its parameter and returns the sorted form might have this heading (`size` is the number of elements in the linked list):

```
public static Node sorted (Node item, int size)
```

For the MergeSort logic, we are to divide the list into two lists of equal size (or differing by 1 if necessary), sort each one separately, and then merge them back together into one sorted list. This can be done without using any extra space (in contrast to arrays) except for a single extra Node we call `result`.

First, we return the `item` list immediately if it has less than two elements, since it is already sorted. Otherwise, we run half-way down the list and set `end` to the Node at the half-way point. The second half of the list starts in `end.getNext()`. We break the list into the two parts, call the `sorted` method for each part, and then call a private `merged` method to merge the two sorted lists into one long sorted list. The coding is in the upper part of Listing 14.14.

Listing 14.14 The MergeSort logic for a linked list

```
public static Node sorted (Node item, int size)// independent
{  if (size < 2)
      return item;
   int halfSize = size / 2;
   Node end = item;
   for (int k = 1;  k < halfSize;  k++)
      end = end.getNext();
   Node secondHalf = end.getNext();
   end.setNext (null);
   return merged (sorted (item, halfSize),
                  sorted (secondHalf, size - halfSize));
}  //======================

private static Node result = new Node (null, null);

private static Node merged (Node one, Node two)
{  Node end = result;  // last Node of completely sorted list
   while (one != null && two != null)
   {  if (((Comparable) one.getData())
                     .compareTo (two.getData()) > 0)
      {  end.setNext (one);
         one = one.getNext();
      }
      else
      {  end.setNext (two);
         two = two.getNext();
      }
      end = end.getNext();
   }
   end.setNext (one == null  ?  two  :  one);
   return result.getNext();
}  //======================
```

In this `merged` method, `one` and `two` denote the two sorted linked lists to be combined and placed on the completely sorted `result` list. It makes the coding simpler if we put a "dummy" Node containing no data on the `result` list to start. We add Nodes from `one` and `two` to the end of the `result` list, depending on the comparison of the data values on the two lists. When we exit the method, we return `result.getNext()`, which will be the actual completely sorted list without the dummy **header node**.

So the `merged` logic repeatedly looks at the first Node on each of `one` and `two`; whichever contains the smaller data is attached to the end of the `result` list and detached from its own list (that is, detached from the `one` list or the `two` list). When one of the lists runs out of Nodes, the other list is attached to the end of the `result` list and the completely sorted list is returned. This coding is in Listing 14.14.

The bucket sort and the radix sort

The NodeGroupPriQue implementation suggests some additional sorting methods that can only be used if each element has a non-negative integer code that gives its priority, obtained perhaps by the method call `anElement.getCode()`. Then you could keep an array of FIFO queues indexed by priority, e.g., `itsItem[k]` is a queue of all elements with the priority number `k`. Putting each element into the data structure is a big-oh of 1 operation using the following statement:

```
itsItem[ob.getCode()].enqueue (ob);
```

After you put all the elements in this data structure, remove them one at a time to get them in order of priority. This is called a **bucket sort**. If the range of priority values is too large (sorting by social security number would need an array with a billion components), you can use a **radix sort** instead: (a) Put each element in one of ten queues based on the last digit of its priority code. (b) Combine the ten queues into one long linked list, then put each element from that list in one of ten queues based on the next-to-last digit of its priority code. (c) Repeat for each digit in order right-to-left, which eventually gets the values sorted in order of priority.

Exercise 14.53 Rewrite the `merged` method to not use a header node.
Exercise 14.54* Explain why the ArrayOutPriQue implementation of a priority queue keeps the highest-priority element at the largest index in the array, but the HeapPriQue implementation keeps it at index 0, the smallest index in the array.
Exercise 14.55* Write the `add` method for the HeapPriQue class described here.
Exercise 14.56* Write the `remove` method for the HeapPriQue class described here.
Exercise 14.57** Revise the `merged` method on the precondition that `one`'s first Node contains the smallest data value. Run down the `one` list, inserting Nodes from `two`'s list wherever appropriate, then return the `one` list. Rewrite the `sorted` method to use this new `merged` method. Is this a faster implementation of the MergeSort logic?

14.9 External Sorting: File Merge Using A Priority Queue

Sometimes we have so much data to put in sorted order that we cannot fit it all into RAM. Say it is stored in a hard-disk file and there 2 million records to sort (a record is the set of current values of instance variables of an object). Our problem is to produce a new hard-disk file (we will call it SORTED.DAT) that contains all of those records in increasing order of IDs (using the `compareTo` method for Comparable values).

The piles-of-files algorithm

It is desirable to do as much of the sorting as possible in RAM, so a reasonable first step is to read as many values as we can into RAM, sort them, and write them out to a file. Then we read some more and write those to another file, etc. For instance, if we can handle 10,000 records at a time in RAM, we could end up with our original 2 million records in 200 different files. Then we could merge them together into one large file as shown in the accompanying design block (`numFiles` is 200 for this example).

STRUCTURED NATURAL LANGUAGE DESIGN for merging 200 files

1. Read the first record from each file into the corresponding component of an array of `numFiles` Comparable objects. Call the array `item`.
2. Find the smallest of those array components. Say it is at index `k`.
3. Write `item[k]` to the SORTED.DAT file.
4. If file number `k` is now empty, delete its entry from the array, otherwise get another value from the corresponding file `k` to go in `item[k]`.
5. Repeat steps 2 through 4 until done.

Total execution time is what is required for:
(a) the internal sorting, 10,000 at a time, plus
(b) reading and writing 2 million records 2 times each, and also
(c) making 199 comparisons 2 million times for a total of 398 million comparisons.
That last calculation generalizes to `N * (N / 10000)` for N records, so overall this is a big-oh of N-squared algorithm.

Object design

This design block is far too specific. We should not be making a commitment to unsorted arrays of values at this point in the design, or to a specific kind of file. To start with, we should just require some kind of object that can provide the next object in a sequential file when asked or accept a new object to add to the end of the file. A reasonable object design is the following:

```
public class ObjectFile  // for generic files of objects
{   // Open the file of that name for output; open a temporary file if the name is "".
    public ObjectFile (String name)
    // Add ob to the end of the file.
    public void writeObject (Object ob)
    // Change over to providing input, not output.
    public void openForInput()
    // Retrieve the next available object in the file.
    public Object readObject()
    // Switch back to providing output, not input.
    public void openForOutput()
    // Tell whether the input file has no more values.
    public boolean isEmpty()
}
```

We can then leave the details of how this is done in terms of the Sun standard library for the coding of this class's methods. You might want to look into Sun's ObjectInputStream class and serialization for a good way to implement this class.

For the process of sorting the data 10,000 units at a time (or whatever is appropriate), we basically need an object that provides two services: It can read 10,000 or so values from a given ObjectFile that has been opened for input; and it can write those 10,000 or so values in order to a given ObjectFile that has been opened for output. It could be as follows:

```
public class ObjectFileSorter  // for sorters of ObjectFiles
{   // Create the object capable of holding max values.
    public ObjectFileSorter (int max)
    // Read max values from the given file, except stop reading at the end of the file.
    public void readManyFromFile (ObjectFile file)
    // Write all values you have to the given file in increasing order using compareTo.
    public void writeManyToFile (ObjectFile file)
}
```

The upper part of Listing 14.15 (see next page) shows the structure of the object that converts one very large unsorted file to a very large sorted file. For the constructor, you supply the names of the unsorted file and the sorted file. Both are initially open for output (according to the specifications for the ObjectFile class). The unsorted file has to be opened for input.

The `makeSortedFiles` method in the middle part of Listing 14.15 creates an ObjectFileSorter object from a numeric parameter that tells how many objects you want to have in the sorter at one time. Then it repeatedly reads 10,000 (or whatever) values from the unsorted file, writes them in sorted order to a temporary output file, and finally adds the temporary file along with its first value to a data structure named `itsItem`.

For the process of merging 200 files (or however many we have), we need a kind of object that can store pairs consisting of a file plus the next available value from that file. When we get a value from this object, we want to receive the pair with the smallest value using the `compareTo` method. The ArrayOutPriQue class is fine for this purpose. Note that it can also do most of the task required of an ObjectFileSorter:

```
public class ArrayOutPriQue // for producing values in order
{   // Create the object that depends the compareTo method to sort values.
    public ArrayOutPriQue()
    // Put the given object in the data structure.
    public void add (Object ob)
    // Remove the smallest value in the data structure.
    public Object remove()
    // Tell whether the data structure has no more values.
    public boolean isEmpty()
}
```

Using only four files

A prime difficulty with this piles-of-files algorithm is that many systems do not allow you to keep more than a dozen or so files open at any one time. So we next present a method that only uses four files.

Stage 1 (for the `makeSortedFiles` method) Write the sorted groups of 10,000 (or whatever) alternately to just 2 files. That is, the first group goes into file `one`, the second into file `two`, the third into `one` again, the fourth into `two`, the fifth into `one`, the sixth into `two`, etc. So you end up with 100 groups of 10,000 in each of the 2 files.

Listing 14.15 File merge with piles of files

```
import ObjectFile;  import ObjectFileSorter;  import PriQue;

public class ManyFilesMerger
{
   private ObjectFile itsInFile;  // the original unsorted input
   private ObjectFile itsOutFile; // the final sorted output
   private ArrayOutPriQue itsItem = new ArrayOutPriQue();


   public ManyFilesMerger (String inf, String outf)
   {  super();
      itsInFile = new ObjectFile (inf);   // for output
      itsInFile.openForInput();           // but we need input
      itsOutFile = new ObjectFile (outf); // for output
   } //============================================

   public void makeSortedFiles (int maxToSort)
   {  ObjectFileSorter sorter = new ObjectFileSorter (maxToSort);
      while ( ! itsInFile.isEmpty())
      {  ObjectFile tempFile = new ObjectFile (""); // for output
         sorter.readManyFromFile (itsInFile);
         sorter.writeManyToFile (tempFile);
         tempFile.openForInput();
         itsItem.add (new Pair (tempFile.readObject(),tempFile));
      }
   } //============================================

   public void mergeFiles()
   {  while ( ! itsItem.isEmpty())
      {  Pair p = (Pair) itsItem.remove();
         itsOutFile.writeObject (p.itsData);
         if (! p.itsFile.isEmpty())
         {  p.itsData = p.itsFile.readObject();
            itsItem.add (p);
         }
      }
   } //============================================

   private static class Pair implements Comparable
   {  public Object itsData;
      public ObjectFile itsFile;

      public Pair (Object data, ObjectFile file)
      {  super();
         itsData = data;
         itsFile = file;
      }

      public int compareTo (Object ob)
      {  return ((Comparable) this.itsData).compareTo
                     (((Pair) ob).itsData);
      }
   } //============================================
}
```

It will be difficult to make use of this data unless you can tell where one group of 10,000 ends and the next begins. So you need a special object value, different from any other, that you can use to mark the boundary between groups. Call this value `itsSentinel`. You will also find it convenient to have exactly the same number of groups in each of the two files; so if the last group goes in file `one`, just write `itsSentinel` again to `two`. That makes it an empty group of sorted values. The implementation for this part of the algorithm is in the upper and middle parts of Listing 14.16 (see next page).

Stage 2 (for the `mergeFiles` method) is the merging process:

1. Open files `one` and `two` for input and create two additional files `three` and `four` for output.
2. Combine the first 10,000 from file `one` with the first 10,000 from file `two` and write the resulting sorted group of 20,000 to file `three`. This requires less than 20,000 comparisons.
3. Combine the second 10,000 from file `one` with the second 10,000 from file `two` and write the resulting sorted group of 20,000 to file `four`. Again, you only need less than 20,000 comparisons.
4. Repeat steps 2 and 3 alternately until files `one` and `two` are empty. Now you have 50 groups of 20,000 in each of the two files `three` and `four`.
5. Open files `one` and `two` for output and files `three` and `four` for input.
6. Combine the first 20,000 from file `three` with the first 20,000 from file `four`; write those 40,000 to `one`.
7. Combine the second 20,000 from file `three` with the second 20,000 from file `four`; write them to file `two`.
8. Repeat steps 6 and 7 alternately until files `three` and `four` are empty. Now you have 25 groups of 40,000 in each of the files `one` and `two`.

Surely you can see where this is going. After the two passes through the data described above, you have 6 more passes to get it down to one completely sorted file of 2 million in just one file. You then write all of its values to the output file (except for the sentinel value at the end, of course). Note that you will occasionally get a group left over that has no group to be merged with (when you have an odd number of groups), in which case you just copy it into the appropriate file. The bottom part of Listing 14.16 contains this logic, except that the key merging logic is left as an exercise.

The total execution time for this algorithm is what is required for:
(a) the internal sorting, plus
(b) reading and writing 2 million records 9 times each, and also
(c) making 2 million comparisons 8 times each for a total of 16 million comparisons.
The 9 and the 8 in this analysis are log2 (200) + 1 and log2(200). This generalizes to `N * log2(N / 10000)` for N records, so overall this is a big-oh of N * log(N) algorithm. But in real-life situations, it is slower than the piles-of-files method described first, since reading and writing disk records is excruciatingly slow.

Exercise 14.58 The ManyFilesMerger object in Listing 14.15 expects that a client class will call first the `makeSortedFiles` method and, directly after that, the `mergeFiles` method. What happens if a client class calls `mergeFiles` first?
Exercise 14.59 Still referring to the situation in the preceding exercise, what happens if a client class calls `makeSortedFiles` twice in a row without calling `mergeFiles`?
Exercise 14.60 Modify Listing 14.16 so that no client class can call `mergeFiles` before it calls `makeSortedFiles` and no client class can call either of those methods twice in a row. Hint: Add a boolean instance variable that tells whether `makeSortedFiles` has been called without an immediately following call of `mergeFiles`; use it appropriately.
Exercise 14.61** Write the coding for the `mergeToOneFile` method in Listing 14.16.

Listing 14.16 File merge with just four files

```
import ObjectFile;  import ObjectFileSorter;  import PriQue;

public class FourFilesMerger
{
   private ObjectFile itsInFile;  // the original unsorted input
   private ObjectFile itsOutFile; // the final sorted output
   private ObjectFile one, two;   // two scratch files
   private Object itsSentinel; // sentinel value to mark the end


   public FourFilesMerger (String inf, String outf, Object sent)
   {  super();                                                    // 1
      itsInFile = new ObjectFile (inf);   // for output          // 2
      itsInFile.openForInput();       // but we need input       // 3
      itsOutFile = new ObjectFile (outf); // for output          // 4
      itsSentinel = sent;                                       // 5
   } //==============================================

   public void makeSortedFiles (int maxToSort)
   {  ObjectFileSorter sorter = new ObjectFileSorter (maxToSort);
      one = new ObjectFile ("");          // for output          // 7
      two = new ObjectFile ("");          // for output          // 8
      while ( ! itsInFile.isEmpty())                            // 9
      {  sorter.readManyFromFile (itsInFile);                   // 10
         sorter.writeManyToFile (one);                          // 11
         one.writeObject (itsSentinel);                         // 12
         if ( ! itsInFile.isEmpty())                            // 13
         {  sorter.readManyFromFile (itsInFile);                // 14
            sorter.writeManyToFile (two);                       // 15
            two.writeObject (itsSentinel);                      // 16
         }                                                      // 17
      }                                                         // 18
   } //==============================================

   public void mergeFiles()
   {  one = mergeToOneFile (one, two, new ObjectFile (""),  // 19
                                      new ObjectFile (")); // 20
      Object data = one.readObject();                        // 21
      while ( ! one.isEmpty())                               // 22
      {  itsOutFile.writeObject (data);                      // 23
         data = one.readObject();                            // 24
      }                                                      // 25
   } //==============================================

   /** Return a file containing all the values in increasing
    *  order, plus a sentinel at the end. */

   private ObjectFile mergeToOneFile (ObjectFile in1,
           ObjectFile in2, ObjectFile out1, ObjectFile out2)
   {  return null;  // left as an exercise
   } //==============================================
}
```

*14.10 About Stack And Deprecated Vector (*Sun Library)*

The **java.util.Vector** class was the original class designed by Sun Microsystems, Inc., to create and modify a collection of Objects. It has been replaced by ArrayList, and its use is now **deprecated**. This means that programmers are strongly encouraged to use ArrayList or something else from the Collection or Map hierarchy in new software they write, but Vector will be retained in the Sun standard library so that existing software will continue to function correctly.

You should be aware of the Vector class because, if you find occasion to maintain older software written in Java, you will often find it using the Vector class. Vector has been "retrofitted" to have all of the methods required by the List interface. But it still provides the following antiquated methods:

- `new Vector()` creates an empty Vector.
- `new Vector(someInt)` creates an empty Vector with a capacity of `someInt`. The capacity will increase if enough elements are added to the Vector.
- `someVector.elementAt(indexInt)` is `get(indexInt)`.
- `someVector.setElementAt(ob, indexInt)` is `set(ob, indexInt)`.
- `someVector.addElement(ob)` is `add(ob)`.
- `someVector.removeElementAt(indexInt)` is `remove(indexInt)`.
- `someVector.removeAllElements()` is `clear()`.
- `someVector.insertElementAt(ob, indexInt)` is `add(ob, indexInt)`.
- `someVector.firstElement()` returns the element at index 0.
- `someVector.elements()` returns an Enumeration object, which is the earlier version of Iterator. Enumeration is now a deprecated class.

Stack objects (from java.util)

The Stack class is a subclass of Vector. It has the following methods. Both `pop` and `peek` throw a java.util.EmptyStackException if the Stack is empty:

- `new Stack()` creates an empty Stack.
- `someStack.push(someObject)` adds the object to the top of the Stack.
- `someStack.pop()` removes and returns the object on top of the Stack.
- `someStack.peek()` returns the object that is on top of the Stack.
- `someStack.empty()` tells whether the stack contains no objects.
- `someStack.search(someObject)` returns 1 if the object is on top, 2 if it is second, 3 if it is third, etc. It returns -1 if the object is not in the Stack.

14.11 Review Of Chapter Fourteen

- Stacks, queues, and priority queues are data structures that allow you to add an element, remove a particular element, see if they are empty, or see what you would get if you removed an element. The particular element you get depends on the structure: A stack gives you the element that has been there the shortest period of time, a queue gives you the element that has been there the longest period of time, and a priority queue gives you the element of highest priority (in case of a tie, the element that has been there longest).
- This book defines three similar interfaces that all have an `isEmpty()` method plus a method to add an element, a method to remove the required element, and a method to see what element would be removed. The StackADT interface has methods `push(ob)`, `pop()`, and `peekFront()`. The QueueADT interface has methods `enqueue(ob)`, `dequeue()`, and `peekFront()`. The PriQue interface has methods `put(ob)`, `remove()`, and `get()`.
- Postfix algebraic notation puts each binary operator directly after the two operands. As a consequence, it is never necessary to parenthesize the expression; a legal postfix expression can only be interpreted in one way. Evaluation of a postfix expression is most easily done with a stack.

Answers to Selected Exercises

14.1
```
public static Object removeSecond (StackADT stack)
{   if (stack.isEmpty())
        return null;
    Object first = stack.pop();
    Object valueToReturn = stack.isEmpty()  ?  null : stack.pop();
    stack.push (first);
    return valueToReturn;
}
```

14.2
```
public static void removeDownTo (StackADT stack, Object ob)
{   if (ob == null)
        while ( ! stack.isEmpty() && stack.peekTop() != null)
            stack.pop();
    else
        while ( ! stack.isEmpty() && ! ob.equals (stack.peekTop()))  // not the opposite order
            stack.pop();
}
```

14.3
```
public static Object removeSecond (QueueADT queue)
{   if (queue.isEmpty())
        return null;
    Object first = queue.dequeue();
    Object valueToReturn = queue.isEmpty()  ?  null : queue.dequeue();
    Object endMarker = new Double (0.0);  // anything brand-new works here
    queue.enqueue (endMarker);
    queue.enqueue (first);
    while (queue.peekFront() != endMarker)
        queue.enqueue (queue.dequeue());
    queue.dequeue();  // remove the endMarker, so first is now at the front
    return valueToReturn;
}
```

14.8
```
public int size()
{   return itsRear - itsFront + 1;
}
```

14.9
```
public String toString()
{   String valueToReturn = "";
    for (int k = itsFront;  k <= itsRear;  k++)
        valueToReturn += '\t' + itsItem[k].toString();
    return valueToReturn;
}
```

14.10 The declaration of itsTop should be: `private int itsTop = -1;`
Replace itsSize by itsTop+1 everywhere else in the coding except where you apply ++ or --.

14.11
```
public void removeBelow (Object ob)
{   int spot = itsSize - 1;
    if (ob == null)
        while (spot > 0 &&  itsItem[spot] != null)
            spot--;
    else
        while (spot > 0 &&  ! ob.equals (itsItem[spot]))
            spot--;
    if (spot > 0)
    {   for (int k = spot;  k < itsSize;  k++)
            itsItem[k - spot] = itsItem[k];
        itsSize -= spot;
    }
}
```
14.12
```
public boolean equals (Object ob)
{   if ( ! (ob instanceof ArrayStack) || ((ArrayStack) ob).itsSize != this.itsSize)
        return false;
    ArrayStack given = (ArrayStack) ob;   // for efficiency
    for (int k = 0;  k < this.itsSize;  k++)
        if ( ! this.itsItem[k].equals (given.itsItem[k]))
            return false;
    return true;
}
```
14.19
```
public void dup()
{   if (isEmpty())
        throw new IllegalStateException ("stack is empty");
    itsTop = new Node (itsTop.getData(), itsTop);
}
```
14.20
```
public void swap2and3()
{   if (itsTop != null && itsTop.getNext() != null && itsTop.getNext().getNext() != null)
    {   Node second = itsTop.getNext();
        Object saved = second.getData();
        second.setNode (second.getNext().getData(), second.getNext());
        second.getNext().setNode (saved, second.getNext().getNext());
    }
}
```
14.21
```
public void enqueue (Object ob)
{   if (isEmpty())
    {   itsFront = new Node (ob, null);
        itsRear = itsFront;
    }
    else
    {   itsRear.setNext (new Node (ob, null));
        itsRear = itsRear.getNext();
    }
}
```
14.22
```
public void add (NodeQueue queue)
{   if ( ! queue.isEmpty())
    {   this.itsRear.setNext (queue.itsFront);
        this.itsRear = queue.itsRear;
        queue.itsFront = null;
    }
}
```
14.23 The answers are 6 and 1.

14.29
```
public void add (Object ob)
{   if (ob != null)
    {   if (itsSize == itsItem.length)
        {   Object[] toDiscard = itsItem;
            itsItem = new Object [itsItem.length * 2];
            for (int k = 0;  k < itsSize;  k++)
                itsItem[k] = toDiscard[k];
        }
        int k = itsSize;
        for ( ;  k > 0 && ! itsTest.prior (ob, itsItem[k - 1]);  k--)
            itsItem[k] = itsItem[k - 1];
        itsItem[k] = ob;
    }
}
```
14.30 It is the same coding as for the remove method in Listing 14.9 except you replace the last four statements by the following:
return itsItem[best];.

14.31 Replace the next-to-last statement in the remove method by the following:

```
for (int k = best;  k < itsSize;  k++)
    itsItem[k] = itsItem[k + 1];
```

14.33 Replace the last three statements of NodeInPriQue's remove method by the following two:

```
holdsBest.setNode (holdsBest.getNext().getData(), holdsBest.getNext().getNext());
return valueToReturn;
```

14.34 In the add method in Listing 14.10, replace the while-loop and the two statements after it by:

```
while (p.getData() != null && itsTest.prior (p.getData(), ob))
    p = p.getNext();
if (p.getData() == null || itsTest.prior (ob, p.getData()))
    p.setNode (ob, new Node (p.getData(), p.getNext()));
```

14.35

```
public int size()
{   int count = 0;
    for (Node p = itsFirst;  p.getNext() != null;  p = p.getNext())
        count++;
    return count;
}
```

14.36

```
public String toString()
{   String valueToReturn = "";
    for (Node p = itsFirst;  p.getNext() != null;  p = p.getNext())
        valueToReturn += '\t' + p.getData().toString();
    return valueToReturn;
}
```

14.37

```
public void removeBelow (Object ob)
{   Node p = itsFirst;
    while (p != null && ! ob.equals (p.getData()))
        p = p.getNext();
    if (p != null)
        p.setNext (null);
}
```

14.38

```
public void removeAbove (Object ob)
{   while (itsFirst.getData() != null && itsTest.prior (itsFirst.getData(), ob))
        itsFirst = itsFirst.getNext();
}
```

14.39

```
public void add (Object ob)
{   if (ob != null)
        if (itsFirst.getData() != null && ! itsTest.prior (ob, itsFirst.getData()))
            itsFirst.setNext (new Node (ob, itsFirst.getNext()));
        else
            itsFirst = new Node (ob, itsFirst);
}
```

14.50 In get write: return ((QueueADT) itsFirst.itsData).peekFront();
In remove write: Object valueToReturn = ((QueueADT) itsFirst.itsData).dequeue();
Also in remove write: if (((QueueADT) itsFirst.itsData).isEmpty())

14.53 Change the last statement to be return result. Then insert the following at the beginning:

```
Node result = ((Comparable) one.getData()).compareTo (two.getData()) < 0  ?  one  :  two;
if (result == one)
    one = one.getNext();
else
    two = two.getNext();
```

14.58 Nothing happens if mergeFiles is called first, since itsItem is empty until after makeSortedFiles executes.

14.59 In effect, nothing happens if makeSortedFiles is called twice. True, on the second time another new ObjectFileSorter object is created. But itsInfile is empty (caused by the preceding call of this method), so the method terminates immediately and the newly-created object is garbage collected.

15 Collections And Linked Lists

Overview

This chapter requires that you have a solid understanding of arrays (Chapter Seven) and some idea of recursion (Section 5.9 or 13.4). It is preferable that also you study the first three sections of Chapter Fourteen (stacks and queues implemented with linked lists).

- Section 15.1 presents a software context in which you need programs that work with large collections of data.
- Section 15.2 introduces the official (Sun standard library) Collection interface and an array-based implementation of the unmodifiable form.
- Sections 15.3-15.5 describe linked lists and their application to implementing the non-modifying methods of the Collection interface, in some cases recursively.
- Sections 15.6-15.7 define the Iterator interface and show how it can be implemented in both the array and linked list forms.
- Section 15.8 implements the methods that modify a Collection object.
- Sections 15.9-15.10 discuss an implementation of ListIterator and List using doubly-linked lists.

This chapter should help you develop a strong understanding of the Collection and List interfaces, strengthen your abilities in working with arrays, and further develop facility with linked lists. In everyday programming, when you want to use a Collection or List object, you will tend to choose the "built-in" ArrayList class or some other standard library implementation rather than one you build yourself (in accordance with the "don't reinvent the wheel" principle). But you can only learn about linked lists by using them to code various methods, so that is what we do here.

15.1 Analysis And Design Of The Inventory Software

You have a client who needs a number of programs that read in two files of data and then process them. One file lists all the retail items that the company currently has on hand (its inventory). Another file lists all the retail items that the company has purchased and, according to its records, not yet sold. Obviously, the two lists should match up. However, discrepancies are common.

For the first such program, it is sufficient to tell whether the two lists are exactly equal and, if not, whether the inventory file at least contains all the retail items purchased (so the store has not had any retail items stolen). If even that is not true, then you need to say how many retail items are in each list. A reasonable plan for the program, developed after a bit more discussion with the client to clear up some details, is shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN of the main logic

1. Read the "inven.dat" file into an appropriate object named `inventory`.
2. Read the "purch.dat" file into an appropriate object named `purchased`.
3. If one list has the same elements in the same order as the other list then...
   Print the message "a perfect match".
4. Otherwise, if the `inventory` list has no elements at all then...
   Print the message "we've been robbed".
5. Otherwise, if every element of the `purchased` list is in the `inventory` list then...
   Print the message "no losses and some gains".
6. Otherwise...
   Print the message "inventory has N and purchased has M"
   where N and M are the number of values in the respective lists.

Object design

We need a class of objects that store sequences of data items. We will call it the BasicSequence class. We will make it a Collection kind of class; Collection is an interface in the Sun standard library that specifies the names of thirteen methods, how they are called, and what they should accomplish. The Collection methods provide all the capabilities that you need for this program. The next section describes them in detail. Listing 15.1 shows how some of them are used to solve one problem the client has.

Listing 15.1 The Inventory application program

```
public class Inventory
{
   /** Read 2 files of data and make comparisons between them. */

   public static void main (String[] args)
   {  BasicSequence inventory;
      BasicSequence purchased;
      try
      {  inventory = new BasicSequence ("inven.dat");
         purchased = new BasicSequence ("purch.dat");
      }catch (java.io.IOException e)
      {  throw new RuntimeException ("A file is defective.");
      }

      if (inventory.equals (purchased))
         System.out.println ("a perfect match");
      else if (inventory.isEmpty())
         System.out.println ("we've been robbed");
      else if (inventory.containsAll (purchased))
         System.out.println ("no losses and some gains");
      else
         System.out.println ("inventory has " + inventory.size()
                  + " and purchased has " + purchased.size());
   }  //=====================
}
```

The BasicSequence constructor takes a String value as a parameter, opens the disk file of that name, and reads in the values from the file one line at a time. The lines from the file are stored in the BasicSequence object in the order they were read. If the file is not accessible, the constructor throws an IOException; this main method catches it and notifies the caller of the method (it throws a new Exception instead of terminating the program because this main method may be called from another method).

The effects of the other methods used in this program should be clear when compared with the design block. In any case, they are explained in detail in the next section. Figure 15.1 is a UML diagram for this Inventory class.

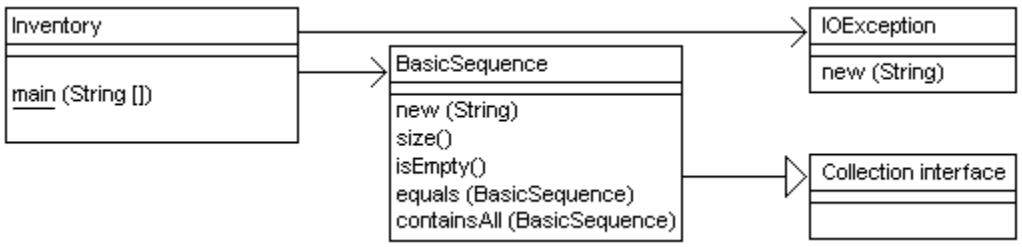


Figure 15.1 UML class diagram for Inventory

15.2 Implementing The Collection Interface With Arrays

A **sequence** is a collection of values in a particular order, called **elements** of the sequence. We define here the BasicSequence class with two constructors and several other methods, implemented using a partially-filled array. The sequence is allowed to have duplicates (two or more elements that are equal to each other), but it is not allowed to have an element be `null`.

The Collection interface

The Sun standard library contains a number of interfaces for describing ways of structuring collections of data. The most fundamental one is the **Collection** interface (in the `java.util` package) for storing Object values. The BasicSequence class will be an implementation of the Collection interface when we finish adding enough methods. Listing 15.2 (see next page) gives the documentation for the Collection interface, with full descriptions of those methods that do not change the elements in the Collection.

People use a number of variants of Collection implementations. The main variants are as follows:

- Some variants do not allow `null` as an element; some do.
- Some variants do not allow duplicates (i.e., all elements must be different from each other); some do.
- Some variants do not allow changes in the number or ordering of the elements once they are set by a constructor; some do. Those that do not allow changes have the last six listed methods throw an UnsupportedOperationException.
- This description of Collection specifies that the order of the elements is important; some variants of Collection do not require this.
- Some variants restrict the class of the objects stored, e.g., only Comparable objects can be stored in certain collections.

The specification for the BasicSequence implementation of Collection is that ordering is important, modifications and duplicates are allowed, but nulls are not. This section discusses the coding of the first seven methods using an array. A later section discusses the Iterator interface so we can implement the `iterator` method. After that, we discuss implementations of the remaining six methods.

Implementing the BasicSequence constructors

The BasicSequence class implements the Collection class using a partially-filled array, very much like the WorkerList class in Chapter Seven. Specifically, each BasicSequence object has two instance variables `itsItem` and `itsSize`. The former is an array that is filled with Objects in components indexed 0 up to but not including the int value `itsSize`. The values in higher-indexed components are not relevant to the logic.

One BasicSequence constructor has the name of a disk file as its parameter. It is to read all the values from the disk file one at a time and put them in its array, starting from index 0. The first question is, how large do you make the array? The file could have 15 lines in it or 15,000. A reasonable approach is to start with an array of moderate size, say 100, and then replace it by one twice as large each time the one you have fills up. You first have to open the disk file with the specified name:

```
BufferedReader file = new BufferedReader
                      (new FileReader (fileName));
```

Listing 15.2 The Collection interface

```
/** The Collection interface specifies 13 method headings in
 * addition to those inherited from Object, e.g., equals. */

public interface Collection
{
   /** Return the number of elements in this sequence. */
   public int size();

   /** Tell whether this sequence has no elements in it. */
   public boolean isEmpty();

   /** Tell whether ob is an element of this sequence. */
   public boolean contains (Object ob);

   /** Tell whether every element of that sequence is
    *  somewhere in this sequence. */
   public boolean containsAll (Collection that);

   /** Return an array filled with the elements of this
    *  sequence in the same order. */
   public Object[] toArray();

   /** The same as the above if array.length < this.size().
    *  Otherwise fill the given array with the elements of the
    *  sequence in the same order and put a null value after
    *  those values if it will fit. */
   public Object[] toArray (Object[] array);

   /** Tell whether the two sequences have the same elements
    *  in the same order. */
   public boolean equals (Object ob);

   /** Return an iterator that goes through the elements of
    *  this sequence in an established order. */
   public Iterator iterator();

   //  The remaining six, to be described fully in Listing 15.10,
   //  are coded as follows when the Collection is unmodifiable:
   //  "throw new java.lang.UnsupportedOperationException();"
   public void clear();
   public boolean add (Object ob);
   public boolean addAll (Collection that);
   public boolean remove (Object ob);
   public boolean removeAll (Collection that);
   public boolean retainAll (Collection that);
}
```

Now you read one line at a time from the file into a String variable named perhaps `s`. When the value of `s` is `null`, you are at the end of the file and you can stop. Otherwise you copy `s` into the next available component `itsItem[itsSize]` and then increment `itsSize`. The coding for this constructor is in the upper part of Listing 15.3.

The other BasicSequence constructor just requires that you make the new `itsItem` an exact copy of the given one (the private `copyOf` method can be used for this as well as for doubling the size of an array, since it is written with an int parameter to specify whether the new array is twice the size of the old one or not). Then you record the size of the newly-constructed array. The coding is in the middle part of Listing 15.3.

Listing 15.3 The BasicSequence class of objects, parts left as exercises

```
import java.io.*;
import java.util.*;

public class BasicSequence extends Object implements Collection
{
   private Object[] itsItem;
   private int itsSize = 0;

   public BasicSequence (String fileName) throws IOException
   {  super();
      itsItem = new Object[100];
      BufferedReader file = new BufferedReader
                            (new FileReader (fileName));
      String s = file.readLine();
      while (s != null)
      {  if (itsSize == itsItem.length)
            itsItem = copyOf (itsItem, 2);
         itsItem[itsSize] = s;
         itsSize++;
         s = file.readLine();
      }
   }  //=====================

   private Object[] copyOf (Object[] given, int big)
   {  Object[] valueToReturn = new Object [given.length * big];
      for (int k = 0;  k < given.length;  k++)
         valueToReturn[k] = given[k];
      return valueToReturn;
   }  //=====================

   public BasicSequence (BasicSequence that)
   {  super();
      this.itsItem = copyOf (that.itsItem, 1);
      this.itsSize = that.itsSize;
   }  //=====================

   public boolean equals (Object ob)
   {  if ( ! (ob instanceof BasicSequence))
         return false;
      BasicSequence that = (BasicSequence) ob;
      if (that.itsSize != this.itsSize)
         return false;
      for (int k = 0;  k < that.itsSize;  k++)
         if ( ! this.itsItem[k].equals (that.itsItem[k]))
            return false;
      return true;
   }  //=====================
}
```

Implementing the equals method

The `equals` method tests whether a given BasicSequence parameter has the same elements in the same order as the executor. The parameter is of type Object, not BasicSequence, because you want this `equals` method to override the one for the Object class. So you first check that the parameter is in fact a BasicSequence kind of Object. The condition `x instanceof Y` tells whether `x` is an object whose class is Y or extends Y or implements Y. This condition is false when `x` is `null`.

The accompanying design block is a reasonable plan for solving this problem. The coding is in the lower part of Listing 15.3. The other methods of the BasicSequence class are left for exercises.

STRUCTURED NATURAL LANGUAGE DESIGN for equals

1. If the parameter is not a BasicSequence kind of object then...
   The parameter is not equal to the executor.
2. Otherwise, if the parameter does not have the same number of elements as the executor then...
   The parameter is not equal to the executor.
3. Otherwise, for each element of the parameter do the following...
   If the current element of the parameter is not equal to the corresponding element of the executor then...
   The parameter is not equal to the executor.
4. The parameter is equal to the executor if you get to this point in the logic.

It is standard procedure to have a constructor that has a Collection parameter, but you cannot write such a method until later in this chapter, when you learn what an Iterator is.

Exercise 15.1 Write the BasicSequence method `public int size()`.
Exercise 15.2 Write the BasicSequence method `public boolean isEmpty()`.
Exercise 15.3 Write the BasicSequence method `public boolean contains (Object ob)`.
Exercise 15.4 Write the BasicSequence method `public Object[] toArray()`.
Exercise 15.5* Write the BasicSequence method `public boolean containsAll (Collection that)`. You may throw an Exception if `that` is not a BasicSequence.
Exercise 15.6* Write the BasicSequence method `public Object[] toArray (Object[] array)`.
Exercise 15.7* Draw the UML class diagram for the Inventory class.

15.3 Linked Lists With A Nested Private Node Class

You may define one class X inside the body of another class with the modifier `static` in X's heading. This makes X a **nested class** of the other class. We define the NodeSequence class with two constructors analogous to those of BasicSequence, and with all the Collection methods, but we implement it as a **linked list of Nodes** with a nested private Node class. That is, we define the Node class within the body of the BasicSequence class. This keeps outside classes from directly changing the value in a Node belonging to a NodeSequence object.

The Node class

The Node class is defined in Listing 15.4. A Node object stores two pieces of information: a reference to a single piece of data (of type Object) and a reference to another Node object. A Node object's data is referenced by `itsData` and the Node that comes next after it in the linked list is referenced by `itsNext`. If for instance `p` refers to a particular Node, then `p.itsData` is the information at that position in the list and `p.itsNext` is the Node containing the information at the following position in the list. If, however, `p.itsData` is the last information in the list, then we normally set `p.itsNext` to `null` to indicate this.

Listing 15.4 The Node class

```
   private static class Node extends Object
       // nested in the NodeSequence class
   {
      public Object itsData;
      public Node itsNext;


      public Node (Object data, Node next)
      {  super();
         itsData = data;
         itsNext = next;
      }
   }  //=====================
```

The natural constructor is defined and no other Node methods. Since this is a private class in the NodeSequence class and it has public instance variables, the NodeSequence class can access the instance variables directly. However, the principle of encapsulation is not violated since no class outside the NodeSequence class can refer to the instance variables of a class that is declared privately inside another class. Note: Chapter Fourteen defined a Node class outside of any class and provided public methods to control access to its private variables. That way of defining the Node class is not better or worse than this one, just different. Both approaches maintain encapsulation.

Nodes in a NodeSequence object

A NodeSequence object will have an instance variable `itsFirst` for the first Node object on its list. Suppose a particular NodeSequence object has a linked list of two or more Nodes. Coding to print the two data values at the beginning of this list is as follows:

```
System.out.println (itsFirst.itsData.toString());
System.out.println (itsFirst.itsNext.itsData.toString());
```

Coding to add a new data value "long" between the first and second Nodes could be as follows (illustrated in Figure 15.2). Actually, this coding will work even if only one Node is in the linked list, but it will throw a NullPointerException if `itsFirst` has the value `null`:

```
Node newNode = new Node ("long", itsFirst.itsNext);
itsFirst.itsNext = newNode;
```

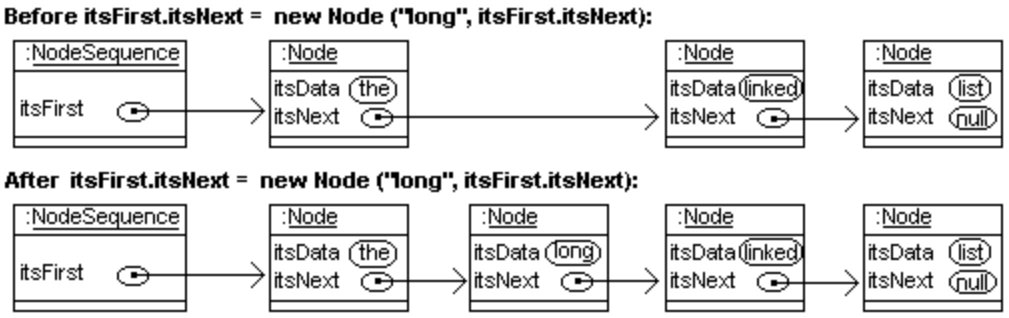


Figure 15.2 Inserting a new Node into a sequence of Nodes

If you want to add a new data value named `givenData` at the front of the list, you could use the following coding, which will work even if no Node at all is in the linked list:

```
itsFirst = new Node (givenData, itsFirst);
```

Looping through a linked list

The standard way you go through all the components in a partially-filled array one at a time, processing each one's data, is as follows:

```
for (int k = 0;  k < itsSize;  k++)
   processData (itsItem[k]);
```

This is precisely analogous to the standard way you go through all the nodes in a linked list one at a time, processing each one's data. Figure 15.3 shows the parallelism with the following coding:

```
for (Node p = itsFirst;  p != null;  p = p.itsNext)
   processData (p.itsData);
```

Components in an array	**Nodes in a linked list**
`k` indicates the current component	`p` indicates the current node
`k = 0` selects the first component	`p = itsFirst` selects the first node
`k == itsSize` if no more components	`p == null` if no more nodes
`k++` moves the indicator to the next component	`p = p.itsNext` moves the indicator to the next node
`itsItem[k]` is the data `k` indicates	`p.itsData` is the data `p` indicates

Figure 15.3 Parallelism of loops through arrays and loops through linked lists

The coding for the `contains` method in the BasicSequence array implementation is the following (as you saw in an exercise):

```
for (int k = 0;  k < itsSize;  k++)
   if (itsItem[k].equals (ob))
      return true;
return false;
```

It follows that the coding for the `contains` method in the NodeSequence linked list implementation is the analogous logic you can see in the upper part of Listing 15.5 (see next page). Compare it piece-by-piece with the array form. This is a form of the Some-A-are-B looping action: Return `true` as soon as you see a good one, return `false` after you looked everywhere and saw only bad ones.

The `containsAll` method requires a form of the All-A-are-B looping action: Return `false` as soon as you see a bad one, return `true` after you looked everywhere and saw only good ones. That logic is in the middle part of Listing 15.5. It throws a ClassCastException if the parameter is not a NodeSequence. This is unavoidable for now, since you do not know iterators, but it is fixed in an exercise in Section 15.6.

The `size` method tells how many elements the sequence has. So the coding has the standard loop heading and it uses the very common Count-cases looping action: Initialize a variable to 0 and then increment it once each time through the loop to see how many times the loop iterates.

Listing 15.5 The NodeSequence class of objects, part 1

```
/** A sequence of non-null values in a particular order. */

import java.io.*;
import java.util.*;

public class NodeSequence extends Object implements Collection
{
   private Node itsFirst = null;


   public NodeSequence()
   {  super();
   }  //=====================

   public boolean contains (Object ob)
   {  for (Node p = itsFirst;  p != null;  p = p.itsNext)
         if (p.itsData.equals (ob))
            return true;
      return false;
   }  //=====================

   public boolean containsAll (Collection that)
   {  for (Node p = ((NodeSequence) that).itsFirst;
                    p != null;  p = p.itsNext)
         if ( ! this.contains (p.itsData))
            return false;
      return true;
   }  //=====================

   public int size()
   {  int count = 0;
      for (Node p = itsFirst;  p != null;  p = p.itsNext)
         count++;
      return count;
   }  //=====================
}
```

Exercise 15.8 Write the NodeSequence method `public boolean isEmpty()`.
Exercise 15.9 Write a NodeSequence method `public int howManyEqual (Object ob)`: The executor tells how many of its elements are equal to `ob`.
Exercise 15.10* Write a NodeSequence method `public boolean allAreStrings()`: The executor tells whether every element is a String value. Hint: Use the `instanceof` operator.
Exercise 15.11* Write a NodeSequence method `public Object removeFirst()`: The executor removes and returns the first element in its sequence. It returns `null` if it is empty.
Exercise 15.12* Write a NodeSequence method `public void takeAway (Object ob)`: The executor removes all Objects from the front of the list down to but not including the first one that equals `ob`. Leave the NodeSequence empty if none are equal.

15.4 Implementing The Collection Interface With Linked Lists

The previous section began the coding of the NodeSequence class (Listing 15.5) with the "easy" methods. This section develops three more complex methods for that class. But first it is best to explicitly state the internal invariant for this class, i.e., the state of the NodeSequence object that every method preserves. It describes the connection between the user's abstract concept of the sequence of data values and the reality of Nodes.

Internal invariant for NodeSequences

- If the sequence does not contain any data values, `itsFirst` is `null`. Otherwise `itsFirst` refers to the first Node in a linked list of Nodes.
- For each Node `x` in that linked list, the value in `x.itsData` is one non-null data value in the abstract sequence of data values.
- The data values are in the linked list in the same order that they are in the abstract sequence of data values, with `itsFirst` containing the first one (if it exists).
- The Nodes in one NodeSequence are all different objects from those in any other.

Implementing the equals method

The coding for the `equals` method is not too much different from the coding for the BasicSequence `equals` in the earlier Listing 15.3. You make sure the parameter actually is a NodeSequence and then you cast it to make the coding easier to read and faster to execute. You should not check the sizes to see if they are equal, because that takes a significant amount of time with linked lists (by contrast, each BasicSequence object knows its size). Next you go through the list of all values in the parameter one at a time. Since the coding in the BasicSequence method is

```
for (int k = 0;  k < that.itsSize;  k++)
   if ( ! this.itsItem[k].equals (that.itsItem[k]))
      return false;
```

the analogous coding for the NodeSequence method should have two Node variables that progress through the two linked lists in tandem:

```
Node p = this.itsFirst;
for (Node q = that.itsFirst;  q != null;  q = q.itsNext)
{  if ( ! p.itsData.equals (q.itsData))
      return false;
   p = p.itsNext;
}
```

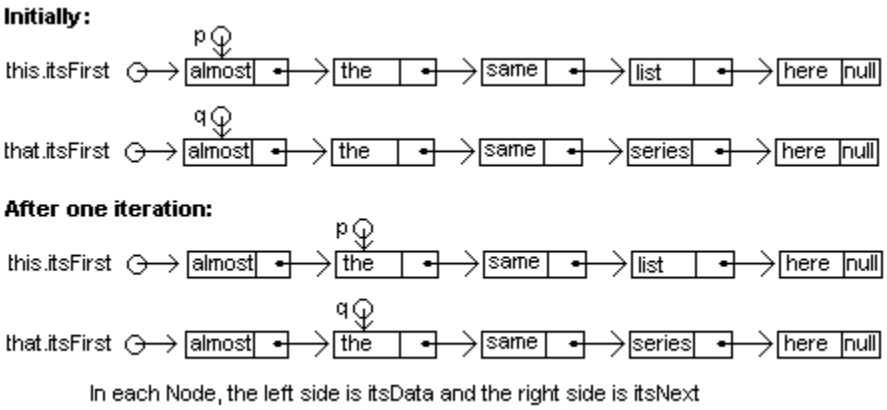


Figure 15.4 Two stages of execution of the equals method

Figure 15.4 shows how the two lists and these two position indicators look. However, you have not checked that the sizes of the two lists match, so it is possible that p could become `null` while the loop executes, which would throw a NullPointerException. You need a crash-guard; have the if-statement check `p == null` and, if so, return `false` without trying to evaluate `p.itsData`. This leads to the coding in the upper part of Listing 15.6. Note that successful completion of the loop does not guarantee equality; you have to verify that the executor's list ran out at the same time as the parameter's list.

Listing 15.6 The NodeSequence class of objects, part 2

```
// public class NodeSequence, 3 more of the methods

   public boolean equals (Object ob)
   {  if ( ! (ob instanceof NodeSequence))                    //1
         return false;                                        //2
      Node p = this.itsFirst;                                 //3
      Node q = ((NodeSequence) ob).itsFirst;                  //4
      for ( ;  q != null; q = q.itsNext)                      //5
      {  if (p == null || ! p.itsData.equals (q.itsData))     //6
            return false;                                     //7
         p = p.itsNext;                                       //8
      }                                                       //9
      return p == null;                                       //10
   }  //=====================

   public NodeSequence (String fileName) throws IOException
   {  super();                                                //11
      BufferedReader file = new BufferedReader                //12
                          (new FileReader (fileName));        //13
      String s = file.readLine();                             //14
      if (s != null)                                          //15
      {  this.itsFirst = new Node (s, null);                  //16
         Node previous = this.itsFirst;                       //17
         s = file.readLine();                                 //18
         while (s != null)                                    //19
         {  previous.itsNext = new Node (s, null);            //20
            previous = previous.itsNext;                      //21
            s = file.readLine();                              //22
         }                                                    //23
      }                                                       //24
   }  //=====================

   public NodeSequence (NodeSequence that)
   {  super();                                                //25
      if (that.itsFirst != null)                              //26
      {  this.itsFirst = new Node (that.itsFirst.itsData, null);
         Node previous = this.itsFirst;                       //28
         for (Node p = that.itsFirst.itsNext;                 //29
                   p != null;  p = p.itsNext)                 //30
         {  previous.itsNext = new Node (p.itsData, null);    //31
            previous = previous.itsNext;                      //32
         }                                                    //33
      }                                                       //34
   }  //=====================
```

Implementing the NodeSequence constructor that uses a file

The logic for constructing a new NodeSequence object out of input from a disk file is only mildly different from the corresponding BasicSequence constructor in the earlier Listing 15.3. After you open the file and read the first line, you check it to see if you obtained `null` (line 15 of Listing 15.6). If so, the file is empty and so you simply leave `this.itsFirst as null`.

If the file is not empty, you put its first String value into a node and make that the first node on the NodeSequence object's node list (line 16). Now it gets a little tricky. You cannot add a node to the end of the linked list unless you change the `itsNext` value in the previous node. So you have to keep track of that previous node throughout the loop that reads String values in from the file.

You initialize a local variable `previous = this.itsFirst` (line 17), since that will be the node previous to the one you are going to add next. Now you write the standard loop for reading data from a file until you run out. The only difference is that the two statements

```
itsItem[itsSize] = s;
itsSize++;
```

in the body of the loop in the BasicSequence constructor are replaced by these two statements (lines 20-21), which do much the same thing:

```
previous.itsNext = new Node (s, null);
previous = previous.itsNext;
```

This coding is in the middle part of Listing 15.6. It illustrates an important design principle: If you have to process a sequence of values and the first value requires a different kind of processing from the rest of them, do not try to have a loop process all the values. Instead, process the first value before the loop and have the loop start its processing with the second value. If you were to try to write the coding for this constructor by having a while-statement but no if-statement, you would quickly see why this is a valuable principle.

Implementing the constructor that has a NodeSequence parameter

To create a NodeSequence object that has the same elements in the same order as a given NodeSequence parameter, you first verify that the parameter does not have an empty list. If it does not, you make the executor's first node a new Node object containing the data from the first node of the parameter. Initialize a local variable `previous = this.itsFirst` and enter a loop processing the second and all later data values on the parameter's list. At each such data value, link a new node after `previous` containing that data value from the parameter and move `previous` on to that newly-constructed node (lines 31-32, analogous to lines 20-21).

This logic is in the lower part of Listing 15.6. Note that again the first node on the executor's list must be processed differently from all other nodes, because the first node is the only one that does not have a previous node.

Exercise 15.13 How would you modify the `equals` method for the NodeSequence class to tell whether the executor is a "prefix" of a NodeSequence parameter, i.e., the elements of the executor are at the beginning of the parameter in the same order, but the parameter may have more elements besides those?
Exercise 15.14 Write the NodeSequence method `public Object[] toArray()`. Hint: Use `new Object[this.size()]`.

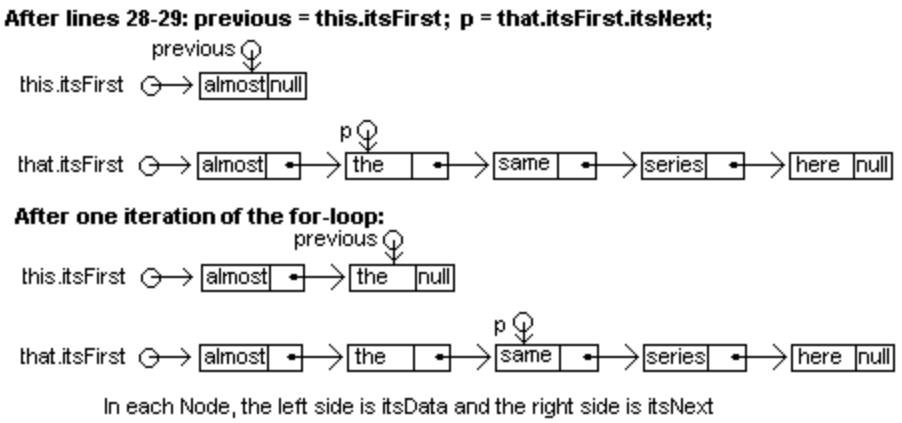


Figure 15.5 Two stages of execution of the second constructor

Exercise 15.15 Write a Node method `public void removeEvens()`: The executor removes every other Node from its linked list, i.e., the second, fourth, sixth, etc.
Exercise 15.16 Write another constructor for the NodeSequence class with an `Object[]` parameter: It constructs a Collection with the same elements in the same order as the array has. Hint: Work backwards from `given[given.length - 1]`.
Exercise 15.17* Write a NodeSequence method `public Collection reverse()`: The executor returns a new NodeSequence object with the same elements as the executor but in the opposite order.

15.5 Recursion With Linked Lists

The coding for each method in Listing 15.6 is lengthy and hard to follow. This is because the coding works with linked lists, which are naturally recursive structures, but it does not use recursion. A **naturally recursive structure** is a structure that can be implemented with a class X of objects that have instance variables of class X. Some people call these "self-referential objects" but technically they are not; `p.itsNext` refers to another Node object, not to `p`.

Deciding whether a String is a palindrome

Let us start with a recursion refresher, a method that uses recursion to tell whether a given String value reads the same forwards as backwards. Such a String is called a palindrome; an example is what Napoleon is rumored to have said: "able was I ere I saw elba". There are two kinds of palindromes, trivial and not:

- Any string that only has one character, or no characters at all, is trivially a palindrome.
- Any string with two or more characters is a palindrome when the first character is the same as the last and the smaller part in-between is a palindrome.

This logic is expressed quite naturally by the following independent method:

```
public static boolean isPalindrome (String s)  // independent
{  return s.length() <= 1
          || (s.charAt (0) == s.charAt (s.length() - 1)
          && isPalindrome (s.substring (1, s.length() - 1)));
}  //=====================
```

The equals method for NodeSequences

Two NodeSequence objects are equal if their linked lists of nodes have exactly the same data in the same order. So the `equals` method for NodeSequences is quite easy to work out if you put off most of the work to a separate `areEqual` method that tests whether two linked lists of nodes are equal:

```
public boolean equals (Object ob)
{  return ob instanceof NodeSequence
            && areEqual (this.itsFirst,
                    ((NodeSequence) ob).itsFirst);
}  //=====================
```

The `areEqual` method determines whether two linked lists have the same data values in the same order. There are two cases to consider, depending on whether one of the linked lists is empty or not:

- If either linked list is empty, then they are equal only if both are empty.
- If both are non-empty, then they are equal only if their first data values (in `itsData`) are equal to each other and their sublists (starting from `itsNext`) are also equal.

The coding for `areEqual` follows directly from those two cases:

```
private static boolean areEqual (Node one, Node two)
{  return (one == null  || two == null)  ?  one == two
          :  one.itsData.equals (two.itsData)
             && areEqual (one.itsNext, two.itsNext);
}  //=====================
```

See how much easier and more natural those two parts are together than the coding in Listing 15.6? Of course, you need a secondary method that recurses through the linked list of Nodes. The reason is that the `equals` method has a NodeSequence executor, and a NodeSequence is not a naturally recursive structure (since it does not have a NodeSequence instance variable). So `equals` calls the `areEqual` method for a pair of Node objects which store the naturally recursive linked lists.

The NodeSequence constructor using a file

The first NodeSequence constructor reads data from a disk file and creates a linked list from that sequence of data values. You can call the constructor of the superclass and create the file, at which point you can call a recursive method to give you the linked list that the file provides:

```
public NodeSequence (String fileName) throws IOException
{  super();
   BufferedReader file = new BufferedReader
                         (new FileReader (fileName));
   this.itsFirst = readFrom (file);
}  //=====================
```

The `readFrom` method gets all the String values in the file and returns the linked list of Nodes containing those values in the order read. It first reads a single String value from the file and then sees which of two cases applies -- the String exists or not:

- If the String value does not exist, then return the empty linked list.
- If the String value exists, then return a non-empty linked list for which the first Node contains the String you just read as its data and the Node's sublist contains all the rest of the String values from the file.

The coding for this logic following directly from those two cases:

```
private static Node readFrom (BufferedReader file)
                              throws IOException
{  String s = file.readLine();
   return s == null  ?  null : new Node (s, readFrom (file));
}  //======================
```

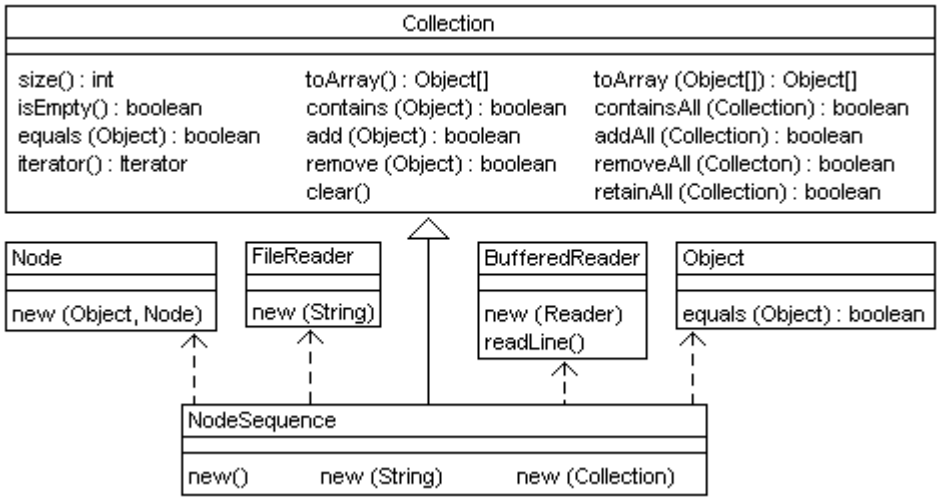


Figure 15.6 UML class diagram for the NodeSequence class

The NodeSequence constructor using another NodeSequence

To make a new NodeSequence that is a copy of another, you call the superclass's constructor and then make a new linked list that is a copy of the given linked list:

```
public NodeSequence (NodeSequence given)
{  super();
   this.itsFirst = copyList (given.itsFirst);
}  //======================
```

This calls a private `copyList` method whose job is to return a copy of the given linked list. This recursive method sees which of two cases applies:

- If the list to copy is empty, then return an empty list as its copy.
- Otherwise, return a non-empty linked list for which the first Node contains the data in the given linked list and the Node's sublist contains all the rest of the data values.

The coding for this logic following directly from those two cases:

```
private static Node copyList (Node toCopy)
{  return toCopy == null ? null : new Node (toCopy.itsData,
                                            copyList (toCopy.itsNext));
}  //======================
```

Exercise 15.18 Rewrite the `contains` method of Listing 15.5 by calling on a private recursive method with a Node parameter. Have only one statement in each method.
Exercise 15.19 Write a recursive Node method `public void removeEvens()`: The executor removes even-numbered Nodes from the linked list, i.e., the second, fourth, sixth, eighth, etc.
Exercise 15.20* Write a recursive Node method `public void omitDups()`: The executor removes all later Nodes containing a data value equal to its own data value.

15.6 Implementing The Iterator Interface For An Array-Based Collection

Your client needs a program that lists all purchased items that are missing from the inventory, both of which are stored in Collection objects. For this task you need to have an Iterator object. The Collection interface prescribes a method for which `someCollection.iterator()` returns an Iterator object connected to the Collection.

Definition of an Iterator

An Iterator provides the values in a Collection one at a time in some order. If it is a Collection for which order is important, it should always return the values in the specified order. A class satisfies the **Iterator interface** (in the `java.util` package) if it has the following three instance methods:

- `hasNext()` tells whether there is an element of the Collection that has not yet been provided by the Iterator.
- `next()` advances to the next element to be provided and returns it.
- `remove()` deletes from the Collection the element that was returned by the most recent call of `next()`. After removal, `next()` provides what it would have provided without the removal. Note: Many implementations of Iterator specify that no one is to call its `remove` method; it throws an UnsupportedOperationException if you call it.

Naturally, `next()` throws an Exception if there is no additional element and `remove()` throws an Exception if `next()` has not yet been called. The application program in Listing 15.7 (see next page) shows how an Iterator object can be used. The same coding would also work if "NodeSequence" were replaced by "BasicSequence" throughout. It is important that the value that `it.next()` returns be assigned to a variable, since the one value is used twice. If you called `it.next()` twice in the body of the loop, it would give you the next two values, not the same value twice.

An Iterator acts as a non-pushable stack containing the elements in the Collection. `it.next()` corresponds to `stack.pop()`; it takes the next available element out of the stack and returns it. `it.hasNext()` corresponds to `!stack.isEmpty()`. Since the stack is not the Collection, popping from the stack does not remove from the Collection itself; that requires `it.remove()`.

A private nested class

An Iterator object constructed for a BasicSequence object needs access to the private instance variables of that sequence. Encapsulation is maintained if you make this BasicSequenceIterator class a private nested class of BasicSequence, which means that no outside class can mention the name BasicSequenceIterator. It is declared inside the BasicSequence class with the following class heading:

```
private static class BasicSequenceIterator extends Object
                                            implements Iterator
```

Listing 15.7 The LostItems application program

```
public class LostItems
{
   /** List all purchased items not in inventory. */

   public static void main (String[] args)
   {  BasicSequence inventory;
      BasicSequence purchased;
      try
      {  inventory = new BasicSequence ("inven.dat");
         purchased = new BasicSequence ("purch.dat");
      }catch (java.io.IOException e)
      {  throw new RuntimeException ("A file is defective.");
      }

      if (inventory.equals (purchased))
         System.out.println ("a perfect match");
      else if ( ! inventory.containsAll (purchased))
      {  System.out.println ("Listing all values we lost:");
         java.util.Iterator it = purchased.iterator();
         while (it.hasNext())
         {  Object data = it.next();
            if ( ! inventory.contains (data))
               System.out.println (data.toString());
         }
      }
   }  //======================
}
```

The instance method in the BasicSequence class that produces an Iterator for outside classes to use can be coded as follows. An outside class (such as LostItems in Listing 15.7) that calls this method must store the object the method returns in an Iterator variable, not a BasicSequenceIterator variable, because the latter name is private:

```
   public  Iterator  iterator()  // in BasicSequence
   {  return new BasicSequenceIterator (this);
   }  //======================
```

The BasicSequence object is passed as a parameter so that its Iterator object can refer to its array and to the size of that array.

Array implementation of an Iterator

For the BasicSequenceIterator class, we choose to keep track of the current position of the Iterator in an int instance variable named `itsPos`. Each time `next()` is executed, we add 1 to the value of `itsPos` and then return the data value in `itsItem[itsPos]`.

We need to keep track of whether calling `remove` is allowed. We can do this with a boolean variable `isRemovable`. When the Iterator object is first created, this variable is initialized to `false`. Whenever `next` is called, this variable is made `true`.

The very first time we execute `next()`, we should get `itsItem[0]`. It follows that `itsPos` must be initialized to -1 so that adding 1 to it puts it at 0. And of course, we cannot execute `next()` if there is no data value in `itsItem[itsPos+1]`, i.e., if `itsPos+1` equals `itsSize`. The coding for the constructor and the `hasNext` and `next` methods is in the upper part of Listing 15.8.

Listing 15.8 The BasicSequenceIterator nested class of objects

```
// This class goes inside the BasicSequence class

private static class BasicSequenceIterator extends Object
                                            implements Iterator
{
   private int itsPos = -1;      // next() is itsItem[itsPos+1]
   private boolean isRemovable = false;
   private BasicSequence itsSeq;


   public BasicSequenceIterator (BasicSequence givenSequence)
   {  super();
      itsSeq = givenSequence;
   }  //=====================

   /** Tell whether there is a next element to be returned. */

   public boolean hasNext()
   {  return itsPos + 1 < itsSeq.itsSize;
   }  //=====================

   /** Advance to the next object to be returned and return it.
    *  Throw NoSuchElementException if hasNext() is false. */

   public Object next()
   {  if ( ! hasNext())
         throw new NoSuchElementException ("hasNext is false");
      isRemovable = true;
      itsPos++;
      return itsSeq.itsItem[itsPos];
   }  //=====================

   /** Remove the object that was just returned by next().
    *  Throw IllegalStateException if next() has never been
    *  called, or if next() has not been called since the
    *  most recent call of remove(). */

   public void remove()
   {  if ( ! isRemovable)
         throw new IllegalStateException ("nothing to remove");
      for (int k = itsPos + 1;  k < itsSeq.itsSize;  k++)
         itsSeq.itsItem[k - 1] = itsSeq.itsItem[k];
      itsSeq.itsSize--;
      itsPos--;
      isRemovable = false;  // no remove twice in a row
   }  //=====================
}
```

Internal invariant for BasicSequenceIterators

- The instance variable `itsSeq` is the BasicSequence it iterates through.
- The instance variable `itsPos` is the int such that `itsSeq.itsItem[itsPos+1]` contains the element that `next()` will return, except `next()` is illegal when `itsPos+1 == itsSeq.itsSize`.
- The instance variable `isRemovable` tells whether a call of `remove` is allowed.

The Iterator interface specifies that a **NoSuchElementException** be thrown if there is no next element to return. This Exception class is in the `java.util` package. The way you throw a NoSuchElementException object is quite simple -- just execute the following statement (the phrase in quotes is whatever you choose):

```
throw new NoSuchElementException ("hasNext is false");
```

The remove method for Iterators

The `remove` method deletes the element that was returned by the most recent execution of `next()`. That of course is impossible if `next` has not yet been called or if the element it returned has already been removed. In such cases you are to throw an **IllegalStateException** (in `java.lang`).

When `remove` is called, you remove the element in `itsSeq.itsItem[itsPos]` (shifting other values down one) and make a note that another immediate call of `remove` is now forbidden (you cannot remove what is already gone). The next call of `next` should return the first value that was shifted down. That means that you should decrement `itsPos` in preparation for the next call of `next`. This coding is in the lower part of Listing 15.8. Figure 15.7 should clarify what is going on.

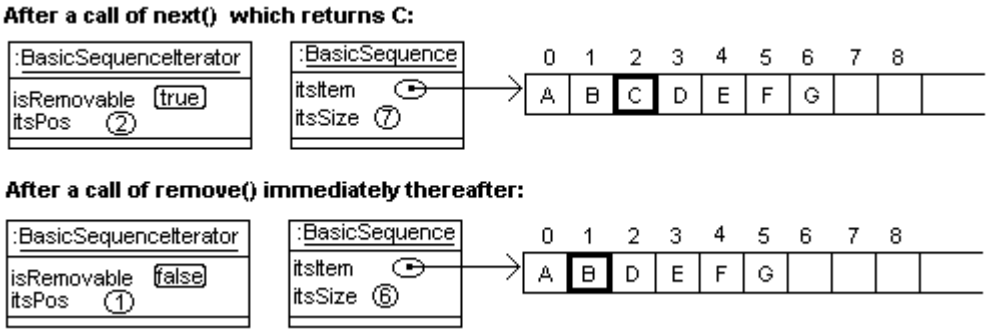


Figure 15.7 UML object diagrams for BasicSequenceIterator operations

Removal of `itsItem[itsPos]` requires shifting each element indexed `itsPos+1` and higher to the component indexed 1 less than itself. If you think about it a while, you will see why this shifting is easier if you work from `itsPos` toward `itsSize` rather than vice versa.

Exercise 15.21 What changes would you make in the `remove` method of Listing 15.8 if it specified that you are to return the Object that is removed?
Exercise 15.22 Write a generic Collection `containsAll` method, i.e., coding that it works correctly for any Collection executor and for any Collection parameter. Hint: Use the parameter's Iterator to go through its elements one at a time.
Exercise 15.23 Modify Listing 15.7 to execute much faster on the precondition that every element of the `inventory` Collection is known to be in the `purchased` Collection and those elements are listed in the same order.
Exercise 15.24 If you studied inner classes in Chapter Nine, rewrite the BasicSequenceIterator class as an inner class. Also rewrite the `iterator` method.
Exercise 15.25* Rewrite the `remove` method in BasicSequenceIterator to explicitly state the executor wherever possible.
Exercise 15.26* Generalize the second BasicSequence constructor in Listing 15.3 to have a Collection parameter. Use its iterator to create the copy.
Exercise 15.27* Rewrite the NodeSequence `equals` method in Listing 15.6 to tell whether it has the same elements in the same order as its parameter, which can be any Collection object (i.e., do not throw an Exception just because it is not a NodeSequence).

Exercise 15.28* Rewrite Listing 15.8 with a different internal invariant, namely, the element that `next()` returns is `itsItem[itsPos+1]` only when `isRemovable` is `true`. Otherwise `next()` returns `itsItem[itsPos]`.
Exercise 15.29* Draw the UML class diagram for the LostItems class.
Exercise 15.30** Rewrite the `remove` method in Listing 15.8 to shift elements down starting from the far end of the array, working from `itsSize-1` on down.

15.7 Implementing The Iterator Interface For A Linked-List-Based Collection

For the NodeSequence's Iterator class, you can use something analogous to the BasicSequenceIterator implementation: You keep track of the current position of the Iterator in a Node instance variable named `itsPos` . Each time you execute `next()`, you advance `itsPos` to the next Node and return the data in that Node. The primary problem occurs when you have to remove the data in the Node that `itsPos` refers to.

The easiest way to do that is to make a note of the Node just before `itsPos`. That is the Node that `itsPos` advances from when `next()` is executed. You could record that information in an instance variable named `itsPrevious`, since it is the Node before the data value that can be removed. You could make its value `itsPos` when `remove()` is not allowed, so you do not need an extra boolean instance variable to keep track of that information. Then the coding for `next` would be the exact analog of its coding for the array implementation, to wit:

```
if ( ! hasNext())
   throw new NoSuchElementException ("hasNext is false");
itsPrevious = itsPos;
itsPos = itsPos.itsNext;
return itsPos.itsData;
```

But now you have another problem: The first node does not have a node before it, so what do you initialize `itsPos` to? This problem is easily fixed: Construction of an iterator creates a **dummy header node** that links to `itsFirst` and initializes `itsPos` to that dummy header node. So the status of the implementation, i.e., the internal invariant, will always be as follows. Compare this description with the internal invariant in the previous section. In particular, `itsPrevious` gives the information required to do the equivalent of `itsPos--`, while the condition `itsPrevious != itsPos` gives the same information as the value of `isRemovable`:

Internal invariant for NodeSequenceIterators

- The instance variable `itsSeq` is the NodeSequence it iterates through.
- The instance variable `itsPos` is the Node such that `itsPos.itsNext.itsData` always contains the element that `next()` will return, except `next()` is illegal when `itsPos.itsNext` is `null`.
- The instance variable `itsPrevious` is the Node before `itsPos` if a call of `remove` is allowed, otherwise `itsPrevious` equals `itsPos`.

The logic for `hasNext` is simple: The iterator has a next value that it can advance to if there is a Node directly after `itsPos`. The coding for the constructor, `hasNext`, and `next` is in the upper part of Listing 15.9. You must also add the following method to the NodeSequence class:

```
public  Iterator  iterator()  // in NodeSequence
{  return new NodeSequenceIterator (this);
}  //=====================
```

Listing 15.9 The NodeSequenceIterator nested class of objects

```
// This class goes inside the NodeSequence class

private static class NodeSequenceIterator extends Object
                                          implements Iterator
{
   private Node itsPos;       // next() is itsPos.itsNext.itsData
   private Node itsPrevious;  // == itsPos when remove disallowed
   private NodeSequence itsSeq;


   public NodeSequenceIterator (NodeSequence givenSequence)
   {  super();
      itsSeq = givenSequence;
      itsPos = new Node (null, itsSeq.itsFirst);
      itsPrevious = itsPos;     // signals no remove allowed
   }  //=====================

   public boolean hasNext()
   {  return itsPos.itsNext != null;
   }  //=====================

   public Object next()
   {  if ( ! hasNext())
         throw new NoSuchElementException ("hasNext is false");
      itsPrevious = itsPos;
      itsPos = itsPos.itsNext;  // so now itsPrevious != itsPos
      return itsPos.itsData;
   }  //=====================

   public void remove()
   {  if (itsPrevious == itsPos)
         throw new IllegalStateException ("nothing to remove");
      itsPrevious.itsNext = itsPos.itsNext;
      if (itsSeq.itsFirst == itsPos)
         itsSeq.itsFirst = itsPos.itsNext;
      itsPos = itsPrevious;     // signals no remove allowed
   }  //=====================
}
```

The remove method for NodeSequenceIterator

The `remove` coding is far simpler than it was for the array implementation. As the internal invariant indicates, as long as `itsPrevious` is not equal to `itsPos`, you can just link from `itsPrevious` around the node `itsPos` to delete it from the linked list.

However, there is a special case: If the node to be removed is the first node on the linked list, then you have to reset `itsSeq.itsFirst` to refer to the currently-second node on the linked list. Either way, you make `itsPos` be `itsPrevious` (which prevents an additional call of `remove`). The full logic is in the lower part of Listing 15.9. Figure 15.8 illustrates how things change for a call of `next` followed by a call of `remove`.

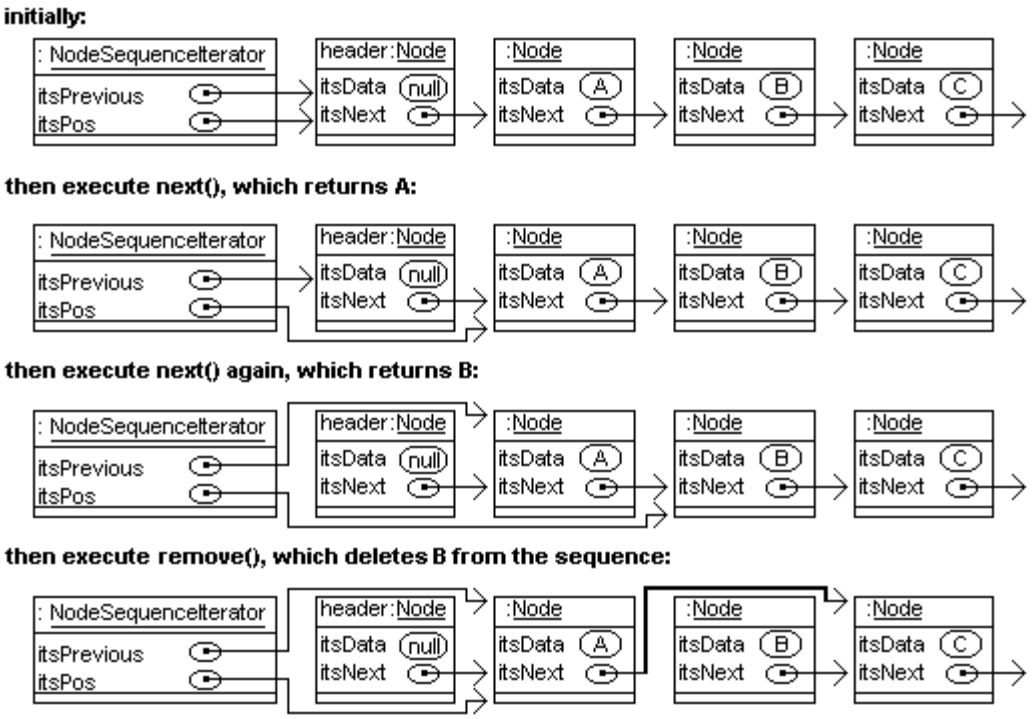


Figure 15.8 UML object diagrams for NodeSequenceIterator operations

Exercise 15.31 How would you change the NodeSequenceIterator class to have a `nextIndex` method which returns the index number of the element that would be returned by the next execution of `next()` (0 for the first, 1 for the second, etc.)?
Exercise 15.32 Generalize the second NodeSequence constructor in Listing 15.6 to have a Collection parameter. Use its iterator to create the copy. Use recursion.
Exercise 15.33 What changes would you make in Listing 15.9 to initialize `itsPos` to `null` and thus avoid having a dummy header node?
Exercise 15.34* Revise Listing 15.9 for a different approach: Omit `itsPos` and keep `itsPrevious` with the same meaning. Have a boolean instance variable `isRemovable` to tell when you may remove a value. Adjust everything accordingly.

15.8 Implementing A Modifiable Collection With Linked Lists

The Collection interface has six methods that modify the Collection. They are described in Listing 15.10 (see next page). This listing plus Listing 15.2 describe the Collection interface in its entirety.

Some implementations of the Collection interface do not allow modifications. The Java convention is that implementations that do not allow modifications are to have these six methods throw an **UnsupportedOperationException** (from `java.lang`), using e.g. the following statement (our two implementations will of course allow modifications):

```
throw new UnsupportedOperationException();
```

In fact, an implementation of Iterator is allowed to have the preceding statement as the body of the Iterator's `remove` method if the Collection is to be unmodifiable. In this sense, the six methods mentioned in Listing 15.10 are **optional operations** for a Collection and `remove` is an optional operation for an Iterator.

Listing 15.10 The Collection interface, part 2

```
/** A Collection class that does not guarantee maintaining a
 *  specific order ignores the ordering specifications here.
 *  A Collection class that does not allow null as an element
 *  throws a java.lang.IllegalArgumentException if you add null.
 *  A method that returns a boolean value returns true
 *  if and only if the method modifies the Collection. */

// public interface Collection, the rest of the 13 methods

   /** Make the Collection have no elements at all. */
   public void clear();

   /** Add the given Object at the end of this sequence.
    *  No effect if the Collection does not allow duplicates. */
   public boolean add (Object ob);

   /** Same as repeated add for each element in sequence. */
   public boolean addAll (Collection that);

   /** Remove the first instance of the given object from the
    *  Collection, if present. */
   public boolean remove (Object ob);

   /** Same as repeated remove for each element in sequence. */
   public boolean removeAll (Collection that);

   /** Remove every element not in that Collection.  Keep the
    *  original order for those elements that remain. */
   public boolean retainAll (Collection that);
```

If you have a Collection implementation that does not maintain a particular order and that does not allow duplicates, the following expressions produce the set-union, set-intersection, and set-difference of Collections A and B, without changing A or B:

```
Collection union = new Collection (A).addAll (B);
Collection intersection = new Collection (A).retainAll (B);
Collection difference = new Collection (A).removeAll (B);
```

The **Set** interface in the standard library (`java.util` package) has the same methods as the Collection interface. The difference is that it does not allow duplicates (but it allows one `null` value) and the order of the elements is not necessarily guaranteed.

Implementing the removeAll method

The `removeAll` method is not difficult if you just repeatedly call on the `remove` method. Specifically, you get an iterator from the Collection parameter to run down its sequence of elements and remove each one. Since you are charged with returning `true` when any change in the executor Collection is made, you can initialize a boolean variable to `false` and then change it to `true` any time one of the `remove` operations succeeds.

The coding for `removeAll` is in the upper part of Listing 15.11. Note that it is **generic**: It could be put in any class that implements Collection and it will work right, assuming the other Collection methods it calls work right.

Listing 15.11 The NodeSequence class of objects, part 3

```
// public class NodeSequence, three more methods

   public boolean removeAll (Collection that)
   {  boolean changed = false;
      Iterator it = that.iterator();
      while (it.hasNext())
         changed = this.remove (it.next()) || changed;
      return changed;
   }  //=====================

   public boolean add (Object ob)
   {  if (ob == null)
         throw new IllegalArgumentException ("no nulls allowed");
      if (itsFirst == null)
         itsFirst = new Node (ob, null);
      else
         itsFirst.addLater (ob);
      return true;  // we accept duplicates of elements
   }  //=====================

   public boolean remove (Object ob)
   {  if (itsFirst == null)
         return false;
      if (itsFirst.itsData.equals (ob))
      {  itsFirst = itsFirst.itsNext;
         return true;
      }
      return itsFirst.removeLater (ob);
   }  //=====================


// private static class Node, 2 more methods

   public void addLater (Object ob)
   {  if (itsNext == null)
         itsNext = new Node (ob, null);
      else
         itsNext.addLater (ob);
   }  //=====================

   public boolean removeLater (Object ob)
   {  if (itsNext == null)
         return false;
      if (itsNext.itsData.equals (ob))
      {  itsNext = itsNext.itsNext;
         return true;
      }
      return itsNext.removeLater (ob);
   }  //=====================
```

The `removeAll` coding uses two methods that not only take action but also return a value. Both are legitimized by the fact that they are part of the Sun standard library, but you can see that doing so much in one phrase can make the logic difficult to follow. Some people feel it would have been better if the Iterator class had been defined with two separate methods such as `getNext()` and `moveOn()` to take the place of the one `next()` method.

Implementing the NodeSequence add method

To add a given Object ob to the end of the linked list whose first node is `this.itsFirst`, you first must check that the Object parameter exists. If the parameter is `null`, you throw an **IllegalArgumentException** object (from `java.lang`). Then if `this.itsFirst` is `null`, you just create a new node to be the only node on the linked list, as follows:

```
this.itsFirst = new Node (ob, null);
```

If, on the other hand, the linked list already has at least one node, you need to add a node containing `ob` at the end of the linked list. Since the logic of `add` has already become rather complex, just ask the first node to do that job (i.e., call an instance method in the Node class).

The Node instance method you call could be named `addLater`. You ask a node on your linked list to add `ob` some place after it. There are two possibilities: Either the node X you ask is the last node, in which case X simply adds a new node after it containing `ob`, or else there is a node after X, in which case X asks that next node to add `ob` later. This coding is in the middle part of Listing 15.11. Figure 15.9 shows an example.

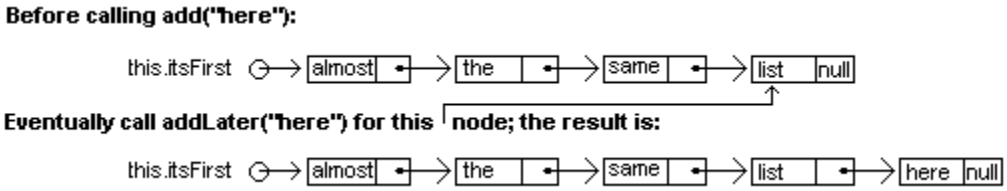


Figure 15.9

Designing the remove method

When you work out a complex logic, it usually helps to figure out what to do in the easy cases and postpone the hard cases to another method. For `remove`, if the executor has no nodes, do nothing. If it has nodes and the first node contains `ob`, delete that first node. Otherwise, ask that first node to do the removing and report back whether it was able to do so. This plan is formalized in the accompanying design block.

SNL DESIGN to remove ob

1. If the executor has no nodes then...
   Return `false` to indicate no change was made.
2. If the first node on the executor's linked list contains `ob` then...
   Make the currently-second node the new first node.
   Return `true` to indicate a change was made.
3. Ask the first node to remove `ob` from a node later in the linked list, if present.
4. Return `true` if `ob` was removed from a later node, otherwise return `false`.

What does that first node do when asked to remove `ob`? The same thing: If it has no node after it, it does nothing, but if the node after contains `ob`, it deletes the node after it, otherwise it asks the node after it to do the removing and report back whether it could do so. That gives the recursive `removeLater` method in the lower part of Listing 15.11.

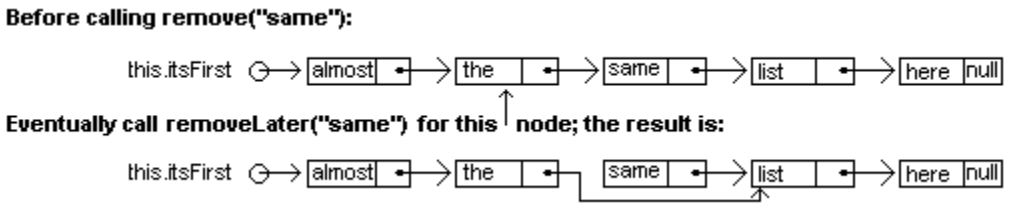


Figure 15.10 Effect of a call of remove("same")

If you compare the coding for `removeLater` with the coding for `remove`, you can see they are word-for-word the same except `itsNext` plays the role of `itsFirst`. And if you compare the coding for `addLater` with the middle four lines of `add`, they have the same resemblance.

Exercise 15.35 Write the NodeSequence method `public void clear()`.
Exercise 15.36 Write the NodeSequence method `public boolean addAll (Collection that)` to repeatedly call the executor's `add` method.
Exercise 15.37 What changes would you make in Listing 15.11 to have `addLater` and `removeLater` be private class methods in the NodeSequence class?
Exercise 15.38 Write the NodeSequence method `public boolean retainAll (Collection that)` as follows: First. repeatedly delete the executor's first node until you see that the Collection parameter contains the data in its first node (or the executor becomes empty). Then go through each node in the executor's linked list one at a time, deleting the nodes after it whose data is not in the Collection parameter.
Exercise 15.39* Write the BasicSequence method `public boolean add (Object ob)`.
Exercise 15.40* Rewrite the `add` method for NodeSequence without using recursion.
Exercise 15.41* Rewrite the `remove` method for NodeSequence without using recursion (use a for-loop).

15.9 Implementing The ListIterator Interface With Linked Lists

An ordinary Iterator allows you to remove an element you come across in the iteration, but it does not allow you to add an element at a specific position in the sequence, nor does it allow you to replace one element by another at a specific position. For this capability you need a ListIterator kind of object. **ListIterator** is an interface in the `java.util` package that extends the Iterator interface. If `lit` is a ListIterator, then `lit.add(ob)` and `lit.set(ob)` are method calls that do just what you want.

However, the ListIterator interface requires two methods named `hasPrevious` and `previous`, which do the same as `hasNext` and `next`, respectively, except they go backwards in the list. This you do not want (how do you go backwards in a straightforward linked list?). Not to worry -- in accordance with the Java convention, you just include implementations of these methods with the standard coding that lets people know not to use them:

```
throw new UnsupportedOperationException();
```

It would be preferable if the Sun standard library offered a SequenceIterator interface that had only the `add` and `set` methods in addition to the Iterator methods. The SequenceIterator interface would extend Iterator and ListIterator would extend SequenceIterator. But apparently they did not think of that.

The ListIterator interface has the nine non-constructor methods given in Listing 15.12, which completes the implementation of NodeSequenceIterator. The ListIterator operations `nextIndex` and `previousIndex` are unsupported. They use zero-based indexing, e.g., if `next()` would return the fourth element of the list, then `nextIndex()` returns 3 and `previousIndex()` returns 2. We add another method in NodeSequence and BasicSequence with the same coding as for the `iterator` method but with the following heading:

```
public ListIterator listIterator()
```

Listing 15.12 The NodeSequenceIterator nested class of objects, revised

```
private static class NodeSequenceIterator extends Object
                                          implements ListIterator
{
   private Node itsPos;       // next() is itsPos.itsNext.itsData
   private Node itsPrevious;  // == itsPos when remove disallowed
   private NodeSequence itsSeq;

   /** Replace the object last returned by next().  Throw an
    *  IllegalStateException if removal is not allowed. */

   public void set (Object ob)
   {  if (ob == null)
         throw new IllegalArgumentException ("no nulls allowed");
      if (itsPrevious == itsPos)
         throw new IllegalStateException ("nothing to replace");
      itsPos.itsData = ob;
   }  //====================

   /** Add the given object just before the element that will
    *  returned by next(), or at the end if hasNext() is false.
    *  Disallow set or remove until next is used again. */

   public void add (Object ob)
   {  if (ob == null)
         throw new IllegalArgumentException ("no nulls allowed");
      itsPos.itsNext = new Node (ob, itsPos.itsNext);
      itsPos = itsPos.itsNext;
      if (itsSeq.itsFirst == itsPos.itsNext)
         itsSeq.itsFirst = itsPos;
      itsPrevious = itsPos;   // so no one can remove it
   }  //====================

   // hasNext(),next(), remove(), and NodeSequenceIterator()
   // are implemented in Listing 15.9

   // The following four do not apply to sequences
   public boolean hasPrevious()  // is there a previous one?
   { throw new UnsupportedOperationException(); }
   public Object previous()      // return the one before
   { throw new UnsupportedOperationException(); }
   public int nextIndex()     // index of what next() returns
   { throw new UnsupportedOperationException(); }
   public int previousIndex() // index of what previous() returns
   { throw new UnsupportedOperationException(); }
}
```

Implementation of the set and add methods

The `set` and `remove` methods can only be called if `next` has been called with no intervening call of `add` or `remove`. That is, calling `next` makes a value available for removing or replacing, and calling `add` or `remove` leaves no value available for removing or replacing. So if `set` is allowed, you just replace the data in the node `itsPos.itsNext`. The coding for this is in the upper part of Listing 15.12.

The `add` method is allowed anytime; when `hasNext()` is `false`, you add at the end of the sequence. You first check that no one is trying to add `null`. Then you create a new node and link it in after the node referenced by `itsPos`. Then you can set `itsPos` to be that new node, since executing `next()` should return the element after the one just added. This coding is in the middle part of Listing 15.12. Figure 15.11 should clarify what is going on here.

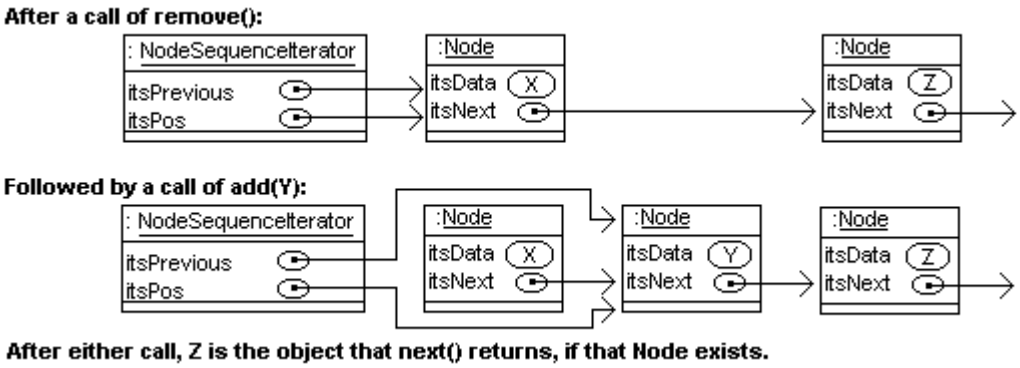


Figure 15.11 UML object diagram before and after a call of add(Z)

One special case occurs: If you added a node before the first node in the sequence, then you have to reset `itsSeq.itsFirst` to now indicate the newly-added node.

The header-node variant

An implementation of NodeSequenceIterator that simplifies the coding and lowers the execution time has the NodeSequence object create one dummy header node that all iterators use. This **header-node implementation** of a sequence is left as a major programming project. Since that dummy header will store the first node containing data as `itsNext` value, the NodeSequence object does not need to keep track of that data Node separately. So `itsFirst` can instead record the one dummy header node that all iterators use, as follows:

```
public NodeSequence()  // header-node implementation
{  super();
   itsFirst = new Node (null, null);
}  //======================
```

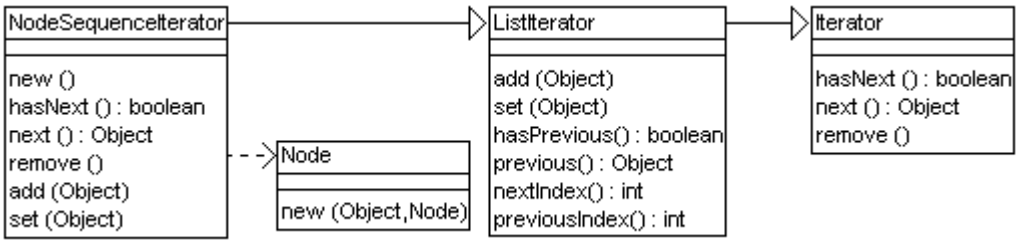


Figure 15.12 UML class diagram for NodeSequenceIterator

Implementing Stacks and Queues with Sequences

You can easily implement a Stack (described in Listing 14.1) as a subclass of any class that implements a modifiable Collection. The `peekTop()` call is to return the value on top of the Stack, so it could be written as follows:

```
public Object peekTop()    // in Stack
{  return iterator().next();
}  //======================
```

If the Stack is empty, `peekTop()` throws an Exception, as it should. The `push(ob)` call is to add a value to the top of the Stack, so it could be as follows:

```
public void push (Object ob)    // in Stack
{  iterator().add (ob);
}  //======================
```

The `pop()` call is to remove the value on top of the Stack; it is a bit more complex:

```
public Object pop()  // in Stack
{  Iterator it = iterator();
   Object valueToReturn = it.next();
   it.remove();
   return valueToReturn;
}  //======================
```

A Queue class (also described in Listing 14.1) can be implemented just as easily. The `dequeue` method is coded the same as `pop` and the `peekFront` method is coded the same as `peekTop`. The `enqueue(ob)` call is just one statement:

```
public Object enqueue (Object ob)    // in Queue
{  this.add (ob);   // put it at the end of the list
}  //======================
```

Exercise 15.42 Write the BasicSequenceIterator method `public void set (Object ob)`: It replace the current value by `ob`.
Exercise 15.43 Write `push` and `pop` methods to be added to the NodeSequence class without using an iterator; just code them directly in terms of Nodes.
Exercise 15.44* Add another instance variable `itsSize` to the NodeSequence class. Modify everything that has to be modified in Listings 15.6 through 15.12 so that the `size` method can simply return `itsSize`.
Exercise 15.45* Write an independent method `public static Object findMax (Collection par)`: It finds the largest value in the Collection. Precondition: All elements of the Collection are mutually Comparable.
Exercise 15.46* Essay: Explain what can go wrong if one creates a BasicSequenceIterator, then calls its methods several times, then uses the `add` or `remove` methods in the BasicSequence class, and then calls more methods in BasicSequenceIterator.
Exercise 15.47** Essay: Same as the preceding exercise, but for Nodes instead.

15.10 Implementing The ListIterator Interface With Doubly-Linked Lists

The ListIterator interface has methods to allow client classes to move backward one step in the sequence of values (described in the last four methods of Listing 15.12). This is highly inefficient with standard linked lists. It becomes quite easy if you define a new kind of node that records the node before it as well as the node after it, as shown in Listing 15.13. This Node class would be defined inside a class named TwoWaySequence that implements the Collection interface.

Listing 15.13 The Node class for the TwoWaySequence class

```
private static class Node extends Object // inside TwoWaySequence
{
   public Object itsData;
   public Node itsNext;
   public Node itsPrevious;


   public Node (Object data, Node next, Node previous)
   {  super();
      itsData = data;
      itsNext = next;
      itsPrevious = previous;
   }  //=====================
}
```

Implementing the TwoWaySequence class

The coding for the TwoWaySequence class is greatly simplified if we use a dummy header node. That is, a TwoWaySequence object has one instance variable `itsHead`, which has `null` for `itsData`. `itsHead.itsNext` is a Node containing the first data value on the list, and the `itsNext` value for that node is a node containing the second data value on the list. This continues to the last node containing data, whose `itsNext` value is the header node (so it is a **circular list**).

For every case in which `p.itsNext` is the Node q, it will be true that `q.itsPrevious` is p, and vice versa. This is called a **doubly-linked list**. If the sequence is empty, then its linked list consists only of the header node, and so `itsHead.itsNext` is `itsHead` and also `itsHead.itsPrevious` is `itsHead`.

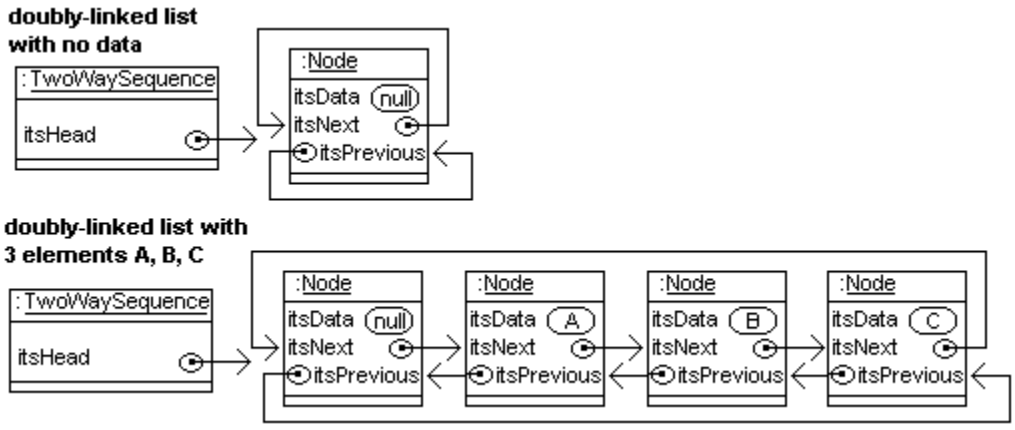


Figure 15.13 Two different doubly-linked lists

The `contains` method for a doubly-linked list requires the Some-A-are-B logic: You have a Node variable `p` go through each node that contains data (starting with the first one `p = itsHead.itsNext`) and return `true` if you see a node where `itsData` equals the parameter. But if you run out of nodes to look in (when `p == itsHead`), return `false`. This method and a constructor are in the upper part of Listing 15.14.

Listing 15.14 The TwoWaySequence class of objects, partial listing

```
import java.io.*;
import java.util.*;

public class TwoWaySequence extends Object implements Collection
{
   private final Node itsHead = new Node (null, null, null);

   public TwoWaySequence()
   {  super();
      itsHead.itsNext = itsHead;
      itsHead.itsPrevious = itsHead;
   }  //=====================

   public boolean contains (Object ob)
   {  for (Node p = itsHead.itsNext; p != itsHead; p = p.itsNext)
         if (p.itsData.equals (ob))
            return true;
      return false;
   }  //=====================

   public TwoWaySequence (Collection that)
   {  super();
      itsHead.itsNext = itsHead;
      itsHead.itsPrevious = itsHead;
      Iterator it = that.iterator();
      while (it.hasNext())
      {  Node last = itsHead.itsPrevious;
         last.itsNext = new Node (it.next(), itsHead, last);
         itsHead.itsPrevious = last.itsNext;
      }
   }  //=====================

   public boolean equals (Object ob)
   {  if ( ! (ob instanceof Collection))
         return false;
      Node p = this.itsHead.itsNext;
      Iterator it = ((Collection) ob).iterator();
      while (it.hasNext())
      {  if (p == this.itsHead || ! p.itsData.equals (it.next()))
            return false;
         p = p.itsNext;
      }
      return p == this.itsHead;  // != means p has more than it
   }  //=====================

   public Iterator iterator()
   {  return new TwoWaySequenceIterator (itsHead);
   }  //=====================
}
```

The constructor that makes a copy of a given Collection parameter starts out by making an empty TwoWaySequence. Then it runs through each element produced by the parameter's iterator, each time finding its `last` node (the one before `itsHead`) and creating a new node linked after that `last` node containing the iterator's element. This coding is in the middle part of Listing 15.14.

For the `equals` method, you first make sure that the parameter is in fact a Collection kind of object, otherwise you return `false`. You then get an iterator for the parameter and start with a local node variable p equal to the first node on the executor's list that contains data. Now verify that the iterator's `next()` value equals p's data at each point. Advance with `p = p.itsNext` each time (the iterator automatically moves on). You also have to make sure that the iterator and p run out of values at the same time. This coding is in the lower part of Listing 15.14. The rest of the TwoWaySequence methods are left as exercises.

Implementing the ListIterator class

The doubly-linked list makes the ListIterator easier to implement. You can make `itsPos` always be the node before the one containing the data that `next()` will return. If `next()` is illegal, then `itsPos` is just before the header node. Execution of `previous()` always returns the element immediately before the one that `next()` would return (except if there is no element before it). Execution of `next()` immediately after a call of `previous` returns the same value that was returned by `previous()`.

When `remove` is called, the one removed is determined by the direction in which the iterator last moved, i.e., you remove the result of the most recent execution of `next()` or `previous()`. So you need to keep track of that direction, say in an int variable named `itsDirection`: +1 if `next()` was the most recent call, -1 if `previous()` was, 0 if removal is not even allowed.

A ListIterator must also be able to return the index (zero-based) of the element that a call of `previous` will return (-1 if there is no previous value) and the index of the element that a call of `next` will return (`size()` if there is none). The easiest way to do that is for the iterator to have another instance variable that keeps track of the index of the node to which it currently refers. Call it `itsIndex`. So `previousIndex()` returns `itsIndex`.

Internal invariant for TwoWaySequenceIterators

- The instance variable `itsPos` is the Node such that `itsPos.itsNext.itsData` always contains the element that `next()` will return, except `next()` is illegal when `itsPos.itsNext` is the header node (recognized by having `itsData == null`).
- The instance variable `itsIndex` is the zero-based index index of the data value that a call of `previous()` would return; it is -1 if `previous()` is illegal.
- The instance variable `itsDirection` is 0 if `remove()` is illegal, otherwise it is +1 or -1 depending on whether `next()` or `previous()` was the most recent method call.

The only thing the `hasNext` method has to do is to say whether the node after `itsPos` contains any data, i.e., is not the header node. The coding for `hasNext` and the constructor is in the upper part of Listing 15.15.

The basic idea of the `previous` method is to back up `itsPos` by one node to be `itsPos.itsPrevious`, make a note that `itsDirection` is -1, and return the data value in the original `itsPos` node. However, if `itsPos` was at the header node, a call of `itsPrevious` should throw a NoSuchElementException. The coding for the `previous` method is in the middle part of Listing 15.15.

Listing 15.15 The TwoWaySequenceIterator nested class of objects

```
private static class TwoWaySequenceIterator extends Object
                                          implements ListIterator
{
   // Internal invariant: itsPos.itsNext is the header node if
   // hasNext() is false; otherwise, itsPos.itsNext is the node
   // containing the data that next() will return.

   private Node itsPos;
   private int itsDirection = 0;  // signals remove() not allowed
   private int itsIndex = -1;     // returned by previousIndex()


   public TwoWaySequenceIterator (Node givenHeaderNode)
   {  super();
      itsPos = givenHeaderNode;
   }  //=====================

   public boolean hasNext()
   {  return itsPos.itsNext.itsData != null;
   }  //=====================

   public Object previous()
   {  if (itsPos.itsData == null)
         throw new NoSuchElementException ("cannot back up");
      itsPos = itsPos.itsPrevious;
      itsDirection = -1;
      itsIndex--;
      return itsPos.itsNext.itsData;
   }  //=====================

   public void add (Object ob)
   {  if (ob == null)
         throw new IllegalArgumentException ("no nulls allowed");
      itsPos = new Node (ob, itsPos.itsNext, itsPos);
      itsPos.itsNext.itsPrevious = itsPos;
      itsPos.itsPrevious.itsNext = itsPos;
      itsDirection = 0;
      itsIndex++;
   }  //=====================

   // the first three of the following are left as exercises
   public Object next()               { return null; }
   public void remove()               {              }
   public void set (Object ob)        {              }
   public boolean hasPrevious()       { return itsIndex >= 0; }
   public int nextIndex()             { return itsIndex + 1;  }
   public int previousIndex()         { return itsIndex;      }
}
```

The `add` method for the TwoWaySequenceIterator can begin by creating a new node containing the given data. Then the new node is linked in after `itsPos` and before `itsPos.itsNext`, and `itsDirection` is set to 0. The value of `itsPos` has to become the new node, so `itsIndex` has to be incremented. The coding for the `add` method is in the upper part of Listing 15.15. The rest of the methods are left as exercises.

The List interface

The **List** interface in the Sun standard library is a sub-interface of Collection (which corresponds to a class extension). It has ten methods in addition to those of Collection, as follows. The first five specify the index where the action is to take place. You may also create a ListIterator that starts at the position whose index is `itsIndex`. These List methods throw Exceptions if the index values are out of range.

- `someList.get(indexInt)` returns the Object at that index.
- `someList.set(indexInt, someObject)` returns the Object that was replaced.
- `someList.add(indexInt, someObject)` inserts the Object at the specified index.
- `someList.remove(indexInt)` removes and returns the Object at that index.
- `someList.addAll(indexInt, someCollection)` adds the entire Collection at the specified index and returns `true`.
- `someList.indexOf(someObject)` returns the first index at which the Object occurs; it returns -1 if the Object is not in the list.
- `someList.lastIndexOf(someObject)` returns the last index instead.
- `someList.subList(fromInt, toInt)` returns a List containing the elements at index `fromInt` on up to but not including `toInt`.
- `someList.listIterator()` returns a new iterator, ready to start at the beginning of the list.
- `someList.listIterator(indexInt)` returns a new iterator ready to start at the specified index, so that `next()` produces `get(indexInt)`.

Exercise 15.48 Write the TwoWaySequence method `public boolean isEmpty()`.
Exercise 15.49 Write the TwoWaySequence method `public int size()`.
Exercise 15.50 Write the TwoWaySequence method `public boolean add (Object ob)` to add ob at the end of the sequence.
Exercise 15.51 Write the TwoWaySequenceIterator method `public Object next()`.
Exercise 15.52 Write the TwoWaySequenceIterator method `public void remove()`.
Exercise 15.53* Write the TwoWaySequence method `public void clear()`.
Exercise 15.54* Write the constructor for the TwoWaySequence class that has a String parameter naming the file from which String values are read.
Exercise 15.55* Write the TwoWaySequence method `public boolean remove (Object ob)`.
Exercise 15.56* Write the TwoWaySequenceIterator method `public void set (Object ob)`.
Exercise 15.57** Add `push`, `pop`, and `peekTop` methods to TwoWaySequence to provide full Stack capabilities. Code them to execute as fast as possible.
Exercise 15.58** Write the NodeSequence method `public Object previous()` to be added to Listing 15.12 (no references to the previous node are stored in any node). You will need a loop to find the node before the current node.

*15.11 About AbstractList and AbstractCollection (*Sun Library)*

If you want to write an implementation of the Collection class, you have to code 13 methods plus some constructors. If you want to write an implementation of the List class, you have to code 23 methods plus some constructors. The AbstractCollection and AbstractList classes are intended to save you most of that trouble.

The AbstractCollection class

AbstractCollection is a class in `java.util` that implements the Collection interface. If you declare a class to be a subclass of AbstractCollection, you only have to write the `iterator` method and the `size` method (overriding those abstract methods in the AbstractCollection class). Your iterator must implement `hasNext` and `next`, though it can leave `remove` to throw an UnsupportedOperationException.

Generic coding for all the other Collection methods is provided for you. It uses the iterator method you provide. For instance, the coding for `contains` might be as follows. You may override this and the other pre-coded methods for efficiency:

```
public boolean contains (Object ob)
{  Iterator it = this.iterator();
   while (it.hasNext())
      if ((ob == null && it.next() == null)
            || (ob != null && ob.equals (it.next())))
         return true;
   return false;
}  //======================
```

If you want your subclass of AbstractCollection to be modifiable, you have to code the `add` method for the AbstractCollection and the `remove` method for the Iterator.

The AbstractList class

AbstractList is a class in `java.util` that implements the List interface (specified in the preceding section). If you declare a class to be a subclass of AbstractList, you only have to write the `get` method and the `size` method (overriding those abstract methods in the AbstractList class). Coding for all the other List methods is provided for you, though the basic methods that modify the List object throw an UnsupportedOperationException. You may override any of the pre-coded methods for efficiency if you want.

If you want your subclass of AbstractList to be modifiable, you have to override one or more of `set(int,Object)`, `remove(int)`, and `add(int,Object)`, since these are the only methods that throw an UnsupportedOperationException. The AbstractList class provides a ListIterator implementation on top of the methods you provide.

The Sun standard library provides the **ArrayList** class, which is a complete implementation of List using an array. Use this class for situations where it is not worthwhile to develop your own implementation tailored to a particular piece of software. Its use is illustrated in Section 7.11.

15.12 Review Of Chapter Fifteen

About the Java language:

- You may declare a class inside of another class X, called a **nested class**. If the word "static" appears before "class", this is no different from declaring it outside X except for visibility: Private variables of X are accessible in the nested class, and public variables of the nested class are accessible in X. The nested class itself is not accessible to classes outside of X. For statements and declarations inside X, the nested class shadows (supercedes) any outside class of the same name.
- A class of objects with an instance variable of that same class is called **naturally recursive**. Naturally recursive nodes are used to form a **linked list**. An extra node with no data at the beginning of the list is a **header node**; it simplifies the coding.

About the java.util.Collection interface:

- A **Collection** of **elements** may not allow duplicates (in which case it is a **java.util.Set** kind of object) or may not guarantee a particular ordering or may not allow `null` (and so throw a **java.lang.IllegalArgumentException** if you try to add `null`). The methods that modify the Collection may be **optional**, which means they may throw a **java.lang.UnsupportedOperationException**. A **sequence**, as the term is used in this book, is a Collection for which a particular order is maintained, duplicates are allowed, but `null` is not allowed.
- `someCollection.size()` returns the number of elements in the executor.
- `someCollection.isEmpty()` tells whether the executor has any elements.
- `someCollection.clear()` removes all elements from the executor.
- `someCollection.iterator()` returns an Iterator over the executor's elements.
- `someCollection.toArray()` returns an array containing the executor's elements.
- `someCollection.toArray(anArrayOfObjects)` returns an array containing the executor's elements. It will be the array parameter if the parameter has room (with `null` at the end if room), otherwise it will be a newly-created array.
- `someCollection.containsAll(aCollection)` tells whether the executor contains every element of the parameter.
- `someCollection.contains(anObject)` tells whether `anObject` is one of the executor's elements.
- `someCollection.add(anObject)` adds `anObject` to the Collection unless `anObject` is already in there and the Collection does not allow duplicates. It returns `true` if and only if the Collection changed, as do the other four methods listed below.
- `someCollection.remove(anObject)` deletes one instance of `anObject` from the executor unless `anObject` was not in there in the first place.
- `someCollection.addAll(aCollection)` in essence repeatedly executes `add` for each element of the parameter.
- `someCollection.removeAll(aCollection)` in essence repeatedly executes `remove` for each element of the parameter.
- `someCollection.retainAll(aCollection)` in essence repeatedly executes `remove` for each element of the executor that the parameter does not contain.

About the java.util.Iterator interface:

- `someIterator.next()` returns the next available element. A call of `next()` throws a **java.util.NoSuchElementException** if called when none is available. Repetition of `next()` calls produces each element of the Collection one time.
- `someIterator.hasNext()` tells whether `next` has more elements available.
- `someIterator.remove()` removes the element most recently returned by `next`. It throws a **java.lang.IllegalStateException** if that element has already been removed or if `next` has not been called. It throws a java.lang.UnsupportedOperationException if removal is not allowed.

About the java.util.List interface:

- The List interface extends the Collection interface, adding the following ten methods:
- `someList.get(indexInt)` returns the Object at that index.
- `someList.set(indexInt, someObject)` puts `someObject` at the index `indexInt` and returns the Object that it replaces.
- `someList.add(indexInt, someObject)` inserts the Object at the specified index.
- `someList.remove(indexInt)` removes and returns the Object at that index.
- `someList.addAll(indexInt, someCollection)` adds the entire Collection at the specified index and returns `true`.
- `someList.indexOf(someObject)` returns the first index at which the Object occurs; it returns -1 if the Object is not in the list.
- `someList.lastIndexOf(someObject)` returns the last index instead.
- `someList.subList(fromInt, toInt)` returns a List containing the elements at index `fromInt` on up to but not including `toInt`.
- `someList.listIterator()` returns a new iterator, ready to start at the beginning of the list.
- `someList.listIterator(indexInt)` return a new iterator ready to start at the specified index, so that `next()` produces the value `get(indexInt)`.

About the java.util.ListIterator interface:

- The ListIterator interface extends the Iterator interface, adding the following six methods. The `remove` and `set` methods delete/replace the element most recently returned by `next` or `previous`, except they throw a java.lang.IllegalStateException if `next` has not yet been called, or if neither `next` nor `previous` has been called since the last call of `remove` or `add`.
- `someListIterator.add(someObject)` puts `someObject` just before the element that `next` would return, or at the end if `hasNext()` is `false`.
- `someListIterator.set(someObject)` puts `someObject` in place of the element that the most recent call of `next` or `previous` returned.
- `someListIterator.nextIndex()` returns the zero-based index of the element that a call of `next` would return; it returns the number of elements if `hasNext()` is `false`.
- `someListIterator.previous()` returns the element immediately before the element `next` would return; it returns the last element if `hasNext()` is `false`. It throws a java.util.NoSuchElementException if such an element does not exist.
- `someListIterator.hasPrevious()` tells whether `previous` has more elements available to be returned.
- `someListIterator.previousIndex()` returns 1 less than what `nextIndex()` returns.

Answers to Selected Exercises

15.1
```
public int size()
{   return itsSize;
}
```
15.2
```
public boolean isEmpty()
{   return itsSize == 0;
}
```
15.3
```
public boolean contains (Object ob)
{   for (int k = 0;  k < itsSize;  k++)
        if (itsItem[k].equals (ob))
            return true;
    return false;
}
```
15.4
```
public Object[] toArray()
{   return copyOf (itsItem, 1);
}
```
15.8
```
public boolean isEmpty()
{   return itsFirst == null;
}
```
15.9
```
public int howManyEqual (Object ob)
{   int count = 0;
    for (Node p = this.itsFirst;  p != null;  p = p.itsNext)
        if (p.itsData.equals (ob))
            count++;
    return count;
}
```
15.13 Replace the if statement in the body of the for-statement by the following:
```
if (loc == null)
    return true;
else if (! p.itsData.equals (loc.itsData))
    return false;
```
15.14
```
public Object[] toArray()
{   Object[] valueToReturn = new Object [this.size()];
    int count = 0;
    for (Node p = this.itsFirst;  p != null;  p = p.itsNext)
    {   valueToReturn[count] = p.itsData;
        count++;
    }
    return valueToReturn;
}
```
15.15
```
public void removeEvens()     // in Node
{   for (Node p = this;  p != null && p.itsNext != null;  p = p.itsNext)
        p.itsNext = p.itsNext.itsNext;
}
```
15.16
```
public NodeSequence (Object[ ] given)
{   super();
    itsFirst = null;
    for (int k = given.length - 1;  k >= 0;  k--)
        if (given[k] != null)
            itsFirst = new Node (given[k], itsFirst);
}
```
15.18
```
public boolean contains (Object ob)
{   return isIn (itsFirst, ob);
}
private static boolean isIn (Node pos, Object ob)
{   return pos != null && (pos.itsData.equals (ob) || isIn (pos.itsNext, ob));
}
```
15.19
```
public void removeEvens()  // in Node
{   if (itsNext != null)
        itsNext = itsNext.itsNext;
    if (itsNext != null)
        itsNext.removeEvens();
}
```
15.21 Put the following statement before the for-statement:
```
Object valueToReturn = itsItem[itsPos];
```
Put the following statement at the end of the method body:
```
return valueToReturn;
```

15.22
```
public boolean containsAll (Collection that)
{     Iterator it = that.iterator();
      while (it.hasNext())
            if ( ! this.contains (it.next()))
                  return false;
      return true;
}
```
15.23 Replace the while statement by the following:
```
Iterator inven = inventory.iterator();
Object stopper = inven.hasNext()  ?  inven.next()  :  null;
while (it.hasNext())
{     Object data = it.next();
      if ( ! data.equals (stopper))
            System.out.println (data.toString());
      else
            stopper = inven.hasNext()  ?  inven.next()  :  null;
}
```
15.24 In Listing 15.8, remove the constructor, the declaration of itsSeq, and all uses of "itsSeq." in the coding of methods. An inner class can refer to itsSize and itsItem directly.
Also omit "static" from the class heading; that is what makes it an inner class.
```
public Iterator iterator()
{     return this.new BasicSequenceIterator();
}
```
15.31 Add another instance variable to the NodeSequenceIterator class, declared as follows:
```
private int itsIndex = 0;
```
Add one statement to the next method: itsIndex++;
Add one statement to the remove method: itsIndex--;
Add the following instance method:
```
public int nextIndex()
{     return itsIndex;
}
```
15.32
```
public NodeSequence (Collection that)
{     super();
      this.itsFirst = copyList (that.iterator());
}
private static Node copyList (Iterator it)   // uses recursion
{     return it.hasNext()  ?  new Node (it.next(), copyList (it))  :  null;
}
```
15.33 Initialize itsPos = null in the constructor. Replace the return statement in hasNext by:
```
return itsPos == null  ?  itsSeq.itsFirst != null  :  itsPos.itsNext != null;
```
Replace the next-to-last statement in next by:
```
itsPos = itsPos == null  ?  itsSeq.itsFirst  :  itsPos.itsNext;
```
Replace the last four statements in remove by the following (adding an "else"):
```
if (itsSeq.itsFirst  == itsPos)
      itsSeq.itsFirst = itsPos.itsNext;
else
      itsPrevious.itsNext = itsPos.itsNext;
itsPos = itsPrevious;
```
15.35
```
public void clear()
{     itsFirst = null;
}
```
15.36
```
public boolean addAll (Collection that)
{     Iterator it = that.iterator();
      boolean valueToReturn = false;
      while (it.hasNext())
            valueToReturn = this.add (it.next()) || valueToReturn;
      return valueToReturn;
}
```
15.37 1. Move addLater and removeLater to the NodeSequence class, changing the method headings as follows:
```
public static void addLater (Node current, Object ob)
public static boolean removeLater (Node current, Object ob)
```
2. Put "current." before each itsNext as a stand-alone variable in addLater or removeLater.
3. Replace the statements in add and remove that call these two methods by the following:
```
addLater (this.itsFirst, ob);
return removeLater (this.itsFirst, ob);
```
4. Replace the statements in addLater and removeLater that call these two methods by:
```
addLater (current.itsNext, ob);
return removeLater (current.itsNext, ob);
```

```
15.38   public boolean retainAll (Collection that)
        {   boolean valueToReturn = false;
            while (this.itsFirst != null && ! that.contains (this.itsFirst.itsData))
            {   this.itsFirst = this.itsFirst.itsNext;
                valueToReturn = true;
            }
            if (this.itsFirst != null)
            {   Node p = this.itsFirst;
                while (p.itsNext != null)
                    if (that.contains (p.itsNext.itsData))
                        p = p.itsNext;
                    else
                    {   p.itsNext = p.itsNext.itsNext;
                        valueToReturn = true;
                    }
            }
            return valueToReturn;
        }
15.42   public void set (Object ob)
        {   if (! isRemovable)
                throw new IllegalStateException();
            BasicSequence.this.itsItem[pos] = ob;
        }
15.43   public void push (Object ob)
        {   itsFirst = new Node (ob, itsFirst);
        }
        public Object pop ()
        {   if (itsFirst == null)
                throw new NoSuchElementException();
            Object valueToReturn = itsFirst.itsData;
            itsFirst = itsFirst.itsNext;
            return valueToReturn;
        }
15.48   public boolean isEmpty()
        {   return itsHead.itsNext == itsHead;
        }
15.49   public int size()
        {   int count = 0;
            for (Node p = itsHead.itsNext; p != itsHead;  p = p.itsNext)
                count++;
            return count;
        }
15.50   public boolean add (Object ob)
        {   if (ob == null)
                throw new IllegalArgumentException ("no nulls allowed");
            itsHead.itsPrevious = new Node (ob, itsHead, itsHead.itsPrevious);
            itsHead.itsPrevious.itsPrevious.itsNext = itsHead.itsPrevious;
            return true;
        }
15.51   public Object next()
        {   if (itsPos.itsNext.itsData == null)   // the sequence's header node
                throw new NoSuchElementException ("already at end");
            itsPos = itsPos.itsNext;
            itsDirection = 1;
            itsIndex++;
            return itsPos.itsData;
        }
15.52   public void remove()
        {   if (itsDirection == 0)
                throw new IllegalStateException ("cannot remove");
            if (itsDirection == 1)
            {   itsPos = itsPos.itsPrevious;
                itsIndex--;
            }
            itsDirection = 0;  // so it cannot be removed again
            itsPos.itsNext = itsPos.itsNext.itsNext;
            itsPos.itsNext.itsPrevious = itsPos;
        }
```

16 Maps And Linked Lists

Overview

The program you write in a text file is called the source code. You then have the compiler translate the source code to object code, stored in another file. This object code is some computer chip's machine code (for Java, it is the machine code of the Java Virtual Machine). By contrast, an **interpreter** translates one line of source code at a time to object code in RAM, executing the result before going on to the next line.

You have been hired to develop a user-friendly interpreter for the Scheme programming language. In the process of developing this software, you will learn how to work with linked lists using recursive logic, and you will see the basis for several different implementations of the Map class from the Sun standard library.

This chapter requires that you have a solid understanding of arrays and have learned about recursion, Exceptions, and interfaces. No new Java language features are in this chapter or in any later chapter.

- Sections 16.1-16.2 give a partial development of the Scheme interpreter with four classes implementing the Item interface.
- Section 16.3 presents the standard Sun library Map interface and a home-grown subset of it called Mapping. It also discusses the Iterator interface.
- Sections 16.4-16.6 show how a Mapping can be implemented using a partially-filled array and/or a linked list, including the nested classes implementing Iterators.
- Section 16.7 introduces the concept of a hash table to implement a Mapping.
- Section 16.8 develops the Scheme interpreter further using a Mapping.

16.1 Basic Scheme Language Elements

The client has the manual for the Scheme language, which is an exact specification in all details. The client wants precisely what is described there with certain exceptions, such as the error messages being far more helpful than what the manual specifies. Your job is to develop the interpreter the way the manual describes, to a substantial extent, before considering modifications.

The manual says that the Scheme interpreter waits for the user to enter a line of input and then prints a response. It gives these examples of inputs and responses (==> indicates the response for the given user input):

```
6.40                ==> 6.4
"Hello"             ==> Hello
"He \"finks\" out"  ==> He "finks" out
pi                  ==> 3.14159
```

These examples illustrate the **atomic** kinds of things in the Scheme language: a real number, a TextString, and a WordUnit.

- A **real** number is the same as Java's ordinary double value: one or more digits, with an optional decimal point before any one of them, and an optional negative sign at the very beginning. Integers are not a separate class of values from reals.
- A **TextString** is the same as Java's String value, beginning and ending with a quote. You may use `\"` to indicate a quote mark inside the string, `\n` to indicate a newline, `\\` to represent a backslash, etc.
- A **WordUnit** is most anything else that does not involve parentheses, apostrophes, quotes, or spaces. WordUnits are used as names of variables and functions.

Function calls

A list of atomic items inside a matched pair of parentheses has a value that can be computed. Here are some examples of addition, multiplication, subtraction, and finding the square root (we slightly simplify the true Scheme description):

```
(+ 5 3)        ==> 8
(* 5 3)        ==> 15
(- 5 3)        ==> 2
(sqrt 9)       ==> 3
```

In most cases, a WordUnit at the beginning of the list indicates an operation to be carried out on the rest of the values in the list. Each such **list** is essentially a method call, where the method name is `+` or `sqrt` or whatever the initial WordUnit is, and the other values in the list are the actual parameters of the method call.

A method is called a **function** in Scheme, and actual parameters are called **arguments**. Every Scheme function returns a value, i.e., there are no void methods in Scheme.

Assignment to variables

Scheme allows you to assign values to WordUnits, so you can use them like variables in Java. The assignment WordUnit `set!` is the **keyword** used for this purpose:

```
(set! x 5)     ==> x is defined
x              ==> 5
(/ 20 x)       ==> 4
val_3          ==> oh-oh: val_3 has no value
(set! val_3 x) ==> val_3 is defined
val_3          ==> 5
```

From these examples you can see that the `set!` command requires two parameters. It assigns to the first parameter, a WordUnit, the value of the second parameter. The value returned by the `set!` command is just `whatever is defined`. Some WordUnits have predefined values (such as `pi` and `true` and `false`). A WordUnit that is not predefined by Scheme and not yet defined by the user produces an **oh-oh error message** when you ask the interpreter to find its value.

The Scheme interpreter accepts one expression at a time containing either an atomic thing (WordUnit, TextString, or real) or a list (in parentheses), evaluates it, and then prints its value. If the expression the user enters is a `set!` command, the interpreter simply adds the variable (`x` and `val_3` in the preceding examples) to its list of defined variables, along with the value defined for it, and returns `whatever is defined`. Here are some examples of illegal expressions:

```
(6 2)    ==> oh-oh: a function call must start with a WordUnit
(bob 9)     ==> oh-oh: bob is not a function name
(sqrt 4 9)  ==> oh-oh: sqrt has too many parameters
```

Only a WordUnit can be used as a function name, and then only if it has been previously defined as a function. Also, you have to have exactly the number of parameters that the function definition requires. When the Scheme interpreter sees that the line the user entered is wrong, it stops processing the line and produces an oh-oh message.

The things on a list can be called its **elements**. A list can contain lists as its elements as well as atomic things (in other words, an argument of a function call can be a function call, just as in Java). The value of a list whose first element is a function name is the result of applying the function to the values of the rest of the elements of the list:

```
(* 3 (+ 5 2))              ==> 21
(sqrt (/ 50 2))           ==> 5
(+ (sqrt 9) (* 2 4))      ==> 11
```

Defining new functions

But of course, this is not programming, it is just using a fancy calculator. Programming requires that you have a way of writing new functions. Scheme provides that with the `define` keyword. The following Scheme coding defines functions for the cube of a value and the average of two values and illustrates their use:

```
(define (cube p)  (* p (* p p)))  ==> cube is defined
(cube 3)                          ==> 27
(cube val_3)                      ==> 125
(define (avg x y) (/ (+ x y) 2))  ==> avg is defined
(avg 6 13)                        ==> 9.5
```

The general format of a **function definition** is a list of three elements: The WordUnit `define`, then the function heading (a list of WordUnits), then the function body (any legal expression to be evaluated). For instance, a function definition with two parameters always has this form:

```
(define (functionName par1 par2)  anExpressionToEvaluateLater)
```

This list is <u>not</u> a function call. If it were, the interpreter would try to evaluate the second and third elements on the list. Instead, the interpreter stores the two elements away unevaluated for now, so it can use them to evaluate any number of calls of that function later in the program. Similarly,

```
(set!  variableName  anExpressionToEvaluateNow)
```

is not a function call. This non-evaluation of the expressions in a list beginning with `set!` or `define` is why `set!` and `define` are keywords.

Comparison with Java

Pause a moment to admire the simplicity and beauty of this programming language (which is taugyt in the first computer science course in many universities). The function call `(sqrt p)` corresponds to a Java method call `sqrt(p)` and the function call `(avg 6 13)` corresponds to a Java method call `avg(6,13)`. It may seem strange that addition is `(+ 2 3)` instead of `2 + 3` but it provides uniformity and simplicity: `+` is considered a function name just as are `sqrt` and `avg`. Moreover:

- Scheme has no other punctuation, such as semicolons, brackets, or periods.
- Precedence of operators does not even arise in Scheme.
- The function definition describes the value to be returned, so you do not have to say `return`.
- You do not bother with classes or public vs. private or instance vs. class: All variables and methods are public class methods and variables of one giant class, which you need not explicitly define (actually, this is a big drawback for advanced programming situations, since you do not have encapsulation).
- Scheme has an if-statement, a looping mechanism, boolean functions, etc., which we do not discuss here.
- www.htdp.org has a complete online textbook for Scheme and a free interpreter.

Exercise 16.1 Define a Scheme function that returns the Fahrenheit temperature corresponding to a given Centigrade temperature. Give an example of its use, as shown in the text for `cube` and `avg`.

Exercise 16.2 Define a Scheme function that returns the length of the hypoteneuse of a right triangle given two parameters, the two sides of the right triangle. Give an example of its use.
Exercise 16.3* Define a Scheme function that returns the area of a circle given its radius. Give an example of its use.
Reminder: The answers to unstarred exercises are at the end of the chapter.

16.2 Design Of The Scheme Interpreter

You now have enough information to begin the design of your Scheme interpreter. The main logic can display a frame for the graphical interface with a textfield for the input and a textarea for the output. The accompanying design block contains a reasonable design for the `actionPerformed` method that responds to the ENTER key within the textfield and prints the results.

STRUCTURED NATURAL LANGUAGE DESIGN of actionPerformed
1. Get the string of characters from the textfield.
2. Convert that string to a WordUnit, TextString, real, or list, whichever kind of item it represents.
3. If the item is a list with the `set!` or `define` keyword then...
   Update the set of variable or function definitions.
4. Find the value of the item.
5. Print the string representation of that value in a textarea.
6. BUT, if any exceptional situation arises in Steps 2 through 5 then...
   Print an oh-oh! message explaining the problem.

Object design

What objects are needed to implement the main logic? Three objects needed for the graphical user interface are a JFrame object, a JTextField object, and a JTextArea object. For the Scheme language itself, it appears good to have WordUnit objects, TextString objects, Real objects, and List objects.

Since these four kinds of objects can all appear as elements of a List, you should have a class or interface from which those four classes inherit. Call it Item. Since there will be no actual Item instances, only instances of the four kinds of Items, an Item interface seems best. It gives the public instance method headings the four classes have in common, with a semicolon in place of each method body.

Step 4 of the structured design only requires a single Java statement if each of the four Item subclasses has a `getValue()` method that returns the value of that kind of Item. Step 5 of the design also requires only a single Java statement if you have an appropriate `toString()` method for each kind of Item. Listing 16.1 (see next page) gives the design of the Item interface and its implementations so far. The bodies of the methods in the four implementations are stubbed for now.

Step 2 of the structured design seems to require some kind of parser object with a method that accepts one line of characters as input and produces the corresponding Item. That parser needs to call on a constructor for each of the four kinds of Item. The constructor for a Real object is passed a double value, and the constructor for a TextString or WordUnit is passed the String of characters forming it. The tricky part is the constructor for a List object. Figure 16.1 shows the set of all object classes so far.

Listing 16.1 Item and its four implementors, stubbed form

```
public interface Item
{
   /** Return the Scheme value of the Item, which is the current
    *  value if a variable, the evaluation if a function call. */

   public Item getValue();

   /** Return the way the Item is presented to the user
    *  on the screen. */

   public String toString();
}
//#################################################################


public class WordUnit implements Item
{
   public WordUnit (String given)              { }
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}


public class TextString implements Item
{
   public TextString (String given)            { }
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}


public class Real implements Item
{
   public Real (double given)                  { }
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}


public class List implements Item
{
   public Item getValue()           {return null; }
   public String toString()         {return null; }
}
```

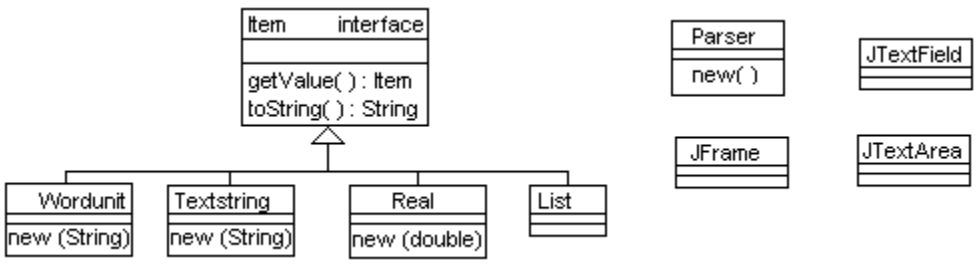


Figure 16.1 UML diagram of object classes needed for the interpreter

The TextString and Real classes

The simplest of the four kinds of Item to implement is TextString. If the constructor simply stores the String parameter away in an instance variable named `itsValue`, the `toString` method can simply return that value when requested. Listing 16.2 shows the complete TextString class.

Listing 16.2 The TextString class

```
public class TextString implements Item
{
   private String itsValue;


   public TextString (String given)
   {  itsValue = given;
   }  //=====================

   public Item getValue()
   {  return this;
   }  //=====================

   public String toString()
   {  return itsValue;
   }  //=====================
}
```

The Real class is not much harder. It should have a double value as its sole instance variable; call it `itsValue`, just as for TextStrings. For function calls such as `+` and `/`, you will need to retrieve the double value from the Real object so you can add or divide the numbers. So the Real class needs a `getNumber` method. These methods are left as exercises.

Top-level implementation

Finding the value of an Item is done by calling its `getValue` method. The interpreter's main logic should have a variable of type Item for which it calls the `getValue` method. This lets the runtime system call the `getValue` method of the appropriate implementation. This is a polymorphic call: The runtime system determines the type of the Item at that point in the execution of the program.

If an Item variable refers to a List object, the call of the `getValue` method in Step 4 of the main logic checks that it is a non-empty list, that the first element on the list is a WordUnit, and that the WordUnit has a meaning. However, all of this also has to be done by Step 3 of the main logic, which determines whether the user's input is a List object beginning with `set!` or `define`. This is duplication of effort. So it seems best to omit Step 3 of the main logic entirely, letting List's `getValue` method do that job.

Throwing an Exception

The parser object could find a syntax error in the input at any of several different stages. And if the expression parses correctly, it may still have a **semantic** error such as the wrong number of parameters for a function, an undefined function name, etc. So errors can arise at many points in the processing of one line of input. Wherever an error arises, you want to terminate all processing back to the main logic and print an oh-oh message. This is a perfect situation for throwing an Exception.

You can create a BadInput subclass of the standard library RuntimeException class. The top part of Listing 16.3 gives the standard way of writing such a subclass of Exception: two constructors and nothing else. Then any time a method finds an error, it need only execute `throw new BadInput (someMessage)` to have all processing terminate back to the `actionPerformed` method. That method contains the try/catch statement that catches all such throws of BadInput Exception objects and displays "oh-oh" followed by that message (Step 6 of the structured design is this try/catch logic).

Listing 16.3 The BadInput class and the main logic

```
public class BadInput extends RuntimeException
{
   public BadInput ()
   {  super();
   }  //====================

   public BadInput (String message)
   {  super (message);
   }  //====================
}
//#############################################################



// the actionPerformed method for the JTextField's ActionListener

   private JTextField field;
   private JTextArea output;
   private Parser parser;

   public void actionPerformed (ActionEvent event)
   {  output.append ("\n" + field.getText() + "    ==> ");
      try
      {  Item userInput = parser.parseInput (field.getText());
         output.append (userInput.getValue().toString());
      }catch (BadInput e)
      {  output.append ("oh-oh: " + e.message());
      }
   }  //====================
```

Converting the string of characters that the user enters into a Scheme expression made up of the various kinds of Items will be fairly complex (though much less so than if it were a Java statement or method definition). So an entirely separate class to convert the string (**parse** it) should be used, with probably only one public method named `parseInput`. This is left as a major programming project.

You now know enough to understand the `actionPerformed` method that reacts to the user's pressing the ENTER key inside the JTextField for the next expression to be processed. The implementation in Listing 16.3 assumes that `field`, `output`, and `parser` are instance variables of a class containing the `actionPerformed` method.

Exercise 16.4 Write the constructor and the `getValue` method for the Real class, assuming a single double instance variable named `itsValue`.
Exercise 16.5 Write the `toString` and `getNumber` method for the Real class under the same assumption.
Exercise 16.6 Write the two statements in the try part of Listing 16.3 as one statement.

16.3 The Map Interface And The Mapping Interface

You need to store the names of Scheme variables that have been defined along with their values. At other times you need to be able to look up the name of a variable to see if it has a value and, if so, what that value is. Also, you need to store the names of functions that have been defined along with their definitions and later look them up. What you need is a Map. The Map interface is part of the Sun standard library, in package `java.util`. It replaces the Dictionary abstract class that was in an earlier version of Java.

A **Map** is a collection of pairs of objects, the first being the ID of the second. An example of one kind of pair is a Social Security Number (SSN) plus the Worker with that SSN. Another example is a dictionary word and its meaning. In general, the ID is called the **key** and the other object is the **value** for that key. Different values must have different keys. Figure 16.2 should give you some idea of what the basic Map methods mean:

- You can add a new Worker to the SSN/Worker Map if you supply its SSN, as in `someMap.put (herSSN, worker)`.
- You can find out whether there is a Worker in the SSN/Worker Map with a given SSN: `someMap.containsKey (herSSN)` returns `true` if `herSSN` is stored as a key, `false` if not.
- You can look up a Worker in a SSN/Worker Map by giving the Social Security Number, as in `worker = someMap.get (herSSN)`; it returns null if there is no Worker with that SSN in the Map.
- You can take a particular SSN/Worker pair out of the Map when you specify the SSN to look for: `someMap.remove (herSSN)` returns the removed value.

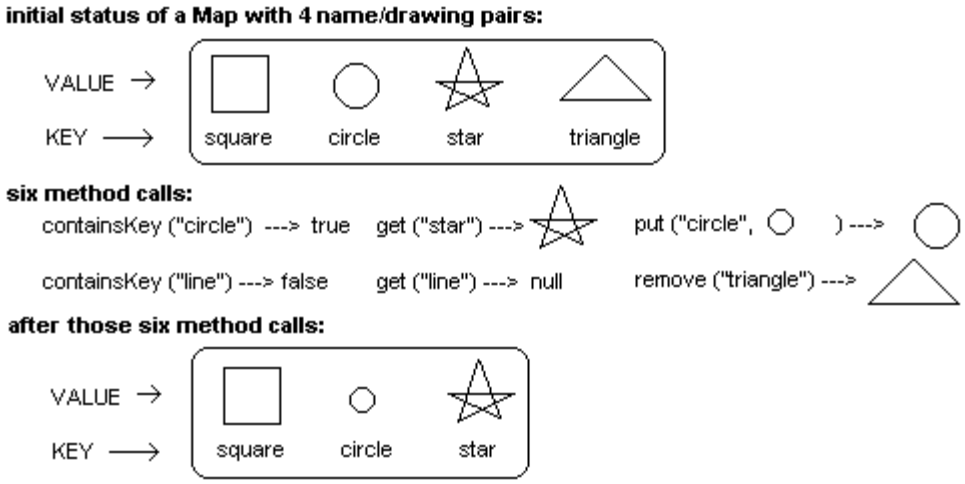


Figure 16.2 Meaning of Map methods; ---> shows what is returned

The Mapping interface

The Sun standard library Map interface prescribes twelve methods that must be implemented. Most implementations do not need all of them. Some of them return Sets or Collections, which for simplicity we do not wish to discuss here. So for most of this chapter, we use a stripped-down form called **Mapping**. This interface is given in the upper part of Listing 16.4 (see next page). Note that it ignores a key/value pair with a null key, although it allows a key/value pair with a null value. By contrast, the standard Map either accepts null or throws an Exception when you try to add null.

Listing 16.4 The Mapping and Iterator interfaces

```
public interface Mapping // simplified Map, specific to this book
{
   /** Put this key/value pair in this Mapping if id is not null.
    *  If some existing key/value pair has key.equals(id),
    *  replace the value, otherwise add the key/value pair.
    *  Return the previous value, or null if there was none. */
   public Object put (Object id, Object value);

   /** Tell whether a key/value pair satisfies key.equals(id). */
   public boolean containsKey (Object id);

   /** Return the value in the key/value pair where
    *  key.equals(id).  Return null if no such key/value pair. */
   public Object get (Object id);

   /** Take out the key/value pair where key.equals(id), if any.
    *  Return the existing value, or null if there was none.  */
   public Object remove (Object id);

   /** Tell whether this data structure has no elements. */
   public boolean isEmpty();

   /** Tell how many elements are in this data structure. */
   public int size();

   /** Return an object that can be used to list all the
    *  elements in this data structure one at a time. */
   public java.util.Iterator iterator();
}
//#############################################################


public interface Iterator  // in java.util
{
   /** Tell whether any more elements can be obtained. */
   public boolean hasNext();

   /** Return the next element.
    *  Throw NoSuchElementException if hasNext() is false. */
   public Object next();

   /** Remove the element most recently returned by next.
    *  This operation is optional -- the implementation may
    *  throw UnsupportedOperationException if not available. */
   public void remove();
}
```

The first six methods listed for the Mapping interface are all in the standard library Map interface. Mapping has in addition a method that returns an Iterator object. You use it to progress through the values in the data structure one item at a time in some order. The "official" Map interface provides an Iterator indirectly, in that one Map method returns a Set object which has a method that returns an Iterator. So the Mapping interface is essentially a subset of the "official" Map interface. The full Map interface from the Sun standard library is discussed in the last section of this chapter, along with two useful Sun library implementations of it. Until then, we use this book-specific Mapping only.

The Iterator interface

The **Iterator** interface is in the `java.util` package. It has the three methods described in the lower part of Listing 16.4, but the choice of whether to provide the ability to remove the current element is optional with the implementor. For this chapter, we will simply have the `remove` method throw an **UnsupportedOperationException** (from the `java.lang` package). An iterator is intended to be used in coding such as the following:

```
Iterator it = someMapping.iterator();
while (it.hasNext())
{  Object nextElement = it.next();
   // whatever processing of nextElement is appropriate
}
```

If you construct an Iterator object to use with a collection of objects that has N elements, then you may call `next()` exactly N times; it will give you the N elements one at a time. If you call `next()` N+1 times, it throws a **NoSuchElementException** (from the `java.util` package). Some sample Java logic for an outside class to print all the key/value pairs in a Mapping object is as follows:

```
public static void display (Mapping given)  // independent
{  Iterator it = given.iterator();
   while (it.hasNext())
      System.out.println (it.next());
}  //=====================
```

If you wanted to only print the values that are not equal to a particular value `badOne`, you would have to have the body of the method be as follows. You store `it.next()` in a local variable because each call of `it.next()` advances to the next value:

```
Iterator it = given.iterator();
while (it.hasNext())
{  Object nextElement = it.next();
   if ( ! nextElement.equals (badOne))
      System.out.println (nextElement);
}
```

Using Map methods in the WordUnit class

Finding the value of a WordUnit (usually a variable) has to be easier than finding the value of a List (a function call). More important, it should give some insight on the latter. So it is a good idea to start with that and work on the List class later (in Section 16.9).

Clearly you will need to store all the current values of all variables that have been assigned so far. So the WordUnit class should have a Mapping class variable named perhaps `theVariables`. The Mapping methods then allow a simple coding of the `getValue` method of the WordUnit class: If `theVariables` contains the given WordUnit as the key field of a key/value pair, the `getValue` method should return its value, otherwise it should throw a BadInput Exception. Note that you need to cast the Object value returned by `get` to have an Item value that can be returned:

```
public Item getValue()                          // in WordUnit
{  if (theVariables.containsKey (this))
      return (Item) theVariables.get (this);
   else
      throw new BadInput (this.toString() + " has no value");
}  //=====================
```

When the List class processes a variable definition, it needs to add a given variable/value pair to `theVariables`. So the WordUnit class needs a public method to provide that service. It only needs to be a simple wrapper around a single statement that tells `theVariables` to include the variable/value pair:

```
public void setVariableValue (Item value)     // in WordUnit
{  theVariables.put (this, value);
}  //======================
```

A Map is not supposed to allow two entries with the same key; if the key is already part of an existing key/value pair in `theVariables`, then the new pair replaces the old pair, which is exactly what you want to have happen when you assign a value to a variable. The rest of the WordUnit methods are much the same as for TextString and Real: The constructor is supplied the String of characters that are the written form of the WordUnit object. That String is stored in an instance variable called `itsValue`, which is the value returned by a call of `toString`.

Services provided by the List class

The `getValue` method in the List class finds the value of a function call or other List. First, though, you have to verify that it is legally-formed. A List is a sequence of a number of Items. For a legally-formed function call, the List object's first element must be a WordUnit that has been defined as a function name. Also, the List object must have the same number of arguments as the function heading has formal parameters.

It is clear that you need to be able to ask a List object for its first element so you can see if it is a WordUnit that has been defined. You also need a List method to retrieve the **sublist** of arguments, i.e., the rest of the List excluding that first element.

If `sam` is a non-empty List object, then `sam.first()` is the first Item in the List. `sam.rest()` is the sublist containing all elements of `sam` except its first element. Figure 16.3 illustrates their meaning for a List of six values. You will also need to be able to tell whether a List is empty; that is the condition `sam.isEmpty()`.

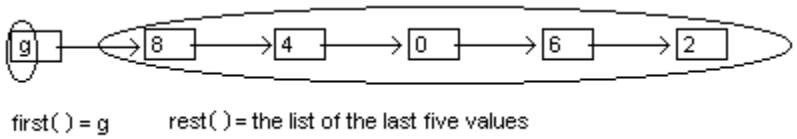


Figure 16.3 Meaning of first() and rest()

Note that, in developing a class such as List, you do not consider the (private) instance variables until after you have most of the public methods. The only reason for having an object know stuff (instance variables) is so it can do stuff (respond to method calls). Once you know what an object has to be able to do, and not before, you will know what the object has to know.

The `set!` and `define` keywords require special handling, as do the predefined functions such as `+` and `sqrt`. You need a different segment of Java coding to process each one. Finding the right code segment can be done easily if you have a different numerical index for each keyword and predefined function name. So the data structure where they are stored should let you look up the numerical index of a WordUnit. Given a WordUnit, first try searching in that data structure to retrieve its numerical index. If you do not find it, treat that WordUnit as a normal function name.

A normal function name should have a definition that has previously been stored, generally using the `define` keyword. The obvious data structure for storing these definitions is a Mapping, perhaps called `theDefs`. And the data structure in which you look up the numerical index of a keyword or predefined function should also be a Mapping, perhaps called `theSpecials`. These will be two class variables in List.

Since a Mapping requires that the value stored with a keyword (such as `set!` or `define`) be an object, not a numerical index, `theSpecials` can use a Real object rather than a numerical index. So the `getValue` method in the List class could include this statement (you need the cast because `get` returns an Object):

```
Real index = (Real) theSpecials.get (this.first());
```

If `this.first()` is found, and thus `index` is not null, you need to call a method to get the value by choosing the right segment of Java logic. Otherwise you need to call a completely different method for a normal List object. So the List class needs two private instance methods `specialValue(Real index)` and `normalValue()`.

For the following exercises, suppose the **Roget** class declares `theDictionary`, a class variable that is an implementation of Mapping. The key is a word and the value is its dictionary definition. Both the word and its definition are of type String. The Roget class also has instance variables `int itsSize` and `String[] itsItem`, with some words stored in array components `0...itsSize-1`. A precondition for all exercises is that no parameter is null unless explicitly stated or its type is Object.

Exercise 16.7 Write a Roget method `public static void inOrOut (String word, String definition)`: Put into `theDictionary` the given word and its definition if the given word is not there, otherwise take the word and its definition out.
Exercise 16.8 Write a Roget method `public boolean hasDefinition()`: The executor tells whether at least one of the words in `itsItem` is in `theDictionary`.
Exercise 16.9 Write a WordUnit method `public boolean equals (Object ob)`: The executor tells whether `ob` is a WordUnit object with the same `itsValue`.
Exercise 16.10 If x is the list of five elements `(a b c d e)`, then what is `x.first()? x.rest()? x.rest().first()?`
Exercise 16.11 Write a List method `public boolean hasThree()`: The executor tells whether it has at least three elements.
Exercise 16.12 (harder) Write a generic Mapping method `public void putAll (Mapping given)`: The executor puts all of the parameter's data in its own structure. Hint: Use an iterator to go through all the values in the `given` Mapping one at a time. For this exercise and the next, you need to know that the iterator produces MapEntry objects thate have a `getKey()` method and a `getValue()` method.
Exercise 16.13* Write a Roget method `public static String getDef (String one, String two)`: It returns the definition of the first parameter if it is in `theDictionary`, otherwise it returns the definition of the second (or null if neither is in `theDictionary`).
Exercise 16.14* Write a Roget method `public int getCount()`: The executor tells how many of the words in `itsItem` are in `theDictionary`.
Exercise 16.15* Write out the complete WordUnit class, assuming a single String instance variable named `itsValue`.
Exercise 16.16* If x is the list of three elements `((ab) (c d) (e f))`, then what is `x.first()? x.rest()? x.first().first()? x.rest().first().rest()?`
Exercise 16.17* Write a List method `public Item third()`: The executor returns its third element. It returns null if it has less than three elements.
Exercise 16.18* Write the Mapping method `public int size()` as a generic method. Hint: Use the Mapping's iterator to go through all the values one at a time.

16.4 Implementing The Mapping Interface With A Partially-Filled Array

The Scheme interpreter software will use an implementation of the Mapping interface named SchemeMap. We now look at how to implement the SchemeMap class, which requires implementing the seven methods described by the earlier Listing 16.4: `containsKey(id)`, `get(id)`, `remove(id)`, `put(id, value)`, `isEmpty()`, `size()`, and `iterator()`.

One way is with a partially-filled array of objects (i.e., only the first n components have useable values). Each object stored in the array is a key/value pair called a MapEntry. The MapEntry class in Listing 16.5 is a straightforward implementation of the Entry interface used for Maps in `java.util` (the `equals` method is complex only because you have to guard against putting a dot on null). Entry is declared as an interface nested in the standard library Map interface. So using the standard library means you must refer to it as `Map.Entry`; the dot is needed. Note that it is a class of immutable objects -- you cannot change the values of the instance variables once you have constructed the object.

Listing 16.5 The MapEntry class of objects

```
public class MapEntry implements java.util.Map.Entry
{
   private final Object itsKey;
   private final Object itsValue;


   public MapEntry (Object key, Object value)
   {  itsKey = key;
      itsValue = value;
   }  //====================


   public boolean equals (MapEntry given)
   {  return given != null
               && (this.itsKey == given.itsKey
                  || (this.itsKey != null
                      && this.itsKey.equals (given.itsKey)))
               && (this.itsValue == given.itsValue
                  || (this.itsValue != null
                      && this.itsValue.equals (given.itsValue)));
   }  //====================


   public Object getKey()
   {  return itsKey;
   }  //====================


   public Object getValue()
   {  return itsValue;
   }  //====================


   public Object setValue (Object ob)
   {  throw new UnsupportedOperationException ("can't setValue");
   }  //====================
}
```

The array for storing MapEntries contains key/value objects stored in components indexed 0 through `itsSize-1` of an array named `itsItem`. The `put` operation adds a key/value object at the end of the array, and the `get` operation searches through the array to find the key and return its corresponding value. Figure 16.4 shows MapEntry objects as musical notes for keys and their musical definitions as values.

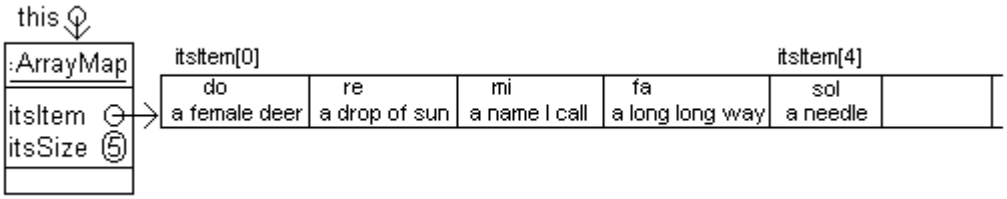


Figure 16.4 Representation of an ArrayMap object

We will be looking at several different ways to implement the SchemeMap class. So we will actually call this first one the ArrayMap class. You can use it for the Scheme interpreter with the following simple name-changing class definition (so `new SchemeMap()` actually makes a new ArrayMap object):

```
public class SchemeMap extends ArrayMap
{  // nothing is needed inside this name-changing class
}
```

The internal invariant for the ArrayMap class, which describes how the abstraction (a data base of MapEntry pairs) corresponds to the concrete (the instance variables) is therefore as follows:

- `itsItem` is an array of MapEntry values.
- `itsSize` is a non-negative int value which is the number of MapEntry pairs in the data structure. The data structure is empty when `itsSize` is zero.
- Each such MapEntry pair is in one of the components `itsItem[0]` through `itsItem[itsSize-1]`, in no specific order.

Implementing the ArrayMap class

Listing 16.6 (see next page) contains several methods of the ArrayMap class; the rest are left as exercises. The `put` method should refuse to add a MapEntry with a null key; the `containsKey` method relies on this refusal. The array length of 100 is arbitrary; an exercise provides for increasing the capacity of the array by 50% when needed. The values are not in any particular order; a major programming project keeps them in order.

The logic for `containsKey` should search through the array from the top down (i.e., higher indexes first), on the assumption that more-recently-defined key/value pairs are used more often. Several other Map methods have to search through the list of values for a particular key, and some of them have to know the exact position where it is found. So this coding uses a private `lookUp` method to find that position if it exists, and `containsKey` simply tells whether the position was found.

The `get` operation also uses the private `lookUp` method. It then returns the value at the position that the `lookUp` method finds (or returns null if it was not found).

Listing 16.6 The ArrayMap class of objects, some methods postponed

```
public class ArrayMap implements Mapping
{
   private MapEntry[] itsItem = new MapEntry [100];
   private int itsSize = 0;  // only has the default constructor


   /** Tell whether a key/value pair satisfies key.equals(id). */

   public boolean containsKey (Object id)
   {  return lookUp (id) >= 0;
   }  //=====================

   /** Return the value in the key/value pair for which
    *  key.equals(id).  Return null if no such key/value pair. */

   public Object get (Object id)
   {  int loc = lookUp (id);
      return (loc < 0)  ?  null  :  itsItem[loc].getValue();
   }  //=====================

   /** Return the index of the key/value pair for which
    *  key.equals(id).  Return -1 if no such key/value pair. */

   private int lookUp (Object id)
   {  int k = itsSize - 1;
      while (k >= 0 && ! itsItem[k].getKey().equals (id))
         k--;
      return k;  // -1 if the object is not in the list
   }  //=====================
}
```

A driver program

When you do the exercises, you will probably want to test out your logic. The program in Listing 16.7 (see next page) is a **driver program**, i.e., it "exercises" several ArrayMap methods in a way that should uncover most bugs in your logic. It will also exercise the methods of the NodeMap class that is discussed in the next section. Figure 16.5 is a UML diagram for it.

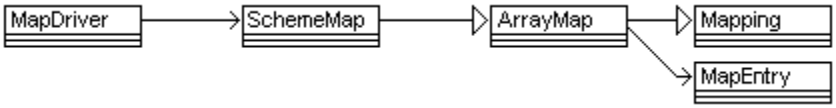


Figure 16.5 Partial UML class diagram for MapDriver

This driver program creates a SchemeMap object. The user inputs several word/definition pairs. Each is added to the SchemeMap, unless the word is already defined, in which case the definition is removed. Consequently, the Map object will contain all the word/definition pairs that have been entered an odd number of times. Exceptions: If the definition is the empty string or null, that is a signal to retrieve the current definition of the word and display it. And if the word is the empty string or null, the program terminates.

Exercise 16.19 Write the ArrayMap method `public int size()`.
Exercise 16.20 Write an ArrayMap method `public int countLess (Comparable id)`: The executor tells how many of its entries have a key that is less than a given key. Throw an Exception if the keys are not from a class that implements Comparable (and so has the usual `compareTo` method).

Listing 16.7 Driver program for an implementation of SchemeMap

```
import javax.swing.JOptionPane;

class MapDriver
{
   public static void main (String [ ] args)
   {  JOptionPane.showMessageDialog (null,
                "Repeatedly enter a word, then its definition.");
      WordProcessor.processManyWords (new SchemeMap());
      System.exit (0);
   }  //====================
}
//#########################################################


public class WordProcessor
{
   public static void processManyWords (Mapping odds)
   {  String word = JOptionPane.showInputDialog ("word?");
      while (word != null && word.length() > 0)
      {  String definition = JOptionPane.showInputDialog
               ("definition in/out?  ENTER key to see it: ");
         if (definition == null || definition.length() == 0)
            System.out.println ("its definition is currently "
                                + (String) odds.get (word));
         else if (odds.containsKey (word))
            System.out.println (word + " had the definition "
                                + (String) odds.remove (word));
         else
         {  odds.put (word, definition);
            System.out.println ("added " + word + " to the map; "
                                + odds.size() + " entries.");
         }
         word = JOptionPane.showInputDialog ("word?");
      }
   }  //====================
}
```

Exercise 16.21 Write the ArrayMap method `public Object remove (Object id)`: The executor first sees if the key is in the data structure. If not, nothing happens except that `remove` returns null. But if the key is there, the executor replaces that key/value pair by the one at the end of the array, makes the array one MapEntry smaller, and returns the value that corresponded to the key. Note that `id` could be null.

Exercise 16.22 (harder) Write an ArrayMap method `public ArrayMap reverse()`: The executor creates and returns a new ArrayMap object with a different array that has the same entries in it in the opposite order.

Exercise 16.23 (harder) Write the ArrayMap method `public Object put (Object id, Object value)`: The executor does nothing but return null if the given key is null. Otherwise it adds the key/value pair unless the key was already in the Map, in which case it replaces the pair. It returns null if no pair with that key was initially in the Map, otherwise it returns the value that corresponded to the key. It throws an IndexOutOfBoundsException when the array is full (improved in the next exercise).

Exercise 16.24* Modify the `put` method in the preceding exercise to verify that the array is not full; when it is, create a new array 50% larger and transfer the entries to it.

Exercise 16.25* Write an ArrayMap method `public boolean equals (Object ob)`: The executor tests whether the parameter is an ArrayMap object with the same key/value pairs in the same order as the executor.

Exercise 16.26* Write the ArrayMap method `public boolean isEmpty()`.
Exercise 16.27* Write an ArrayMap constructor with an ArrayMap parameter: Construct the new ArrayMap to have the same key/value pairs as the parameter, in the same order.
Exercise 16.28* Write an ArrayMap method `public void display()`: The executor prints all of its keys to `System.out`. This is useful when you are debugging a program.
Exercise 16.29* Omit the private `lookUp` method in ArrayMap. Instead, have a private class variable `thePosition` that `containsKey` sets to the correct location when it finds the value it is looking for. Then rewrite `containsKey`, `get`, `put`, and `remove` (the latter two are in the preceding exercises) to call on `containsKey` and, when the `id` is found, use the value of `thePosition`.
Exercise 16.30** Write a `put` method that keeps the values of the ArrayMap in ascending order. Assume keys are Comparable. Use binary search.
Reminder: Double-starred exercises are harder; the answers are not in the book.

16.5 Implementing The Mapping Interface With A Linked List

A more flexible way to implement the SchemeMap is as a linked list, similar to the linked lists in Scheme itself. As was done for the ArrayMap, the actual SchemeMap class will indirectly use a class named NodeMap (in place of the earlier definition of SchemeMap):

```
public class SchemeMap extends NodeMap
{  // nothing is needed inside this name-changing class
}
```

A NodeMap object will have two instance variables, `itsData` and `itsNext`. A NodeMap is conceptually a large number of **nodes** on the linked list where each node contains a single MapEntry (`itsData`) plus a reference to the next node on the linked list (`itsNext`). But in the last node (the only node if the map is empty), `itsData` is null and so is `itsNext`. This last node is called a **trailer node**.

The advantage of using a linked list is that you do not have to worry about (a) you crash the program because you made the array too small, or (b) you waste space because you made the array too big. Figure 16.6 shows an example of a linked list with trailer.

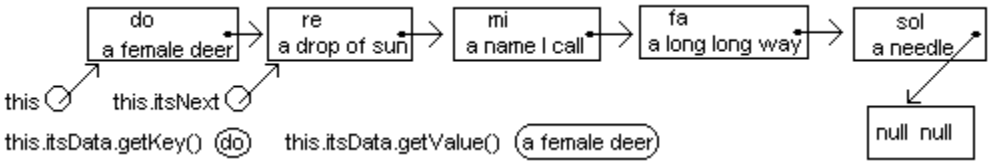


Figure 16.6 Representation of a NodeMap object

The containsKey method

A recursive logic for `containsKey` is now quite straightforward: There are two kinds of lists, an empty list and a nonempty list (this mantra reminds you that you always begin by considering these two alternatives). An empty list does not contain the key. A nonempty list `this` contains the key when and only when `this.itsData` is a MapEntry with that key or `this.itsNext` refers to a sublist that contains the key.

```
public boolean containsKey (Object id)   // in NodeMap
{  return ! isEmpty() && (this.itsData.getKey().equals (id)
                        || this.itsNext.containsKey (id));
}  //=====================
```

However, a private method to find the position where a given key appears would be more useful, since it could do most of the work for several different methods:

```
private NodeMap lookUp (Object id)       // in NodeMap
{  return this.isEmpty() || this.itsData.getKey().equals (id)
           ?  this  : this.itsNext.lookUp (id);
}  //=====================
```

Then `containsKey` only needs one simple return statement, telling whether `lookUp(id)` is empty. You will also find it profitable to study a non-recursive version of this `lookUp` logic, as follows:

```
private NodeMap lookUp (Object id)       // in NodeMap
{  NodeMap p = this;
   while ( ! p.isEmpty() && ! p.itsData.getKey().equals (id))
      p = p.itsNext;
   return p;  // the trailer node if it gets this far
}  //=====================
```

You should compare that logic with the following from Listing 16.6 step by step:

```
private int lookUp (Object id)           // in ArrayMap
{  int k = itsSize - 1;
   while (k >= 0 && ! itsItem[k].getKey().equals (id))
      k--;
   return k; // -1 if the object is not in the list
}  //=====================
```

You can see that `p` corresponds to `k` exactly:

- `p` or `k` indicates your current position in the sequence of values.
- `p` starts at the first position in the list (`this`) and `k` starts at the first position from the end (at `itsSize - 1`).
- The loop continuation condition requires that `p` or `k` not be beyond the last place where a MapEntry occurs (signaled by `p.isEmpty()` and `k < 0`, respectively).
- You move on by changing `p` or `k` to the following position.
- `p.itsData` corresponds to `itsItem[k]` exactly; either way, it is the MapEntry value at the current position. The rest of the logic is identical.

A NodeMap object represents a linked list, that is, a list that contains many entries. But a NodeMap object has only two instance variables `itsData` and `itsNext`, and only one of them is an entry. If X is the executor of a NodeMap method, then X's node and X's linked list are two different mental concepts. In the preceding `lookUp` method, `p` moves through the linked list by being initialized to refer to the first node of the linked list, then to the second node of the linked list, then to the third one, etc.

The get method, the size method, and the constructors

The natural constructor for the NodeMap class fills in its two instance variables with the corresponding parameter values. You also need a public constructor with no parameters; outside classes can use it to make an empty NodeMap object. That would of course be the empty list of values, which can be recognized by the fact that `itsNext` is null. These two constructors are in the upper part of Listing 16.8 (see next page). The natural constructor is private because it is only used internally to modify the data structure. Both are required, even though the public one looks like the default constructor.

Listing 16.8 The NodeMap class, some methods postponed

```
public class NodeMap implements Mapping
{
   private MapEntry itsData;          // trailer node logic
   private NodeMap  itsNext;


   private NodeMap (MapEntry data, NodeMap next)
   {  itsData = data;
      itsNext = next;
   }  //=====================

   public NodeMap()
   {  itsData = null;
      itsNext = null;
   }  //=====================


   public boolean isEmpty()
   {  return itsNext == null;
   }  //=====================

   public boolean containsKey (Object id)
   {  return ! lookUp (id).isEmpty();
   }  //=====================

   public Object get (Object id)
   {  NodeMap loc = lookUp (id);
      return loc.isEmpty() ?  null  :  loc.itsData.getValue();
   }  //=====================

   public int size()
   {  return this.isEmpty() ?  0  :  1 + this.itsNext.size();
   }  //=====================


   /** Return the node containing the key/value pair for which
    *  key.equals(id). Return the trailer node if no such
    *  key/value pair exists.  */

   private NodeMap lookUp (Object id)
   {  return this.isEmpty() || this.itsData.getKey().equals (id)
                 ?  this  :  this.itsNext.lookUp (id);
   }  //=====================
}
```

The executor of the `get` method should call `lookUp` to see if the `id` is already in the data structure. If so, it returns the corresponding value, otherwise it returns null. This can be written in Java using the same logic as in the earlier Listing 16.6 except that `itsItem[loc].getValue()` is replaced by `loc.itsData.getValue()`. This `get` method is in the middle part of Listing 16.8.

The `size` method is the simplest kind of recursion: An empty list has no data; a non-empty list has one piece of data (in `itsData`) plus whatever data is in the rest of the list. Listing 16.8 collects together the progress to date on this linked list implementation. Only the `put` and `remove` methods are left to be developed, which is done in the next section, and the `iterator` method, done in the section after that.

In the Scheme programming language, once you make a linked list, you cannot change it in any way. The List class is immutable the way the String class is. By contrast, you need to be able to modify the state of a NodeMap object.

The put method

The `put` method requires that you change the number of values in the linked list represented by a NodeMap object. But you can never change the value of the executor node by a method call (which is why a statement beginning `this=` is illegal). So you have to change the values of the instance variables of the nodes.

To put a new key/value pair in the NodeMap list with a non-null key, first you see whether there is already a MapEntry with that key. If so, you replace its value and return the previous value. Otherwise you want to add the key/value pair to the beginning of the list. But the executor `this` must continue to refer to the same node.

A workable solution is to create a new node that goes right after the first node, copy the two instance variables from the first to the second node, then put the new entry into the original first node. This logic is in the upper part of Listing 16.9. There are other logics that execute perhaps slightly faster, but they are moderately more complex.

Listing 16.9 The put and remove methods for the NodeMap class

```
// public class NodeMap, continued

   public Object put (Object id, Object value)
   {  if (id == null)   // Mappings ignore a null key        //1
         return null;                                        //2
      NodeMap loc = lookUp (id);                             //3
      if (loc.isEmpty())                                     //4
      {  this.itsNext = new NodeMap (this.itsData, this.itsNext);
         this.itsData = new MapEntry (id, value);            //6
         return null;                                        //7
      }                                                      //8
      Object valueToReturn = loc.itsData.getValue();         //9
      loc.itsData = new MapEntry (id, value);                //10
      return valueToReturn;                                  //11
   }  //=====================


   public Object remove (Object id)
   {  NodeMap loc = lookUp (id);                             //12
      if (loc.isEmpty())                                     //13
         return null;                                        //14
      Object valueToReturn = loc.itsData.getValue();         //15
      loc.itsData = loc.itsNext.itsData;                     //16
      loc.itsNext = loc.itsNext.itsNext;                     //17
      return valueToReturn;                                  //18
   }  //=====================
```

This logic works because a NodeMap is a linked list with a **trailer node**: When the linked list contains `N` values, it contains `N+1` nodes, the last one having null for both of its instance variables. Figure 16.7 shows the result of calling this `put` method twice.

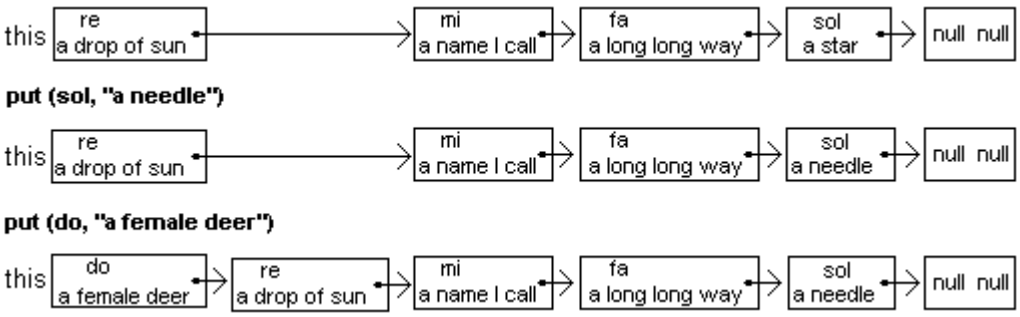


Figure 16.7 Result of two consecutive calls of put(Object, Object)

The remove method

For the `remove` method, if the linked list does not contain the key, then of course you simply return null. Otherwise, `lookUp` gives you a NodeMap value `loc` such that `loc's` node contains the MapEntry value that you want to remove. If it happens to be the first node in the linked list, you cannot delete the node because it is the executor itself (i.e., it is `this`). What you can do is delete the node after the one `loc` refers to. If you first move the MapEntry value from that deleted node up into `loc's` node, you have removed `loc`'s MapEntry value from the list.

Even if `loc` refers to some node other than the first one, you cannot easily delete that node itself from the linked list, because you have no convenient way to hook up the node before it to the node after it. So just do the same as at the first node: Move the MapEntry value in the following node up into `loc`'s node (which removes the entry you wanted to remove) and then remove the node after `loc`'s node from the linked list. This logic is in the lower part of Listing 16.9. Figure 16.8 shows the result of applying this `remove` logic to a linked list.

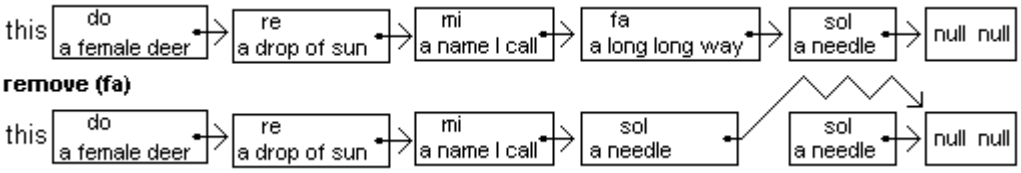


Figure 16.8 Result of a call of remove(Object)

You might think that this logic gives the wrong result when you are removing the last entry in the list, because there is no entry that comes after it. However, the empty list comes after it, consisting of a single node with both of its instance variables being null. So the `remove` logic simply copies both null values up to the current node. That makes the current node be the empty node at the end. The node that was last is no longer referred to by any other node, so garbage collection will recycle it eventually.

An alternative implementation using simple linked lists

Another way to implement a Mapping is to use a **simple linked list**, which is a linked list in which every node contains a data value (thus there is no trailer node). This saves a little space in RAM, but it makes the logic more complex. A Mapping object implemented this way contains just one instance variable `itsFirst` that is the first Node on a list, but is null when the list has no data. Node is a `private static class` **nested** within the Mapping class. That means that its methods can access the instance variables of a Node object but outside classes cannot (and thus encapsulation is maintained).

The `get` method has exactly the same coding as in Listing 16.8 except that the condition `loc.isEmpty()` is replaced by the condition `loc == null`. The `lookUp` method for a simple linked list could have the same logic as the non-recursive version in the previous section, except declare `Node p = this.itsFirst` and have the heading of the while-loop be the following:

```
while (p != null && ! p.itsData.getKey().equals (id))
```

Another alternative is to keep the data in ascending order of IDs, if the ID values are Comparable. This alternative is explored in the exercises. In any case, most methods in both ArrayMap and NodeMap execute in big-oh of N time since `lookUp` does.

Exercise 16.31 Write a NodeMap method `public Object secondOne()`: The executor returns the value part of its second entry, except it returns null if its map has less than two values.
Exercise 16.32 Write a NodeMap method `public void swapTwo()`: The executor swaps the first and second entries in its map, except it has no effect if its map has less than two values.
Exercise 16.33 Write a NodeMap method `public void clear()`: The executor makes the linked list empty.
Exercise 16.34 What changes would you make in the `put` method (Listing 16.9) to add the new MapEntry object directly after the first NodeMap object using `itsNext = new NodeMap (new MapEntry (id, value), itsNext)` whenever appropriate?
Exercise 16.35 (harder) Write a NodeMap method `public boolean equals (NodeMap given)`: The executor tells whether it has the same data values as the parameter in the same order. Use recursion. Precondition: `given` is not null.
Exercise 16.36 (harder) Suppose the keys in a NodeMap are in ascending order and are Comparable. Write a recursive `putRecursive` method that inserts the new MapEntry in a way that maintains order, called from `put` with the following statement: `return id == null ? null : putRecursive ((Comparable) id, value);`.
Exercise 16.37 (harder) Revise all NodeMap coding to keep the keys in ascending order. Assume that all keys are mutually Comparable. Hint: Have `lookUp` stop earlier.
Exercise 16.38 (harder) Suppose the keys in every NodeMap are in ascending order and are Comparable. Write a NodeMap method `public NodeMap intersection (NodeMap given)` that returns a new NodeMap that contains all MapEntries that are in both `given` and the executor. Use recursion. When two MapEntries have the same key, keep only the value that goes with the executor's key. Precondition: `given` is not null.
Exercise 16.39* Write a NodeMap method `public boolean isAscending()`: The executor tells whether or not the keys in its map are in ascending order. Precondition: All keys can be cast to Comparable (i.e., have the usual `compareTo` method).
Exercise 16.40* Write a NodeMap method `public int countLess (Comparable id)`: The executor tells how many of its entries have a key that is less than the given key. Precondition: All keys can be cast to Comparable.
Exercise 16.41* Write a NodeMap method `public Comparable firstKey()`: The executor returns the smallest key in its map. Precondition: All keys can be cast to Comparable. Return null if the map is empty.
Exercise 16.42* Write a NodeMap constructor with a NodeMap parameter: Construct the new NodeMap to have the same key/value pairs as the parameter.
Exercise 16.43* Write a NodeMap method `public void display()`: The executor prints all of its keys to `System.out`. This is a useful method when you are debugging a program using NodeMaps. Use recursion.
Exercise 16.44** Write a NodeMap `public void removeValue (Object ob)`: the executor removes every MapEntry object that has `ob` as its value.
Exercise 16.45** Write a NodeMap method `public NodeMap reverse()`: the executor returns a new NodeMap with the same entries in it but in the opposite order.

Part B Enrichment And Reinforcement

16.6 Implementing Mapping Iterators; Ordered Tables

In many situations, you need to go through the entries stored in a Mapping object one at a time. An Iterator object is the appropriate way to do this. The Iterator interface is part of the standard library in the `java.util` package. As Listing 16.4 showed, it requires three methods to be in any class that would implement it, though `remove` can be an unsupported operation (i.e., can always throw an Exception):

```
public interface Iterator
{  public Object next();     // return next available element
   public boolean hasNext(); // tell whether next() is allowed
   public void remove();   // remove what next() last returned
}
```

Listing 16.10 contains an implementation of the Iterator interface that can be placed inside the ArrayMap class (`remove` is deactivated). That lets it access the instance variables of an ArrayMap object. The `itsPos` instance variable gives the position of the next value that the Iterator returns. The `next` method returns the value stored at that position after advancing `itsPos` by 1. The `java.lang` package contains the UnsupportedOperationException class.

Listing 16.10 The MapIt class nested in the ArrayMap class

```
private static class MapIt implements java.util.Iterator
{
   private int itsPos = 0;
   private ArrayMap itsMap;


   public MapIt (ArrayMap given)  // given will not be null
   {  itsMap = given;
   }  //=====================

   /** Return true if the Iterator can produce another one.*/

   public boolean hasNext()
   {  return itsPos < itsMap.itsSize;
   }  //=====================

   /** Return the next element in the sequence, if any. */

   public Object next()
   {  if ( ! hasNext())
         throw new java.util.NoSuchElementException
                              ("iterator has no next element!");
      itsPos++;
      return itsMap.itsItem [itsPos - 1];
   }  //=====================

   /** Remove the element last returned by a call of next().*/

   public void remove()
   {  throw new UnsupportedOperationException ("no remove!");
   }  //=====================
}
```

The MapIt classes

The **MapIt** class is declared with the heading `private static class MapIt` inside the ArrayMap class. This kind of declaration is called a **nested class**. Now the iterator method inside the ArrayMap class is easy to code:

```
public java.util.Iterator iterator()
{  return new MapIt (this);
}  //======================
```

The **MapIt** class nested in the NodeMap class has the same methods as shown in Listing 16.10; only the coding is different. The implementation in Listing 16.11 has the instance variable `itsPos` be the NodeMap object containing the next value that the Iterator will return. So `itsPos` is initially the first node on the linked list in the NodeMap. Then the `hasNext` method simply tests whether `itsPos` contains data, and the `next` method returns the data in that node after moving forward to the next node.

Listing 16.11 The MapIt class nested in the NodeMap class

```
public java.util.Iterator iterator()  // member of NodeMap
{  return new MapIt (this);
}  //=====================

private static class MapIt implements java.util.Iterator
{
   private NodeMap itsPos;


   public MapIt (NodeMap given)  // given will not be null
   {  itsPos = given;
   }  //=====================

   public boolean hasNext()
   {  return itsPos.itsNext != null;
   }  //=====================

   public Object next()
   {  if ( ! hasNext())
         throw new java.util.NoSuchElementException
                              ("iterator has no next element!");
      Object valueToReturn = itsPos.itsData;
      itsPos = itsPos.itsNext;
      return valueToReturn;
   }  //=====================

   public void remove()
   {  throw new UnsupportedOperationException ("no remove!");
   }  //=====================
}
```

Ordered tables

Sometimes it is essential to be able to iterate through the data values in ascending order of IDs, where the IDs are Comparable values. This implies you should store them in ascending order, for efficiency. Doing so for the NodeMap class would improve execution of the search methods (`get` and `containsKey`) when searching for an ID that is in the **table** (another name for a map), as well as `put` and `remove`.

One of the fastest-executing ways of maintaining an ordered table is the following, which we will implement as the **BinarySearchListMap** class. It is structured as follows:

- All of the MapEntry values are kept in ascending order in a linked list of nodes that has a trailer node (although having a header node instead would work about as well).
- An array named `itsArray` contains references to approximately every fourth node (although any number from 3 to 8 would work about as well) starting with the first node and ending with the trailer node.
- A non-negative int `itsSize` is the number of MapEntry values in the data structure.
- To find an ID, perform binary search on the array until you find the Node `p` such that the ID you want is or should be in the four or so Nodes beginning at `p`. Then perform a sequential search on the linked list starting at `p`.

As data values are inserted and deleted, the portions of the linked list between two consecutive array positions will vary from four. But execution time to find or insert or remove a MapEntry remains big-oh of log(N) where N is the number of data values, as long as hardly any of those portions are more than seven or eight nodes in length. The reason is that binary search of up to eight values takes about as many comparisons of IDs as the average number of comparisons required for sequential search.

Listing 16.12 (see next page) contains part of the coding for this class. The `put` method is quite similar to the NodeMap `put` method in the earlier Listing 16.9. In fact, lines 7-13 are exactly the same as lines 5-11 in Listing 16.9 except that here we insert at position `loc` and there we insert at position `this` (the beginning of the list).

The `put` method calls on two private methods: The `lastLessEqual` method performs a binary search through the array to find the node of highest index containing an ID less than or equal to the given ID. If all IDs whose nodes are in the array are larger than the given ID, `lastLessEqual` returns index zero. Then `firstGreaterEqual` does a sequential search starting from the node that was returned, until it finds a node `loc` containing an ID equal to or greater than the given ID (but stopping at the trailer node). The new MapEntry goes right before the MapEntry in `loc`, except it may have the same ID, in which case it replaces the MapEntry in `loc`.

The constructor's parameter is an object that iterates through some Mapping or Collection of MapEntries in ascending order. In practice, you might store the information from an ordered Mapping in a file at the end of a business week, then read the file in again the next day to build a new ordered table as a BinarySearchListMap object. This class is best used for situations where the data structure does not grow or shrink drastically during use; more than a fifty percent change in one business week can reduce the efficiency of the data structure. But an exercise suggests how to rewrite `put` to allow for a great increase in size.

This constructor simply creates the linked list with trailer node from the values the iterator produces, meanwhile counting them. Then it calls a private method to store references to the first, fifth, ninth, thirteenth, etc. nodes in the array, starting at `itsArray[0]`. The rest of this BinarySearchListMap class is left as a major programming project. A related project uses the same idea for a very fast implementation of set operations.

All of the operations in BinarySearchListMap execute in big-oh of log(N) time. You should research the concept of "skip lists" if you want to see another popular way of maintaining an ordered table with expected big-oh of log(N) time for all operations.

Exercise 16.46 (harder) Revise the MapIt class for NodeMap to have the `remove` method work right. It should throw an Exception if it cannot remove the MapEntry most recently returned by a call of `next()`. Add another instance variable `itsPrevious` which is the node containing the MapEntry that can be removed, when one exists.

Listing 16.12 The BinarySearchList class

```
public class BinarySearchListMap implements Mapping
{
   private Node[] itsArray;
   private int itsSize = 0;

   public Object put (Object id, Object value)
   {  if (id == null)                                                  //1
         return null;  // Mappings ignore a null key                  //2
      Node start = itsArray[lastLessEqual ((Comparable) id)];//3
      Node loc = firstGreaterEqual ((Comparable) id, start); //4
      if (loc.itsNext == null                                          //5
                     || ! loc.itsData.getKey().equals (id))  //6
      {  itsSize++;                                                    //7
         loc.itsNext = new Node (loc.itsData, loc.itsNext);  //8
         loc.itsData = new MapEntry (id, value);                       //9
         return null;                                                  //10
      }                                                                //11
      Object valueToReturn = loc.itsData.getValue();                   //12
      loc.itsData = new MapEntry (id, value);                          //13
      return valueToReturn;                                            //14
   }  //====================

   /** Create a Mapping that contains the data values produced
    *  by the iterator.  Precondition:  The iterator produces
    *  MapEntry objects in ascending order of keys. */

   public BinarySearchListMap (java.util.Iterator it)
   {  Node header = new Node (null, null), p;                          //15
      for (p = header;  it.hasNext();  p = p.itsNext)                  //16
      {  p.itsNext = new Node ((MapEntry) it.next(), null);  //17
         itsSize++;                                                    //18
      }                                                                //19
      p.itsNext = new Node (null, null);  // trailer node             //20
      createArrayContainingEveryFourthOne (header.itsNext);  //21
   }  //====================

   private static class Node
   {  public MapEntry itsData;
      public Node itsNext;

      public Node (MapEntry data, Node next)
      {  itsData = data;                                               //22
         itsNext = next;                                               //23
      }  //====================
   }
}
```

Exercise 16.47 (harder) Write a BinarySearchListMap method `public void putAll (Mapping given)`: The executor adds to itself all the MapEntries in `given`, replacing the value where duplicate keys occur. Precondition: All keys are mutually Comparable.
Exercise 16.48* Write another constructor for Listing 16.12 to make an empty Mapping.
Exercise 16.49* Rewrite the `put` method in Listing 16.12 to discard the current array and create a new one when you reach seven times as many MapEntries as components (this requires roughly 1 second per 10 million data values with a 1400MHz PC chip).
Exercise 16.50* Rewrite the constructor in Listing 16.12 to not create the extra Node.
Exercise 16.51* Write a BinarySearchListMap method `public void clear()`: The executor removes all of its MapEntries, leaving itself empty.

16.7 Hash Tables

Say you need to store around 100,000 Info values in a Mapping. Then `lookUp()` needs an average of up to 50,000 comparisons to search for a value in the simplest implementations, and even binary search through the data will require at least 17 comparisons for the average look-up. Wouldn't it be great if you had an implementation of the Mapping interface that could `lookUp` or `remove` a given data value with only one comparison?

One way to do this is to use the ID of each data value as the index of an array where data values are stored. For instance, if a company has employee IDs that range in value from 1 to 99,999, you could have an array of size 100,000 and store e.g. the data value with ID 7413 in `itsItem[7413]` (you store null there if that ID is not in use). Or if each data value corresponds to a particular day in the year, you could have an array of size 366. Listing 16.13 shows a start on this kind of data structure.

Listing 16.13 Two methods in the IndexedMap class of objects

```
public class IndexedMap implements Mapping
{
   private MapEntry[] itsItem = new MapEntry [100000]; //all null


   /** All methods throw a RuntimeException if the given id is
    *  not an Integer object with intValue() in 0...999999. */

   public boolean containsKey (Object id)
   {  int k = ((Integer) id).intValue();
      return itsItem[k] != null;
   }  //=====================


   public Object get (Object id)
   {  int k = ((Integer) id).intValue();
      return itsItem[k] != null ? itsItem[k].getValue() : null;
   }  //=====================
}
```

The collision problem

The kind of situation just described is quite rare, so we will not carry it further in this book. You would have to know that no ID is larger than `itsItem.length-1`. Now you might be thinking that Social Security Numbers (abbrev. SSN) would usually do, since they never exceed 1 billion (SSNs have nine digits). But they cause a different problem: You would need an array of size 1 billion. If you are using four bytes for each reference to a MapEntry, that is a total of four gigabytes of RAM, which may not be available. And even if it were available, it would be a horrendous waste to use that size of array to store just a few hundreds of thousands of values.

One approach is to use the last 5 digits of the SSN for the index into an array with 100,000 components. This is called **open addressing**. But some of the tens of thousands of people will surely have the same last 5 digits. Even if we use the last 6 digits (thus requiring an array of size 1 million), probability theory tells us we are virtually guaranteed to have quite a few cases of people with the same last 6 digits.

This is the **collision problem**: Several data values "collide" when they all try to get into the same storage space. One collision-resolution solution is **linear probing**: When you have a data value X that goes in a position at index K, and some data value is already there, put it at K+1; but if there is something there, put it at K+2 instead, or at K+3 if necessary, etc. If you go beyond the end of the array, start searching for a space at index 0, then 1, etc. Of course, this causes all sorts of problems when you try to find the data value later, since it is not in the right spot. And deletions cause even more problems.

The chained hash table

The **chained hash table** solution in Listing 16.14 (see next page) is a **ChainedHashMap** with an array of 100,000 linked lists. Each list contains all the data values whose SSN has the last 5 digits equal to the index (e.g., 347-82-1347 will be stored in the linked list at index 21,347). This approach executes faster than open addressing.

If the IDs are strings of characters (such as names), you calculate a number in the range from 0 to 99,999 from the Unicode values of the letters of the ID and store the data value in the linked list at that array index. Java provides a useful function for this purpose:

- The Object class has a method **`hashCode`**`()` that returns an int value. If `x.equals(y)` then `x.hashCode()` and `y.hashCode()` are the same int value.
- The Integer, Double, and Long classes override this `hashCode` method with an appropriate calculation from the bits in the corresponding primitive value.
- The String class overrides this `hashCode` method with a calculation from the Unicode values of the letters of the characters. Any class you use for IDs in a hash table should define a `hashCode` method for which `x.equals(y)` means that `x` and `y` have the same hash code value. If equal objects have the same `toString` value, you could use `x.toString().hashCode()` for your hash codes.

The calculation that Java's String class provides is as follows. For the chained hash table, divide this `hashCode` value by 100,000 and use the remainder as the index into the array of 100,000 values:

```
public int hashCode()
{  int result = 0;
   for (int k = 0;  k < length();  k++)
      result = result * 31 + charAt (k);
   return result;
}  //======================
```

You choose the array size of 100,000 when you think that the data set will normally contain between 50,000 and 100,000 data values. In general, choose an array size only somewhat more than the maximum you expect to store in the Mapping structure. For instance, you might use an array of size 10,000 for a company with five to eight thousand employees.

The average linked list will have less than one data value. Because there will tend to be many cases where several people have the same last five digits of their SSNs, or have the same `hashCode` value for their names, many components will be empty and many will have more than one data value. However, it should be quite rare to have more than 3 or 4 data values stored on the same linked list. Thus the look-up process should take on average less than two comparisons, which is much better than the seventeen or more that binary search would require.

Warning Do not use a hash table unless you know that the keys being inserted have overridden the `hashCode()` method that the Object class provides. That basic `hashCode()` method is calculated from the RAM address of the object, not from the values stored in the object.

Listing 16.14 The ChainedHashMap class of objects, some methods left for exercises

```
public class ChainedHashMap implements Mapping
{
   private Node[] itsItem = new Node[100000];    // all null
   private int itsNumValues = 0;


   /** Tell whether a key/value pair satisfies key.equals(id). */

   public boolean containsKey (Object id)
   {  if (id == null)
         return false;
      int index = Math.abs (id.hashCode()) % itsItem.length;
      return lookUp (id, itsItem[index]) != null;
   }  //======================

   /** Return the value in the key/value pair for which
    *  key.equals(id).  Return null if no such key/value pair. */

   public Object get (Object id)
   {  if (id == null)
         return null;
      int index = Math.abs (id.hashCode()) % itsItem.length;
      Node p = lookUp (id, itsItem[index]);
      return (p == null)  ?  null  :  p.itsData.getValue();
   }  //======================

   /** Return the Node containing the key/value pair for which
    *  key.equals(id).  Return null if no such key/value pair. */

   private static Node lookUp (Object id, Node list)
   {  return (list == null)  ?  null
             : list.itsData.getKey().equals (id)  ?  list
             : lookUp (id, list.itsNext);
   }  //======================

   private static class Node
   {
      public MapEntry itsData;
      public Node itsNext;

      public Node (MapEntry data, Node next)
      {  itsData = data;
         itsNext = next;
      }  //======================
   }
}
```

You will usually have fewer collisions if you use a prime number for the size of a hash table. Most sets of data have patterns where some sets of integers occur much less frequently than others as hash codes. For instance, all of them might be even numbers, which leaves half the table empty if its size is 10,000. If you choose a prime number such as 10,007 or 10,009 for the array length, you do not have this problem.

RAM usage

The space requirements for various Mapping implementations can be calculated with some effort. A **word** of storage is 4 bytes, sufficient space for a reference to an object or for an int value. If N is the number of data values stored, then an ArrayMap (Listing 16.6)

takes between N and 1.5*N words of storage for `itsItem` (since the array grows by 50% each time it fills up). A NodeMap (Listing 16.8) takes 2*N words of storage for the N Nodes in use. A ChainedHashMap (Listing 16.14) or a java.util.HashMap (described in Section 16.10) takes about 1.5*N words of storage for `itsItem` and 2*N words of storage for the N Nodes in use, a total of 3.5*N words. A java.util.TreeMap (described in Section 16.10) takes about 4*N words (the tree Nodes store 3 values instead of 2, plus the red-black information it requires).

However, this count is misleading in that each object carries around an extra word of storage containing a reference to its own class. So you have to add N more words for each of the implementations just mentioned other than the ArrayMap.

This is where open addressing has a great advantage over chaining for maps that are based on hash tables: Open addressing uses less than half the space. You can strike a nice balance if you store a partially-filled array of entries in each component of `itsItem`:

- To add an entry where none was before, create an array of length 1 for it.
- To add an entry when you already have an array holding some entries for that same index, put it at the next available place in the array, unless the array is full, in which case you transfer the information to another array twice as large.
- To delete an entry, replace it by the entry at the end of its array (no shifting around).

This way, you use less than three-quarters of the space of chaining and you avoid probing, but at the cost of some execution time. The development of this **ArrayHashMap** implementation is left as a major programming project.

More on open addressing

Even if the collision problem is not too severe for open addressing, linear probing causes **clustering**: Putting an ID elsewhere fills up a space that many other IDs would otherwise go into. This can mean that, for data such that chaining might not have more than three data values for any one index, you could still have to search through dozens of entries to find what you are looking for with open addressing.

This problem can be avoided by using a probe interval that depends on the key you are putting in. This solution is called **double-hashing**. For instance, you could probe forward by jumps of 1 to 13, where the size of the jump is calculated from the hash code of the ID you are looking for. You could start with the coding in Listing 16.15 for an implementation of Mapping named **OpenHashMap** that uses this technique.

Listing 16.15 The containsKey method in the OpenHashMap class of objects

```
public class OpenHashMap implements Mapping
{
   private MapEntry[] itsItem = new Node[100000];    // all null
   private int itsNumValues = 0;


   public boolean containsKey (Object id)
   {  if (id == null)
         return false;
      int index = Math.abs (id.hashCode()) % itsItem.length;
      int jump  = Math.abs (id.hashCode()) % 13 + 1;
      while (itsItem[index] != null
                     && ! itsItem[index].getKey().equals (id))
         index = (index + jump) % itsItem.length;
      return itsItem[index] != null;
   }  //=====================
}
```

This `containsKey` coding calculates the index where the `id` should go. If that component is empty or it contains the given `id`, the search ends. Otherwise it looks 1 to 13 components further down the array (unless that would put it past the end of the array, in which case it "wraps around" to the beginning of the array). It keeps making jumps of the same size until it finds what it is looking for.

That still leaves a serious problem for open addressing that does not exist for chaining: What if you delete an entry? Say you wanted to put X at index 5000 and there was already a value there, so you tried index 5009 and found a value already there. Then you tried index 5018 and that was available. So you put X at index 5018. Later you remove the entry that is at index 5009. Later still you look for X, starting at 5000 of course. You see something there, so you look at index 5009. You see nothing there, so you think that X is not in the data set. But it is.

One standard solution for this problem is to simply forbid removal of a data value once you have put it in the data structure. That means that you have the `remove` method throw an UnsupportedOperationException. But that is a cop-out.

A better way is to create a special Object different from all others. Call it NOTHING. For instance, you could define `NOTHING = new Integer(0)`. Now each time you remove a data value, you replace it by NOTHING instead of by null. Each time you search for a data value, you treat NOTHING as a non-match rather than as indicating that your search should stop. And each time you put a new value in, you replace NOTHING if you find it while you are searching.

Now the last four lines of the `containsKey` method for the OpenHashMap class are replaced by the following coding:

```
   while (itsItem[index] != null && itsItem[index] != NOTHING
                  && ! itsItem[index].getKey().equals (id))
      index = (index + code % 13 + 1) % itsItem.length;
   return itsItem[index] != null && itsItem[index] != NOTHING;
```

The rest of the development of OpenHashMap is left as a major programming project.

Exercise 16.52 Write the `size` and `isEmpty` methods for the ChainedHashMap class.
Exercise 16.53 (harder) Write the `put` method for the ChainedHashMap class.
Exercise 16.54* Write a ChainedHashMap `public void clear()`: The executor removes all of its MapEntries, leaving it empty.
Exercise 16.55* Write a ChainedHashMap `public boolean containsValue (Object ob)`: The executor tells whether it contains a MapEntry with the given value.
Exercise 16.56** Write a ChainedHashMap `public Object firstKey()`: The executor returns the key that is less than all other keys in the Mapping. Throw an Exception if the executor is empty. Assume `compareTo` does not return zero.
Exercise 16.57** Write a ChainedHashMap `public Mapping subMap (Object firstKey, Object lastKey)`: The executor returns a new Mapping with all the MapEntries whose key is greater-equal `firstKey` and less than `lastKey`. Throw an Exception if the usual exceptional situations arise.
Exercise 16.58** Write the `remove` method for the ChainedHashMap class.
Exercise 16.59** Write an Iterator nested class for the ChainedHashMap class, leaving `remove` unsupported. Have three instance variables: `itsPos` is the Node containing the data value that `next()` will return, `itsIndex` is the index of that Node's linked list in the array, and `itsItem` is the array itself.
Exercise 16.60** Revise the entire ChainedHashMap class to allow an ID to be null: Do not put it in the array, but instead add an instance variable `itsNullValue` to record the value that corresponds to a null key.

16.8 Further Development Of Scheme's List Class

The earlier Scheme analysis found most of the public methods that a List needs (Section 16.3), so you could now decide what a List object needs to know to perform those methods (its instance variables). It need not have an array. It is enough that a List object store just two Items: the first element on the List plus the list of all the rest of the elements. The latter gives you access to the second, third, or whatever elements. These two instance variables correspond to the `first` and `rest` methods in the List class.

You could name the List instance variables `itsFirst` and `itsRest`. The natural List constructor therefore has two parameters that supply the values for these two instance variables. Since there is no reason to change these values once they are assigned, you might as well make them final instance variables. Final instance variables do not have to be initialized in their declarations, as long as they are initialized in every constructor.

An empty list has no first element, so `itsFirst` is null. The value of `itsRest` for the empty List should also be null. There is no point in wasting space by having many empty lists, so you can have just one class variable `EMPTY` that is the only empty List in use.

The List class also needs a class variable `theDefs` to hold the definition of each function and a class variable `theSpecials` to hold the list of special keywords and predefined function names. Both of these are SchemeMaps, an implementation of the Mapping interface. `theDefs` starts off as an empty SchemeMap, but `theSpecials` has to have the various keywords and predefined function names put in it.

This all leads to the List class in Listing 16.16 (see next page), with the obvious methods filled in. Only three of the special WordUnits are listed in the `addThemAll` method; more can be added later. This coding leaves only two List methods to work on: `getValue` and `toString`.

The toString method in List

The `toString` method should print the List value the way the user would have typed it in. It is normal in Scheme to use the WordUnit `empty` to stand for the empty list, rather than two parentheses with nothing inside them. The structured design could therefore be as shown in the accompanying design block. The coding is in the lower part of Listing 16.16; note that `p` must be declared outside the for-statement.

STRUCTURED NATURAL LANGUAGE DESIGN of List's toString method

1. If the list is empty, then...
   1a. Return the String value "empty".
2. Start with a left parenthesis stored in some String variable; call it `result`.
3. For each element of the list except the last one, do...
   3a. Add its String form to `result`, followed by a blank to separate it from the next one.
4. Add the String form of the last element of the list to `result`, followed by a right parenthesis.
5. Return `result` as the answer.

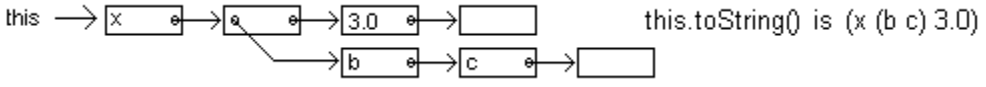


Figure 16.9 Effect of toString()

Listing 16.16 The List class except for the getValue method

```
public class List implements Item
{
   public static final List EMPTY = new List (null, null);
   private static SchemeMap theDefs = new SchemeMap();
   private static SchemeMap theSpecials = addThemAll();
   ////////////////////////////////
   private final Item itsFirst;
   private final List itsRest;


   public List (Item first, List rest)
   {  itsFirst = first;
      itsRest  = rest;
   }  //======================

   private static SchemeMap addThemAll()
   {  SchemeMap result = new SchemeMap();
      result.put (new WordUnit ("set!"), new Real (1));
      result.put (new WordUnit ("define"), new Real (2));
      result.put (new WordUnit ("+"), new Real (3));
      // many more will be added later
      return result;
   }  //======================


   public Item first()
   {  return itsFirst;
   }  //======================

   public List rest()
   {  return itsRest;
   }  //======================

   public boolean isEmpty()
   {  return itsRest == null;
   }  //======================


   public String toString()
   {  if (this.isEmpty())
         return "empty";
      String result = "(";
      List p;
      for (p = this;  ! p.itsRest.isEmpty();  p = p.itsRest)
         result += p.itsFirst.toString() + " ";
      return result + p.itsFirst.toString() + ")";
   }  //======================
}
```

Any function name defined by the user must be stored in `theDefs` along with its definition. The function name is the key in the Map object; the value stored with it is a two-element List object, one element for the heading and the other for the body. For instance, `(define (cube p) (* p (* p p)))` would be stored in `theDefs` with the key `cube` and the value `((cube p) (* p (* p p)))`. So when you get the value `def` from `theDefs` corresponding to `cube`, `def.itsFirst` is the heading `(cube p)` and `def.itsRest.itsFirst` is the body `(* p (* p p))`.

The getValue method

An empty list (normally used as a parameter of a function call) has the value EMPTY. A non-empty list might begin with a keyword or other previously-defined WordUnit, so `getValue` looks up that WordUnit in `theSpecials` Mapping. If it is there, the corresponding value will be a Real (a number) indexing the code segment that tells how it is to be evaluated. Otherwise `getValue` calculates the normal value.

The normalValue method

A reasonable design for the `normalValue` method is in the accompanying design block. Note the great advantage of throwing BadInputs when necessary. The `getValue` method might throw an Exception, or it might call the `normalValue` method which throws an Exception. Instead of the `getValue` method having to handle both of those situations with an if-statement, the `normalValue` method terminates without returning a value; the `getValue` method terminates without returning a value; and the main logic catches the throw and handles it.

STRUCTURED NATURAL LANGUAGE DESIGN of List's normalValue method

1. Retrieve the definition of `this.itsFirst` from `theDefs`.
2. If the definition is not found for this function call then...
   2a. Throw an Exception.
3. Find the value of each of the actual parameters, storing those values in a new List of arguments. Call it `actuals`.
4. Assign the arguments of `actuals` to the corresponding formal parameters. Throw an Exception if the two lists of parameters are not the same length.
5. Evaluate the body of the function using the newly-assigned values of the formal parameters.
6. Return the value obtained.

Step 5 of the `normalValue` design calls the `getValue` method in the class of the Item listed as the function body. The function body might be `(avg 3 2)`, which is a List object. So it calls List's `getValue` method, which calls the `normalValue` method. So `normalValue` can conceivably call a method that calls `normalValue`. Recursion will be needed here several times, because a List is a highly recursive structure: A List is a sequence of elements some of which may be Lists.

Steps 3 and 4 of the design cannot be done in just one or two Java statements, so it makes sense to have them be method calls that are developed next. The upper part of Listing 16.17 (see next page) gives the completed logic for the `normalValue` method. As an example, the `normalValue` method applied to `(avg 6 b)` when `b` is a variable with the value 13 produces the following:

- `this.itsFirst` is `avg`, the name of the function being called.
- `this.itsRest` is `(6 b)`, the list of arguments of the function call.
- `this.itsRest.allValues()` produces `(6 13)`, which is assigned to `actuals`.
- `def.itsFirst` is `(avg x y)`, the heading of the function.
- `def.itsFirst.itsRest` is `(x y)`, the list of formal parameters in the heading.
- The call of `assignValues` stores 6 in `x` and 13 in `y`.
- The call of `getValue` applies to `def.itsRest.itsFirst`, i.e. `(/ (+ x y) 2)`, which evaluates as `(/ (+ 6 13) 2)` and thus as 9.5.

Listing 16.17 The normalValue method in the List class

```
// public class List implements Item, continued

   public Item getValue()
   {  if (this.isEmpty())                                        //1
         return EMPTY;                                           //2
      Real index = (Real) theSpecials.get (this.first());        //3
      return index != null ? this.rest().specialValue (index)//4
                           : this.normalValue();                 //5
   }  //====================

   /** Find the definition of the function, evaluate the
    *  function call, and return the calculated value. */

   private Item normalValue()
   {  List def = (List) theDefs.get (this.itsFirst);             //6
      if (def == null)                                           //7
         throw new BadInput (this.itsFirst.toString()            //8
                             + " is not a function name");       //9
      List actuals = this.itsRest.allValues();                   //10
      ((List) def.itsFirst).itsRest.assignValues (actuals);      //11
      return def.itsRest.itsFirst.getValue();                    //12
   }  //====================

   private List allValues()
   {  return this.isEmpty()  ?  EMPTY  :  new List            //13
           (this.itsFirst.getValue(), this.itsRest.allValues());
   }  //====================

   private void assignValues (List actuals)
      // Protecting against different lengths is an exercise
   {  if ( ! this.isEmpty())                                     //15
      {  WordUnit parameter = (WordUnit) this.itsFirst;          //16
         parameter.setVariableValue (actuals.itsFirst);          //17
         this.itsRest.assignValues (actuals.itsRest);            //18
      }                                                          //19
   }  //====================
```

Private methods used by the normalValue method

The `normalValue` method calls `allValues()` to get a List of the values of all the elements in `this.itsRest`, which is the list of arguments. The logic of `allValues` is: There are two kinds of Lists, empty Lists and nonempty Lists. The list of values for an empty List `this` is the empty List itself. The list of values for a nonempty List `this` is the List consisting of the value of the first element of `this` followed by the List of the values of all the elements in the rest of `this`. And that is recursion! The logic is in the middle part of Listing 16.17, with the recursive call in boldface.

The `normalValue` method then calls `assignValues(actuals)`. This method call is to assign each formal parameter in its executor List the corresponding value in the `actuals` List. The logic of `assignValues` is: There are two kinds of Lists, empty Lists and nonempty Lists. An empty List need do nothing. A nonempty List `this` needs to assign `actuals.itsFirst` to its first element and then have the rest of the List be the executor of the method call `assignValues(actuals.itsRest)`. And that is recursion! The logic is in the lower part of Listing 16.17, with the recursive call in boldface. Figure 16.10 shows what happens during a call of `assignValues`.

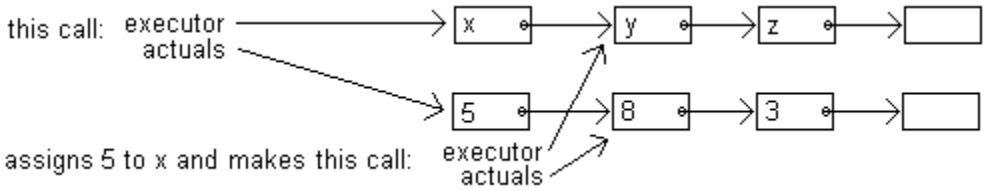


Figure 16.10 Example of a call of the assignValues(List) method

The logic developed here only gives the right results if every parameter of every Scheme function the user defines has a name different from any globally defined variable and from any other parameter. This problem can be corrected. A simple solution is to have the interpreter store the name of each parameter in a distinctive way, e.g. `p` in the `cube` function would be stored as `cube.p`. That solves all problems until the user writes a recursive Scheme function. We will not get into that in this book, however.

Exercise 16.61* Rewrite the List method `public String toString()` to use a StringBuffer object (described in Section 7.12) for efficiency.
Exercise 16.62* Revise the `assignValues` method in Listing 16.17 to throw a BadInput Exception if either List is longer than the other.
Exercise 16.63* Rewrite the `allValues` method without using recursion.
Exercise 16.64* Rewrite the `assignValues` method without using recursion.

*16.9 About Map, AbstractMap, HashMap, And TreeMap (*Sun Library)*

The **java.util.Map** interface prescribes twelve methods (in addition to those that any object has: `equals` and `hashCode`) -- the six described in Listing 16.4 (`put`, `remove`, `get`, `containsKey`, `isEmpty`, and `size`) plus another six:

- `someMap.clear()` removes all entries from the data structure.
- `someMap.containsValue(valueObject)` tells whether some entry contains `valueObject` as a value.
- `someMap.putAll(aMap)` in effect repeatedly calls `put` for each entry in `aMap`.
- `someMap.entrySet()` returns a Set object that contains all of the executor's entries. Sets are described in Chapter Fourteen; they are Collections that do not allow duplicate values (since any two entries have to have different keys).
- `someMap.keySet()` returns a Set object that contains all of the executor's keys.
- `someMap.values()` returns a Collection object containing all of the executor's values.

Note that Map does not have a method for directly obtaining an Iterator over the MapEntries. However, you can obtain an Iterator indirectly with the following statement:

```
Iterator it = someMap.entrySet().iterator();
```

The **java.util.Map.Entry** interface is a sub-interface of Map. It has the `getKey()` and `getValue()` methods described in Listing 16.5, as well as a `setValue()` method so you can change the value associated with a given ID.

The **java.util.AbstractMap** class makes it easy to implement the Map interface. It has only one abstract method, the `entrySet` method. If you have a class that extends AbstractMap and replaces the `entrySet` method, it will be a full implementation of Map, though it will not allow modifications of the Entries in the Map. If you also replace the `put` method (which otherwise throws an UnsupportedOperationException) in AbstractMap and/or the `remove` method, the Map will be modifiable.

TreeMap objects (from java.util)

The **TreeMap** class is a full implementation of Map that uses a binary search tree (to be discussed in detail in the next chapter). Keys must be Comparable. In addition, it colors each node either red or black according to a set of rules that guarantees that the tree cannot become too out of balance. Specifically, the worst-case time for `get`, `put`, and `containsKey` can never be more than twice as much as log2(N), where N is the number of entries in the tree. The class includes the following constructors:

- `new TreeMap()` creates an empty data structure.
- `new TreeMap(aMap)` creates a data structure containing each entry in `aMap`.

Actually, TreeMap implements **SortedMap**, an extension of the Map interface that prescribes that the iterator will produce Entry objects in ascending order of keys. The additional methods include `firstKey()` and `lastKey()` as well as `subMap (fromKey, toKey)` that returns a view of the MapEntries whose keys are at least `fromKey` but less than `toKey`.

HashMap objects (from java.util)

The **HashMap** class is another full implementation of Map provided in the Sun library. The `get`, `put`, and `containsKey` operations have an execution time that is big-oh of 1, i.e., in constant time independent of the number of values in the Map. The HashMap class allows a null key and null values. It includes the following constructors:

- `new HashMap()` creates an empty data structure.
- `new HashMap(aMap)` creates a data structure containing each entry in `aMap`.

The HashMap class uses a hash table and each Object's `hashCode` method to get this speed, similar to ChainedHashMap. This implementation of Map makes no guarantee as to the order of the entries. The order might even change substantially over time.

16.10 Review Of Chapter Sixteen

About the Sun standard library java.util.Map interface:

- `someMap.containsKey(Object id)` tells whether id/someValue is in the Map.
- `someMap.get(Object id)` returns `someValue` when id/someValue is in the Map, and otherwise returns null.
- `someMap.put(Object id, Object someValue)` adds that id/someValue pair to the Map, replacing any id/otherValue pair with the same id.
- `someMap.remove(Object id)` removes the id/someValue pair from the Map, returning `someValue`; but it returns null if the id/someValue pair was not in the Map.
- `someMap.isEmpty()` tells whether the Map object contains no id/someValue pairs at all.
- `someMap.size()` tells the number of id/someValue pairs in the Map.
- The Map interface has several more methods, for which the AbstractMap class gives generic implementations as long as you provide coding for the `entrySet` method.

About the Sun standard library Iterator class:

- `someIterator.hasNext()` tells whether a collection of objects has an object that `next` will retrieve.
- `someIterator.next()` advances to the next element and returns that Object, if there is one. It throws `java.util.NoSuchElementException` if there is none.
- `someIterator.remove()` takes out of the collection the element just returned by a call of `next`. It throws `java.lang.IllegalStateException` if that cannot be done. An Iterator implementation that does not allow removal should have `remove` throw a `java.lang.UnsupportedOperationException`.

Other vocabulary to remember:

- An **interpreter** translates one line of source code (written by the programmer) to object code (executable by the chip) at a time. It executes that one line before going on to translate the next line.
- A **naturally recursive** data structure is one that contains a smaller such data structure as part of itself, except when it is empty.
- A linked list is made up of nodes, each of which contains a reference to one piece of data and a reference to another node (or to null if it is the last node in the list). A **simple linked list** has data in each node. If N is the number of data values in a list, then a **linked list with trailer node** has N+1 nodes, the last node having null in place of its data.

Answers to Selected Exercises

16.1
```
(define (cent-to-fahr x)  (+  32  (*  1.8  x)))    ==> cent-to-fahr is defined
(cent-to-fahr 20)                                   ==> 68
```
16.2
```
(define (hypoteneuse a b) (sqrt (+ (* a a) (* b b))))    ==> hypoteneuse is defined
(hypoteneuse 6 8)                                         ==> 10
```
16.4
```
public Real (double par)
{     itsValue = par;
}
public Item getValue()
{     return this;    // because a Real is a constant
}
```
16.5
```
public String toString()
{     return "" + itsValue;
}
public double getNumber()
{     return itsValue;
}
```
16.6
```
output.append (parser.parseInput (field.getText()).getValue().toString());
```
16.7
```
public static void inOrOut (String word, String definition)
{     if (theDictionary.containsKey (word))
            theDictionary.remove (word);
      else
            theDictionary.put (word, definition);
}
```
16.8
```
public boolean hasDefinition()
{     for (int k = 0;  k < itsSize;  k++)
      {     if (theDictionary.containsKey (itsItem[k]))
                  return true;
      }
      return false;
}
```
16.9
```
public boolean equals (Object ob)
{     return ob instanceof WordUnit && this.itsValue.equals (((WordUnit) ob).itsValue);
}
```
16.10 x.first() is a. x.rest() is (b c d e). x.rest().first() is b.

16.11
```
public boolean hasThree()
{     return ! isEmpty() && ! rest().isEmpty() && ! rest().rest().isEmpty();
}
```

```
16.12   public void putAll (Mapping given)
        {   Iterator it = given.iterator();
            while (it.hasNext())
            {   MapEntry entry = (MapEntry) it.next();
                this.put (entry.getKey(), entry.getValue());
            }
        }
16.19   public int size()
        {   return itsSize;
        }
16.20   public int countLess (Comparable id)
        {   int count = 0;
            for (int k = 0;  k < itsSize;  k++)
            {   if (((Comparable) itsItem[k].getKey()).compareTo (id) < 0)
                    count++;
            }
            return count;
        }
16.21   public Object remove (Object id)
        {   int loc = lookUp (id);
            if (loc < 0)
                return null;
            Object valueToReturn = itsItem[loc].getValue();
            itsItem[loc] = itsItem[itsSize - 1];
            itsSize--;
            return valueToReturn;
        }
16.22   public ArrayMap reverse()
        {   ArrayMap valueToReturn = new ArrayMap();
            valueToReturn.itsItem = new MapEntry [this.itsItem.length];
            valueToReturn.itsSize = this.itsSize;
            for (int k = 0;  k < this.itsSize;  k++)
                valueToReturn.itsItem[k] = this.itsItem[this.itsSize - 1 - k];
            return valueToReturn;
        }
16.23   public Object put (Object id, Object value)
        {   if (id == null)
                return null;
            int loc = lookUp (id);
            if (loc < 0)
            {   itsItem[itsSize] = new MapEntry (id, value);
                itsSize++;
                return null;
            }
            Object valueToReturn = itsItem[loc].getValue();
            itsItem[loc] = new MapEntry (id, value);
            return valueToReturn;
        }
16.31   public Object secondOne()
        {   return (itsData == null || itsNext.itsData == null)  ?  null  :  itsNext.itsData.getValue();
        }
16.32   public void swapTwo()
        {   if (this.itsData == null || itsNext.itsData == null)
                return;   // a void method forbids returning a value
            MapEntry saved = this.itsData;
            this.itsData = itsNext.itsData;
            itsNext.itsData = saved;
        }
16.33   public void clear()
        {   itsData = null;
            itsNext = null;   // the other nodes are garbage-collected.
        }
16.34   Insert the following directly after the only left brace inside the body of that put method:
        if ( ! isEmpty())
        {   itsNext = new NodeMap (new MapEntry (id, value), itsNext);
            return null;
        }
16.35   public boolean equals (NodeMap given)
        {   return this.isEmpty()  ?  given.isEmpty()  :  ! given.isEmpty()
                    && this.itsData.equals (given.itsData) && this.itsNext.equals (given.itsNext);
        }
```

16.36
```
private Object putRecursive (Comparable id, Object value)
{   if ( ! this.isEmpty() && id.compareTo (this.itsData.getKey()) > 0)
        return this.itsNext.putRecursive (id, value);
    if (this.isEmpty()  ||  id.compareTo (this.itsData.getKey()) < 0)
    {   this.itsNext = new NodeMap (this.itsData, this.itsNext);
        this.itsData = new MapEntry (id, value);
        return null;
    }
    MapEntry save = this.itsData;
    this.itsData = new MapEntry (id, value);
    return save.getValue();
}
```
16.37 Use the put method of the preceding exercise. Replace the body of lookUp by the following coding:
```
    return this.isEmpty() || ((Comparable) this.itsData.getKey()).compareTo (id) >= 0
            ?  this  :  this.itsNext.lookUp (id);
```
Replace the body of the containsKey method by the following:
```
    NodeMap loc = lookUp (id);
    return ! loc.isEmpty() && loc.itsData.getKey().equals (id);
```
Replace the loc.isEmpty() condition in the get and remove coding by the following condition:
```
    loc.isEmpty() || ! loc.itsData.getKey().equals (id)
```
16.38
```
public NodeMap intersection (NodeMap given)
{   if (this.isEmpty() || given.isEmpty())
        return new NodeMap (null, null);
    int comp = ((Comparable) this.itsData.getKey()).compareTo (given.itsData.getKey());
    return comp > 0  ?  this.intersection (given.itsNext)
                :  comp < 0  ?  this.itsNext.intersection (given)
                :  new NodeMap (this.itsData, this.itsNext.intersection (given.itsNext));
}
```
16.46 Declare in the MapIt class nested in NodeMap:
```
private Node itsPrevious;
public void remove()
{   if (itsPrevious == itsPos)   // to signal that you have already removed it or never called next()
        throw new IllegalStateException ("nothing to remove");
    itsPrevious.itsData = itsPos.itsData;
    itsPrevious.itsNext = itsPos.itsNext;
    itsPos = itsPrevious;   // a signal that no further remove is allowed
}
```
Add this statement as the third-to-last statement in next and as the last one in the constructor:
```
    itsPrevious = itsPos;
```
16.47
```
public void putAll (Mapping given)
{   java.util.Iterator it = given.iterator();
    while (it.hasNext())
    {   MapEntry toAdd = (MapEntry) it.next();
        this.put (toAdd.getKey(), toAdd.getValue());
    }
}
```
16.52
```
public int size()
{   return itsNumValues;
}
public boolean isEmpty()
{   return itsNumValues == 0;
}
```
16.53
```
public Object put (Object id, Object value)  // compare with put in Listing 16.9
{   if (id == null)
        return null;
    int index = Math.abs (id.hashCode()) % itsItem.length;
    Node loc = lookUp (id, itsItem[index]);
    if (loc == null)
    {   itsItem[index] = new Node (new MapEntry (id, value), itsItem[index]);
        itsNumValues++;
        return null;
    }
    Object valueToReturn = loc.itsData.getValue();
    loc.itsData = new MapEntry (id, value);
    return valueToReturn;
}
```

17 Binary Trees

Overview

This chapter introduces a standard data structure called a binary tree. You have seen that a node for a linked list contains data and also a reference to one other node (or to null, at the end of the sequence). A node for a binary tree contains data and also references to two other nodes (or to null).

- Sections 17.1-17.3 develop an application to keeping track of descendents of a particular person. This application uses a family tree to store the ancestor relationships and discusses various ways of traversing the family tree.
- Sections 17.4-17.6 implement the Mapping interface and iterators with a binary search tree, on the condition that data values be Comparable with each other.
- Sections 17.7-17.8 discuss how to keep binary search trees decently balanced -- red-black trees, AVL trees -- and prove useful properties of these trees.
- Section 17.9 discusses more advanced applications of binary tree nodes, e.g., 2-3-4 trees and B-trees.

17.1 Analysis And Design Of The Genealogy Software

Problem statement

A client doing a genealogical study needs a program to track all of the blood descendents of one individual. She will begin by entering that one individual's name. Thereafter, she will enter the name of each person only after that person's parent has previously been entered. She wants the program to be able to do two things at any point during and after this data entry process, upon request: (a) list all children of a person who has already been entered, one child per line, and (b) display a list of all the people currently in the dataset, one person per line, with those further from the ancestor-of-all listed earlier.

You are part of a software development group that takes on this assignment. You begin by performing an analysis of the requirements for the software, then continue by developing a design for the software. The design is in two parts: The overall main logic and a description of the objects needed by the main logic, along with the services offered by those objects. Services offered by a class of objects are usually expressed as the public methods that can be called for those objects.

Analysis of requirements

First you find out what details are missing from the problem statement, details that you will need to solve the problem. Then you get clarification from the client:

Question In what order should the children of a given person be listed?
Client's Answer "I don't care." So you wait until you develop the algorithm and use whatever is easiest.

Question How will the client be able to tell who is the child of whom in the full listing?
Client's Answer "Oops, I didn't think of that. What do you suggest?"
Our Response Have each person's name indented three spaces further than its parent, and in such a way that you can visually find the parent of any person by going down the list to the first person who is "out-dented" further to the left than that person. The client accepts the suggestion.

Question What if the user asks for information about a person who is not already in the dataset?
Client's Answer Have the program say, "I'm sorry, that person is not in the family" and carry on.

Question How does the user indicate which of the possible choices is wanted?
Client's Answer Enter a single letter: L = list, A = add, S = search, and E = exit.

Question What if two people have the same name?
Client's Answer Assume that the user will never do this and act accordingly.

Main logic design

This family-tree software could proceed as shown in the accompanying design block. It gives the overall logic of the software expressed in ordinary English (though you may use whatever natural language you wish). However, selection between alternative actions and repetitions of actions are indicated by indenting the subordinate actions.

STRUCTURED NATURAL LANGUAGE DESIGN of the main logic

1. Ask the user for the ancestor-of-all and store it in the dataset.
2. Give directions on the four options available: list all, add one, search for one, exit.
3. Repeat the following as long as the user does not choose the exit option...
    3a. If the user's choice is to add one person then...
        Get the parent and search the dataset for it. If there...
            Ask for the name of the child and add that person.
    3b. If the user's choice is to search then...
        Get the parent and, if there, list all of his or her children.
    3c. If the user's choice is to list everyone then...
        List all of the people of the dataset with proper indenting.

Object design

The master design makes it seem quite natural to have a Genealogy dataset object that allows the three capabilities corresponding to the three choices the user has. The IO class developed in Listing 10.2 offers simple input/output services we will use here: `IO.say` prints a message using `JOptionPane.showMessageDialog` and `IO.askLine` prompts for String input using `JOptionPane.showInputDialog`. This basic object design then leads to the coding in Listing 17.1 (see next page).

```
Genealogy (String lilith)
    // constructor to create a dataset of parent-child relations with lilith at the root
public void addChild (String parent)
    // if the given parent is in the dataset, ask for a child of that parent and then add it
public void listChildren (String parent)
    // list all children of the parent, unless the parent is not in the dataset
public void listEverybody()
    // List all values in the dataset, 1 per output line, with all children of any data
    // value X listed before X and in the order they were entered, and each indented
    // 3 spaces further than X.  Also, for each child of X, the first line below that child
    // that is indented less than the child is X itself.
```

Exercise 17.1* Discuss the drawbacks of allowing two people of the same name.
Exercise 17.2* How would the design change if the user were allowed an additional option, namely, to find the parent of a given person?

Listing 17.1 The main logic for the genealogy problem

```
class GenealogyApp
{
   /** Read in names of members of a family with their
    *  parent-child relationships.  Then answer questions about
    *  what people are the children of what other people.  */

   public static void main (String[] args)
   {  String s = IO.askLine ("Enter the ancestor of everyone:");
      Genealogy family = new Genealogy (s);
      IO.say ("Your choices: L = list everyone\n"
            + "\t A = add a child of a person\n"
            + "\t S = search for children of 1 person\n"
            + "\t E = exit the program ");

      String choice = IO.askLine ("Choose L, A, S, or E ");
      while ( ! choice.equals ("E"))
      {  if (choice.equals ("A"))
            family.addChild (IO.askLine
                  ("parent of the person to be added? "));
         else if (choice.equals ("S"))
            family.listChildren (IO.askLine
                  ("parent of children to be listed? "));
         else  // it must be "L"
            family.listEverybody();
         choice = IO.askLine ("Choose L, A, S, or E ");
      }
      System.exit (0);
   }  //======================
}
```

17.2 Implementing The Genealogy Software With A Binary Tree

The abstraction we are using here is a non-empty collection of data values organized into a **general tree**. A general tree G has one data value known as its root value. And the tree G has a number of subtrees, also general trees, which contain the other data values of G. G could have 0 or 2 or 13 subtrees, as long as no two subtrees contain the same data value. The root value of each subtree of G is a child of the root value of G.

Implementing a general tree using a binary tree

You can implement a general tree in coding quite easily as follows: A TreeNode object stores the root data value of the general tree plus a linked list of its children, reading left-to-right for the most-recently-added down to the first-one-added. For a TreeNode `X`, `X.itsLeft` is the TreeNode containing the leftmost child of `X`. Also, `X.itsRight` is a reference to the next child of the parent of `X` to `X`'s right. Concretely...

- `X.itsData` is the root value of X.
- `C1 = X.itsLeft` contains X's leftmost child, the last-added child of X.
- `C2 = C1.itsRight` contains X's second child from the left, added just before `C1.itsData`.
- `C3 = C2.itsRight` contains X's third child from the left, etc.

Figure 17.1 is an example of a Genealogy dataset, with single letters for data values. The abstract general tree is on the right; the concrete TreeNodes are on the left.

If `b` is the TreeNode whose root value is B, then `b.itsRight` is the TreeNode with root value H and `b.itsLeft` is the TreeNode with root value C. The TreeNode `b.itsLeft.itsRight` has root value D. The TreeNode class in Listing 17.2 (see next page) can be used to implement the Genealogy class. The coding in Listing 17.2 contains some methods not needed for the Genealogy software, just to give you some feel for how to work with tree structures. All but the last one allow you to get information about a tree but not modify it.

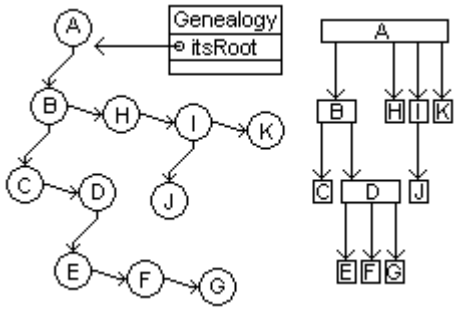


Figure 17.1 TreeNodes vs. general

We let **ET**, standing for Empty Tree, be one particular empty tree (i.e., having no data). This turns out to be advantageous at times; it is like a trailer node for linked lists. The following coding adds `newInfo` to the left of a `node`, thus making `newInfo` the most recently added child of the `node`'s data value. It puts the current linked list of the children of the `node` off to the right of the newly-added child, and it automatically puts `ET` to the left of the newly-added `node` to signal that this child has no children (as yet):

```
TreeNode newNode = new TreeNode (newInfo);
newNode.itsRight = node.itsLeft;
node.itsLeft = newNode;
```

The `size` method in the middle part of Listing 17.2 tells the number of data values in the TreeNode executor. It is zero if the executor is empty, otherwise it is 1 for the executor's root value plus the number of all data values to the left of the executor plus the number of all data values to the right of the executor. If you were to call `b.size()` for `b` being the node containing B in Figure 17.1, it would get 5 from its call of `c.size()` (`c` being the node containing C) and get 4 from its call of `h.size()` (`h` being the node containing H), and so it would return 10 (which is 1 + 5 + 4).

The `firstNode` method in Listing 17.2 returns the node that is furthest left in the executor's tree, assuming that the executor is a non-empty tree. This furthest-left node is (a) itself if the executor has an empty left subtree, otherwise it is (b) the furthest-left node of its left subtree. If you were to call `a.firstNode()` for `a` being the node containing A in Figure 17.1, it would call `b.firstNode()`, which would call `c.firstNode()`, which would return c.

The `deleteLastNode` method in Listing 17.2 deletes the furthest node to the right in the executor's tree, assuming that the executor has a non-empty node to its right. If the executor's right (call it R) has only the empty tree to its right, the executor replaces R by whatever is to R's left (which may or may not be empty). Otherwise the executor asks R to delete the furthest node to the right of R. If you were to call `c.deleteLastNode()` for `c` being the node containing C in Figure 17.1, it would see that `d.itsRight` is empty (`d` is the node containing D) and therefore set `c.itsRight` to be `e` (the node containing E).

Binary trees

The TreeNode class implements a binary tree structure. A **non-empty binary tree** is a structure consisting of one data value (its **root value**) plus two trees called its **left subtree** and its **right subtree**. It is represented in coding as a **node** object, called the **root node** of that tree. There are two kinds of nodes, internal nodes and external nodes. An **internal node** has a minimum of three attributes (instance variables): a data value stored there plus references to the root nodes of the left subtree and the right subtree. A **leaf node** is an internal node for which both subtrees are empty.

Listing 17.2 The TreeNode class used by the Genealogy class

```
public class TreeNode
{
   public static final TreeNode ET = new TreeNode (null);
   private Object itsData;  // the data value stored in this tree
   private TreeNode itsLeft  = ET; // left subtree of this tree
   private TreeNode itsRight = ET; // right subtree of this tree


   public TreeNode (Object given)
   {  itsData = given;
   }  //======================


   public boolean isEmpty()
   {  return this.itsData == null;  // generally, only ET
   }  //======================


   public Object getData()
   {  return itsData;
   }  //======================


   public TreeNode getLeft()
   {  return itsLeft;
   }  //======================


   public TreeNode getRight()
   {  return itsRight;
   }  //======================


   /** Return the number of data values in the tree. */

   public int size()
   {  return this.isEmpty() ? 0
             :  1 + this.itsLeft.size() + this.itsRight.size();
   }  //======================


   /** Return the leftmost node in the tree rooted at this.
    *  Throw an Exception if this is an empty tree.  */

   public TreeNode firstNode()
   {  return itsLeft.isEmpty()  ?  this : itsLeft.firstNode();
   }  //======================


   /** Precondition: this is a non-empty node with a non-empty
       node to its right.  Postcondition: the rightmost node of
       the subtree rooted at this is removed. */

   public void deleteLastNode()
   {  if (itsRight.itsRight.isEmpty())
         itsRight = itsRight.itsLeft;
      else
         itsRight.deleteLastNode();
   }  //======================
}
```

An **empty binary tree** does not contain any data or any subtrees; it is an **external node**. In figures, internal nodes are typically drawn as ellipses and external nodes are drawn as rectangles. Figure 17.2 shows all binary trees of 1 to 3 data values, though only the first three show the external nodes explicitly. We often draw only internal nodes; it is understood that each node drawn with less than two subtrees has an external node for each undrawn subtree.

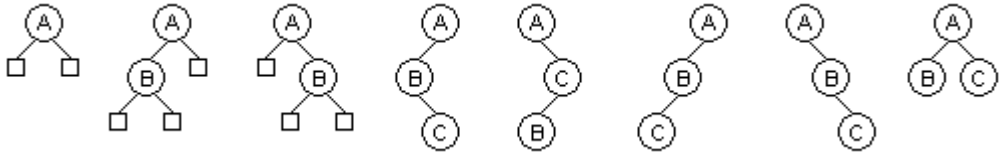


Figure 17.2 Eight binary trees, the first three with external nodes drawn

A **path** in a tree from one node to another is a sequence of nodes beginning with the first and ending with the second such that, for each time node Y follows node X in the sequence, node Y is the root of a subtree of node X. For instance, the fourth binary tree in Figure 17.2 has two paths of length 1 (from the A-node to the B-node and from the B-node to the C-node) and one path of length 2 (from the A-node via the B-node to the C-node).

A binary tree has a limitation on the way in which nodes can be connected, namely, you cannot have two different paths from one internal node to another internal node, and you cannot have a path from an internal node to itself. In particular, none of the nodes in the left subtree of a binary tree can be in the right subtree.

Another method that could be in the TreeNode class is the following, which prints all the data values in a given binary tree in order from left to right. The logic is (a) an empty tree has no data values to print, but (b) printing the data values in a non-empty tree requires that you first print all the data values in its left subtree in order from left to right, then print the root value, then print all the data values in its right subtree in order from left to right.

```
public void printInOrder()   // in the TreeNode class
{  if (this.isEmpty())
      return;  // a void method forbids returning with a value
   itsLeft.printInOrder();
   System.out.println (itsData.toString());
   itsRight.printInOrder();
}  //=======================
```

For the fourth tree in Figure 17.2, this method would print the data in the order B, C, A. For the eighth tree in Figure 17.2, this method would print the data in the order B, A, C.

Exercise 17.3 State the order in which the `printInOrder` method would print the data in the fifth, sixth, and seventh trees in Figure 17.2.
Exercise 17.4 Describe the action of `c.size()` on the tree in Figure 17.1, using the nomenclature of this section.
Exercise 17.5 Describe the action of `b.deleteLastNode()` on the tree in Figure 17.1, using the nomenclature of this section.
Exercise 17.6 Write a TreeNode method that returns the rightmost node in the tree.
Exercise 17.7 Write a TreeNode method that deletes the leftmost node in the tree.
Exercise 17.8 State the big-oh of the average execution time of the `printInOrder` method and of the `deleteLastNode` method.
Exercise 17.9* Draw all binary trees with four data values.
Exercise 17.10* Draw all binary trees with five data values.

17.3 Searching Through A Binary Tree

For the Genealogy software, you will sometimes need to search for the TreeNode that contains a particular person. This search process can potentially go through every node in the entire genealogy dataset, since there are no ordering properties to speed the search. If the search method finds the person somewhere in the dataset, the `addChild` method then asks the user for the child to be added for that person and puts the child in the dataset. Similarly, the `listChildren` method searches the dataset for a user-specified person and then lists all the children that person has, in last-in-first-out order.

One way of searching a binary tree is **breadth-first search**. This means that you first look at the data at the root, then at the data to the right of the root, then at the data to the left of the root, and only then at the data below them. And only after checking out all of the data two steps away from the root would you search through the data that is three steps away from the root. This kind of search can be controlled by a queue as shown in Listing 17.3.

Listing 17.3 A non-recursive search method in TreeNode using a queue

```
/** The search process implemented with a queue. Return the
 *  TreeNode containing given, or ET if given is not there.
 *  Precondition:  The executor is not empty. */

public TreeNode search (Object given)
{  if (this.itsData.equals (given))
      return this;
   QueueADT people = new NodeQueue();
   if ( ! this.itsLeft.isEmpty())
      people.enqueue (this.itsLeft);
   while ( ! people.isEmpty())
   {  TreeNode node = (TreeNode) people.dequeue();
      if (node.itsData.equals (given))
         return node;
      if ( ! node.itsRight.isEmpty())
         people.enqueue (node.itsRight);
      if ( ! node.itsLeft.isEmpty())
         people.enqueue (node.itsLeft);
   }
   return ET;
}  //======================
```

QueueADT is described in Chapter Fourteen; all you really have to know for this chapter is that the `enqueue` method adds an Object value (its parameter) to the rear, the `dequeue` method removes an Object value from the front, and the `isEmpty` method checks to see whether you have a value to remove. **NodeQueue** is one particular implementation of the QueueADT interface.

Similarly, **StackADT** is described in Chapter Fourteen; all you really have to know for this chapter is that the `push` method adds an Object value (its parameter) to the top, the `pop` method removes an Object value from the top, and the `isEmpty` method checks to see whether you have a value to remove. **NodeStack** is one particular implementation of the StackADT interface.

This search method visits the root node, which is at the first **level** of the binary tree, testing `this.itsData.equals (given)`. It then it visits the node on the second level of the binary tree: It puts `this.itsLeft` in the queue, then starts the loop, whose first action is to remove from the Queue `node = this.itsLeft` (it ignores `this.itsRight`, because the root of a Genealogy dataset has nothing to its right).

Then it visits all nodes on the third level of the binary tree: It puts `node.itsRight` and `node.itsLeft` on the Queue, etc. This is considered breadth-first whether you look at the right before the left or vice versa. In the earlier Figure 17.1, this search process would visit the data values in this order: A, then B, then H and C (2 levels away), then I and D (3 levels away), then K, J, and E (4 levels away), then F, then G. Note that levels are expressed in terms of the binary tree structure, not the data structure itself.

Depth-first search

Depth-first search means that, after looking at the data in a particular node, you look at all the data in its subtree before you look elsewhere. Concretely, first you check out the data in the root. If you find what you want there, you quit. Otherwise you search through the entire left subtree of the root, going down however many levels are necessary. Only if you do not find what you are looking for there do you go on to the right subtree of the root and search through all of the levels of that subtree.

A non-recursive depth-first search can be coded exactly the way it is done in Listing 17.3 except you replace "Queue" by "Stack" and therefore "enqueue" by "push" and "dequeue" by "pop". In the earlier Figure 17.1, this search process would visit the data values in this order: A, then B, then B's left subtree C, D, E, F, and G, then B's right subtree H, I, J, and K.

For a recursive depth-first search, you would first check out the data in the root (if any; if the root is empty, then of course the result of the search is an empty tree). If you do not find what you want in the root, call the search method recursively for the left subtree of the root and see what you get back from that call. If you get back anything but the empty tree, you have found the desired TreeNode value and you do not need to search further. Otherwise, search the right subtree of the root and return whatever it produces.

In Figure 17.3, the breadth-first coding in Listing 17.3 will look at the information in the following order (one level at a time): 54, 76, 28, 90, 64, 40, 17. But depth-first search would look at the information in the following order: 54, 28, 17, 40, 76, 64, 90.

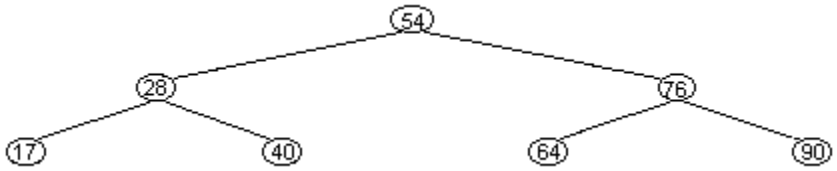


Figure 17.3 A binary tree with 7 nodes

Traversals

A recursive depth-first search is called **left-to-right preorder traversal**. "Traversal" means you visit each node in the tree (potentially). "Left-to-right" means that you visit the nodes in the left subtree before you visit any node in the right subtree; the alternative is right-to-left traversal. "Preorder" means that you visit the root node of each subtree before you visit any of the nodes in its subtrees. The alternatives to preorder are **postorder traversal** (visit the root node after you visit all the nodes of the subtrees) and **inorder traversal** (visit the root node in between visiting the nodes of the subtrees).

For instance, the `printInOrder` method coded at the end of the preceding section is a left-to-right inorder traversal. An example of right-to-left inorder traversal is the following `toString` method to return a String value containing a representation of the values in the entire binary tree. It is right-to-left because Java always evaluates the operands of the plus operator from left to right.

```
public String toString()  // in the TreeNode class
{  if (this.isEmpty())
      return "";
   else
      return " (" + itsRight.toString() + itsData.toString()
                  + itsLeft.toString() + ") ";
}  //=======================
```

For the binary tree in Figure 17.3, this `toString` method would return the String form of the right subtree followed by 54 followed by the String form of the left subtree. The String form of the right subtree is "((90) 76 (64))". The String form of the left subtree is "((40) 28 (17))". So the output from a call of `toString` would be "(((90) 76 (64)) 54 ((40) 28 (17)))".

Listing 17.4 (see next page) codes the Genealogy class. It calls on the `search` method in Listing 17.3 and a recursive `listAll` method in TreeNode that traverses the tree, listing all the data values with each value indented appropriately. This `listAll` method is left as an exercise. It is a right-to-left postorder traversal. In the Genealogy class, `itsRoot` is a reference to the root of a binary tree containing the family members.

Exercise 17.11 Consider a binary tree with the same structure as shown in Figure 17.3, but with different data values in the seven nodes. If the order of the data values produced by the breadth-first search method in Listing 17.3 is A,B,C,D,E,F,G, then what is the order produced by the depth-first search method described in this section?
Exercise 17.12 Consider a binary tree with the same structure as shown in Figure 17.3, but with different data values in the seven nodes. If the order of the data values produced by the recursive depth-first search method described in this section is A,B,C,D,E,F,G, then what is the order produced by the breadth-first search method in Listing 17.3?
Exercise 17.13 Is it possible to have a binary tree of seven nodes in which both of the algorithms mentioned in the preceding exercise produce exactly the same sequence of data values? If so, what does it look like?
Exercise 17.14 (harder) What is the relation between the number of parentheses in the output of the `toString` method at the end of this section and the levels of the data values?
Exercise 17.15** Write the `search` function that must be added to the TreeNode class in order to make the Genealogy methods work correctly. It does not assume that the dataset is ordered in any way. You are to code it recursively as described in the text, without a stack or queue. Be sure to have it stop the search as soon as it finds a match.
Exercise 17.16** Genealogy's `listEverybody` method calls a recursive tree traversal method `itsRoot.listAll("")` with one parameter, which is a String containing the right number of blanks to indent the executor node's data. Each data value should be indented 3 blanks for every generation it is removed from the root data. Code `listAll` so that it prints all the data values stored in the right subtree of the root node, then prints all the data values stored in the left subtree of the root node, then prints the data in the root node. For instance, for the data in Figure 17.1, it will print A at the bottom of the list; it will print K, I, H, and B in that order, each indented 3 blanks (since they are the children of A), with K first; and it will print J just before I and indented 6 blanks.

Listing 17.4 The Genealogy class

```
public class Genealogy
{
   private TreeNode itsRoot;   // the root node of the tree

   /** Create a dataset of parent-child relations with lilith as
    *  the root data value and no other values in the dataset. */

   Genealogy (String lilith)
   {  itsRoot = new TreeNode (lilith);
   }  //=======================

   /** If the given parent is in the dataset, ask for a child of
    *  that parent and then add that child to the dataset. */

   public void addChild (String parent)
   {  TreeNode node = itsRoot.search (parent);
      if (node == TreeNode.ET)
         IO.say ("I'm sorry, that person is not in the family");
      else
         node.pushLeft (IO.askLine ("What's the child's name?"));
   }  //=======================

   /** List all children of the parent. */

   public void listChildren (String parent)
   {  TreeNode node = itsRoot.search (parent);
      if (node == TreeNode.ET)
         IO.say ("I'm sorry, that person is not in the family");
      else if (node.getLeft() == TreeNode.ET)
         IO.say ("None are in the dataset");
      else
      {  TreeNode kid = node.getLeft();
         String s = kid.getData().toString();
         for (kid = kid.getRight();  kid != TreeNode.ET;
                                     kid = kid.getRight())
            s += ", " + kid.getData().toString();
         IO.say (s);
      }
   }  //=======================

   /** List all values in the dataset, one per output line,
    *  with all children of any data value X listed before X
    *  and in the order they were entered (right-to-left
    *  postorder), and each indented 3 spaces further than X. */

   public void listEverybody()
   {  itsRoot.listAll ("");  // left as a recursive exercise
   }  //=======================
}

// The following method is added to the TreeNode class

   public void pushLeft (Object newInfo)
   {  TreeNode newNode = new TreeNode (newInfo);
      newNode.itsRight = this.itsLeft;
      this.itsLeft = newNode;
   }  //=======================
```

17.4 Implementing The Mapping Interface With A Binary Search Tree

The **Mapping** interface (Listing 16.4) simplifies Sun's standard library Map interface:

```
public Object put (Object id, Object value);  // add id/value
public boolean containsKey (Object id); // is this id in it?
public Object get (Object id); // return the value for this id
public Object remove (Object id); // remove the id/value
public boolean isEmpty();         // has it no id/value pairs?
public int size();                // how many id/value pairs?
public java.util.Iterator iterator(); // for listing all pairs
```

One way to implement the Mapping interface is to use a binary search tree, as long as all the keys are from a class that implements Comparable (and so contains the usual `compareTo` method). A nonempty tree contains one MapEntry, its root data, plus two subtrees, one on its left and one on its right. An empty tree contains no entry or subtree. A **MapEntry** object (described in Listing 16.5) has two final Object instance variables whose values can be obtained using `me.getKey()` and `me.getValue()`.

In a **binary search tree**, keys in a left subtree are less than the key for the root data, and keys in a right subtree are greater than or equal to the key for the root data. Mappings do not allow duplicate keys. So if `Lkey` is any key from a MapEntry data value in X's left subtree, and if `Rkey` is any key from a MapEntry data value in X's right subtree, then `Lkey.compareTo (((MapEntry) X.itsData).getKey()) < 0` and also `Rkey.compareTo (((MapEntry) X.itsData).getKey()) > 0`.

If the tree is well balanced, then about half of the entries will be in the left subtree and about half will be in the right subtree. A perfectly balanced tree with 31 entries has 15 entries in its left subtree and 15 entries in its right subtree (and so the root data would be the middle value as determined by `compareTo`). Each of those subtrees has 7 entries in each of its two subtrees, etc. The tree has a total of five levels; the fifth level has 16 subtrees each with two empty subtrees. Figure 17.1 shows such a tree.

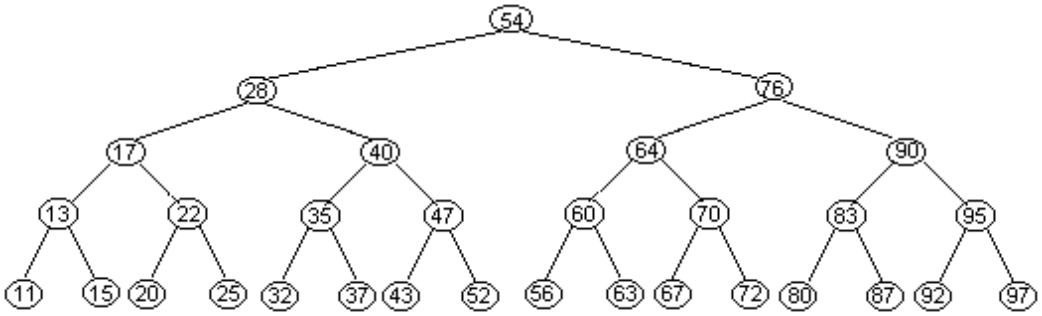


Figure 17.4 Example of a balanced binary search tree with 31 entries

Searching in a binary search tree

When you search for a particular key in this 31-node tree, you only need to look at at most five different entries before you find the entry you are looking for (or find that it is not in the data structure at all). Specifically, you compare the given `id` with the root data 54. That tells you whether to look in its left subtree or in its right subtree (unless it equals the root data). Either way, you are in a subtree with only 15 possibilities left instead of the original 31. When you compare the given `id` with the root of the subtree you are in, it tells you whether to look in its left subtree or in its right subtree. That reduces the number of possibilities to 7, again cutting them in half (unless you have already found it on the first two levels). This process continues for at most five levels.

There are only ten levels in a balanced binary tree with a thousand entries and only twenty levels in a balanced binary tree with a million entries. So you can find a value you are looking for by looking at only twenty different values among a million entries. That is just not possible with a linked list. Of course, you get the same speed using binary search in an ordered array (in Section 13.3). A balanced binary search tree has a close correspondence with an ordered array (this is why it is called a binary search tree): The root data is roughly the middle value in the array; the root data of the left subtree is roughly the value a quarter of the way up from the bottom of the array; etc.

Inserting in a binary search tree

The drawback to using an ordered array is that putting a new entry in the array or taking an existing one out is so slow. If you have a million entries, you may have to make almost a million assignments of variables to make room for a new one that is near the front of the array. And you may have to make almost a million assignments to move entries down when you remove an existing one that is near the front of the array.

With a binary search tree, you only have to make less than half-a-dozen assignments to put a new entry in the data structure or remove an existing entry, once you find where it goes. That is an enormous amount of time saved. And that is true even if you have a million million million entries.

Our **BintMap** implementation of Mapping is in Listing 17.5 (see next page) has just one instance variable, the TreeNode stored in `itsRoot`. Each TreeNode in the binary search tree stores a MapEntry value in `itsData` (though `itsData` is declared as type Object). To simplify coding comparisons of Comparable `id` values with the key value in `itsData`, we use the `compare` method defined in the bottom of Listing 17.5.

Since the Object class has a `hashCode` method, we could instead use each ID's int `hashCode` value (described in Section 16.8) to decide where a data value is stored. Then the IDs would not have to be Comparable; they would just have to come from a class of objects that overrides the Object `hashCode` method appropriately.

Each TreeNode X has references to its two subtrees, `X.itsLeft` and `X.itsRight`. The nodes at the roots of those two subtrees are the **left child** and **right child** of X. X is their **parent**. If a node has no data to its left, `itsLeft` is an empty tree (normally `ET`), and similarly for `itsRight`.

The lookUp method

The `size` and `isEmpty` methods for a BintMap simply return `itsRoot.size()` and `itsRoot.isEmpty()`, respectively (calling methods in Listing 17.2). The `containsKey` method returns `false` if the `id` parameter is null or if the root node has no data. Otherwise `containsKey` calls a recursive `lookUp` method in the TreeNode class to return the node that has `itsData.getKey()` equal to `id` (if any). The logic of the `lookUp` method follows the structure of a binary search tree:

- If the `id` is less than the root data, the search should be made in the left subtree, and the result that search produces is what the method returns.
- If the `id` is larger than the root data, the search should be made in the right subtree, and the result that search produces is what the method returns.
- The only case left is that the `id` equals the root data, in which case your search is successfully completed and you can return this current node.

For the `get` method, if the `id` is null or if the executor is empty of data or if `lookUp` returns null, `get` returns null, otherwise `get` returns the value part of the id/value pair in the node that `lookUp` returns.

Listing 17.5 The BintMap class, partially done

```
//Methods throw a RuntimeException for non-Comparable ids.

public class BintMap implements Mapping
{
   private TreeNode itsRoot;


   public BintMap()
   {  itsRoot = TreeNode.ET;
   }  //======================

   public int size()
   {  return itsRoot.size();
   }  //======================

   public boolean containsKey (Object id)
   {  return id != null && ! this.isEmpty()
                  && itsRoot.lookUp ((Comparable) id) != null;
   }  //======================

   public Object get (Object id)
   {  if (id == null || this.isEmpty())
         return null;
      TreeNode loc = itsRoot.lookUp ((Comparable) id);
      return (loc == null)  ?  null
            :  ((MapEntry) loc.getData()).getValue();
   }  //======================
}



// The following 2 methods are added to the TreeNode class

   /** Return null if id is nowhere in the subtree rooted at
    *  this node.  Otherwise return the TreeNode containing id.
    *  Precondition: id is not null, nor is this.itsData; also,
    *  this tree is a binary search tree. */

   public TreeNode lookUp (Comparable id)
   {  if (compare (id, itsData) < 0)
         return itsLeft.isEmpty() ? null : itsLeft.lookUp (id);
      if (compare (id, itsData) > 0)
         return itsRight.isEmpty() ? null : itsRight.lookUp (id);
      return this;  // this is the node with the id
   }  //======================

   private static int compare (Comparable id, Object data)
   {  return id.compareTo (((MapEntry) data).getKey());
   }  //======================
```

Example for `lookUp` If the root node in Figure 17.4 performs `lookUp` for the `id` 47, it asks the node to its left containing 28 to perform `lookUp`, which asks the node to its right containing 40 to perform `lookUp`, which asks the node to its right containing 47 to perform `lookUp`, which returns itself. But if the `id` had been 45, the node containing 47 would then have asked the node to its left containing 43 to perform `lookUp`, which would have seen `ET` to its right and therefore returned null.

The put method

The `put` method for the BintMap class is in Listing 17.6. After first checking that the `id` parameter is not null and the BintMap is not completely empty, `put` calls a recursive TreeNode method structured similarly to `lookUp`. Closely compare `putRecursive` with `lookUp`. Note the **indirect recursion**: `putRecursive` calls one of two other methods, each of which can call `putRecursive` again.

Listing 17.6 The put method for the BintMap class

```
   public Object put (Object id, Object value)
   {  if (id == null)                                          //1
         return null;                                          //2
      if (this.isEmpty())                                      //3
      {  itsRoot = new TreeNode (new MapEntry (id, value));    //4
         return null;                                          //5
      }                                                        //6
      return itsRoot.putRecursive ((Comparable) id, value);    //7
   }  //=====================

// The following 3 methods are added to the TreeNode class

   /** Add the id/value pair to the non-empty binary search tree.
    *  Return the value the id previously had (null if none). */

   public Object putRecursive (Comparable id, Object val)
   {  if (compare (id, itsData) < 0)    // go left             //8
         return putInLeftSubtree (id, val);                    //9
      if (compare (id, itsData) > 0)    // go right            //10
         return putInRightSubtree (id, val);                   //11
      Object valueToReturn = ((MapEntry) itsData).getValue();//12
      itsData = new MapEntry (id, val);                        //13
      return valueToReturn;                                    //14
   }  //=====================

   private Object putInLeftSubtree (Comparable id, Object val)
   {  if (itsLeft.isEmpty())                                   //15
      {  itsLeft = new TreeNode (new MapEntry (id, val));      //16
         return null;                                          //17
      }                                                        //18
      else                                                     //19
         return itsLeft.putRecursive (id, val);                //20
   }  //=====================

   private Object putInRightSubtree (Comparable id, Object val)
   {  if (itsRight.isEmpty())                                  //21
      {  itsRight = new TreeNode (new MapEntry (id, val));     //22
         return null;                                          //23
      }                                                        //24
      else                                                     //25
         return itsRight.putRecursive (id, val);               //26
   }  //=====================
```

The executor of the `putRecursive` method puts the MapEntry in the left or right subtree depending on whether the `id` is smaller or larger than its data. But if it is equal, it simply replaces the corresponding value. Putting a value in the left subtree consists in calling the `putRecursive` method unless the left subtree is empty, in which case it is replaced by a one-node subtree containing the new MapEntry value. Putting a value in the right subtree is line-for-line analogous.

Example for `put` If the root node in Figure 17.4 performs `putRecursive` for the id 45, it asks the node to its left containing 28 to perform `putRecursive`, which asks the node to its right containing 40 to perform `putRecursive`, which asks the node to its right containing 47 to perform `putRecursive`, which asks the node to its left containing 43 to perform `putRecursive`, which sees that 45 goes to its right where it has an empty subtree, so it creates a new node to its right and stores the 45 in it.

The remove method

The hardest method to implement for a binary search tree is the `remove` method. This method requires that you search through the tree to find a data value whose key matches the one given by the parameter. Then you remove it without altering the relationships of the other data values to each other. You are to return null if the search fails.

If the root node is empty or the `id` is null, you can simply return null. Otherwise you need to find the node that contains the data with that `id`; `lookUp` will do this for you. If `lookUp` returns null, the `id` is not in the BintMap. Otherwise, you can extract the data from that node, remove that data from the binary tree, and return the data's value. This logic is in the upper part of Listing 17.7.

Listing 17.7 The remove method for the BintMap class

```
   public Object remove (Object id)
   {  if (id == null || this.isEmpty())                        //1
         return null;                                          //2
      TreeNode loc = itsRoot.lookUp ((Comparable) id);         //3
      if (loc == null)                                         //4
         return null;                                          //5
      MapEntry data = (MapEntry) loc.getData();                //6
      loc.removeData();                                        //7
      return data.getValue();                                  //8
   }  //======================


// The following method is added to the TreeNode class

   /** Precondition: The executor is a non-empty binary search
    *  tree.  Postcondition:  The data in the executor node is
    *  removed, leaving a binary search tree 1 node smaller.  */

   public void removeData()
   {  if (itsRight.isEmpty())                                  //9
      {  itsData  = itsLeft.itsData;                           //10
         itsRight = itsLeft.itsRight;                          //11
         itsLeft  = itsLeft.itsLeft;                           //12
      }                                                        //13
      else if (itsRight.itsLeft.isEmpty())                     //14
      {  itsData  = itsRight.itsData;                          //15
         itsRight = itsRight.itsRight;                         //16
      }                                                        //17
      else                                                     //18
      {  TreeNode p = this.itsRight;                           //19
         while ( ! p.itsLeft.itsLeft.isEmpty())                //20
            p = p.itsLeft;                                     //21
         this.itsData = p.itsLeft.itsData;                     //22
         p.itsLeft = p.itsLeft.itsRight;                       //23
      }                                                        //24
   }  //======================
```

For the `removeData` method: Removing the data value in a linked list is easy – you simply copy the values from the next node into the current node, thereby replacing its data value by the next node's data and then bypassing the next node. You can also do this in a binary tree if the current node only has a left subtree, i.e., its right subtree is empty (this is in lines 9-12). What do you do if the right subtree is not empty? You obtain a replacement data value from the right subtree.

With which value in the current node's right subtree do you replace the current node's data? Only the data in the leftmost node, since that is the data value that comes immediately after the current node's data value. And since that leftmost node has an empty left subtree, you may replace it by its own right subtree without altering the ordering relationships of the data values. Study Listing 17.7 carefully to see this.

Example for `remove` If the root node in Figure 17.4 performs `remove` for the id 43, the `lookUp` method goes through the nodes containing 54, 28, 40, and 47 to eventually return the node containing 43. When that node is asked to `removeData`, it copies the data from its left subtree, which makes it an empty node. But if the root node performs `remove` for the id 54, `lookUp` returns the root node. Since the root node has a non-empty right subtree, it copies the 56 up from the "next node" (the leftmost node in its right subtree) and makes the left subtree of the node containing 60 be the empty tree.

Creating a balanced tree from an array

If you have a filled array of one or more MapEntry values in ascending order (every component having a MapEntry value), you could create a nicely balanced binary tree from those values by calling the following method in the TreeNode class with the statement `make(item, 0, item.length - 1)`:

```
public static TreeNode make (Object[] item, int lo, int hi)
{  int middle = (lo + hi + 1) / 2;  // halfway from lo to hi
   TreeNode root = new TreeNode (item[middle]);
   if (lo < middle)
      root.itsLeft = make (item, lo, middle - 1);
   if (middle < hi)
      root.itsRight = make (item, middle + 1, hi);
   return root;
}  //=======================
```

Exercise 17.17 Rewrite the `lookUp` method to only call `compare` at most one time for each pair of values, instead of wasting execution time calling it twice.
Exercise 17.18 Rewrite the `removeData` method so that it checks for an empty left subtree and makes the easy removal in that case. Does this improve overall execution time?
Exercise 17.19 (harder) Write a TreeNode method `public TreeNode copy()`: The non-empty executor returns a new binary search tree containing exactly the same entries as the executor's binary search tree and in the same relative positions. Use recursion.
Exercise 17.20 (harder) Write a TreeNode method `public int under (Comparable id)`: The executor tells how many of its MapEntries have a key less than that of the given key.
Exercise 17.21* Write a BintMap method `public Comparable firstKey()` that returns the smallest key. Return null if the executor is empty. Use recursion.
Exercise 17.22* Rewrite the `lookUp` method in Listing 17.5 without using recursion.
Exercise 17.23* Write a BintMap method `public BintMap reverse()`: the executor returns a new binary search tree with the same entries in the opposite order.
Exercise 17.24** Write a BintMap method `public boolean equals (BintMap given)`: The executor tells whether the BintMap parameter has the same key/value pairs in the same order and the same positions in the tree as the executor. Use recursion.
Exercise 17.25** Write out the internal invariant for the BintMap class.

17.5 Implementing The Iterator Interface For A Binary Search Tree

A Mapping needs an Iterator object so a client can obtain the values in the Mapping one at a time. An obvious implementation is to keep track of the TreeNode that contains the information to be returned by a call of `next()`. This is like what the array implementation and the linked list implementation in Chapter Sixteen did:

In ArrayMap, `next()` returns `itsItem[itsPos]` if `itsPos < itsSize`.
In NodeMap, `next()` returns `itsPos.itsData` if `itsPos.itsData != null`.

For the BintMap class, we could have `next()` return `itsPos.itsData` where `itsPos` is a TreeNode, just as we did for the NodeMap class. This coding is in Listing 17.8 (see next page). Our Mapping iterators disallow `remove` to make this chapter simpler. When we need to move on to the next TreeNode, we do one of two things:

- If the right subtree of `itsPos` contains any data, then the next value is the first value in that right subtree. So we set `itsPos` to be the leftmost node in that right subtree.
- Otherwise, the next value after the one in `itsPos` is a value that is on the path from the root node down to `itsPos`. It will be in a node that contains `itsPos` in its left subtree, since the value in `itsPos` comes before it. If there are two or more nodes on that path that have `itsPos` in the node's left subtree, we want the one that is furthest down the path.

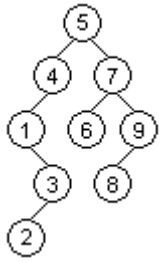


Figure 17.5 Binary search tree

The algorithm described above is left-to-right inorder traversal of the nodes, which is in ascending order using `compareTo`. Figure 17.5 shows a binary search tree with nine values in it. The effects of the first few accesses to the iterator are as follows:

- The constructor sets `itsPos` to refer to the leftmost node, which contains 1.
- A call of `next()` returns that 1 and then, when it sees that `itsPos.itsRight` is not the empty tree, sets `itsPos` to the leftmost node in that right subtree, which contains 2.
- Another call of `next()` returns that 2 and then, when it sees that `itsPos` has an empty right subtree, starts from the root to search down the tree for `itsPos`. It notes that `itsPos` is to the left of the node containing 5, and of the node containing 4, and of the node containing 3. Since the node containing 3 is lowest in the tree of all of those, it sets `itsPos` to that node containing 3.
- Another call of `next()` returns that 3 and then, when it sees that `itsPos` has an empty right subtree, starts from the root to search down the tree for `itsPos`. It notes that `itsPos` is to the left of the node containing 5 and also of the node containing 4. Since the latter is lower in the tree, it sets `itsPos` to that node containing 4.

Listing 17.8 uses a throw statement when someone calls a method that should not be called (`remove`), or calls a method under conditions when it should not be called (`next` when `hasNext()` is false). That is a bug in the coding that calls it. Notification of the bug is through the following kind of statement, which immediately terminates the method containing the statement:

```
throw new RuntimeException ("some explanatory message");
```

Listing 17.8 The MapIt class inside the BintMap class, providing iterators

```
   public java.util.Iterator iterator()    // member of BintMap
   {  return new MapIt (this.itsRoot);                          //1
   }  //=====================


   private static class MapIt implements java.util.Iterator
   {
      private TreeNode itsPos;
      private TreeNode itsRoot;


      public MapIt (TreeNode given)
      {  itsRoot = given;                                       //2
         itsPos = given.isEmpty() ? null : given.firstNode();//3
      }  //=====================


      public boolean hasNext()
      {  return itsPos != null;                                 //4
      }  //=====================


      public Object next()
      {  if ( ! hasNext())                                      //5
            throw new java.util.NoSuchElementException          //6
                     ("iterator has no next element!");         //7
         Object valueToReturn = itsPos.getData();               //8
         itsPos = itsPos.nextIn (itsRoot);                      //9
         return valueToReturn;                                  //10
      }  //=====================

      public void remove()
      {  throw new UnsupportedOperationException ("no remove!");
      }  //=====================
   }


// The following method is added to the TreeNode class

   /** Return the next TreeNode after the executor that appears
    *  in left-right inorder traversal; return null if there is
    *  none. Precondition: The executor is not empty, . */

   public TreeNode nextIn (TreeNode ancestor)
   {  if ( ! this.itsRight.isEmpty())                           //12
         return this.itsRight.firstNode();                      //13
      Comparable id = (Comparable) ((MapEntry) itsData).getKey();
      TreeNode lastLeftTurn = null;                             //15
      while (ancestor != this)                                  //16
      {  if (compare (id, ancestor.itsData) > 0)                //17
            ancestor = ancestor.itsRight;                       //18
         else                                                   //19
         {  lastLeftTurn = ancestor;                            //20
            ancestor = ancestor.itsLeft;                        //21
         }                                                      //22
      }                                                         //23
      return lastLeftTurn;                                      //24
   }  //=====================
```

Using parent pointers

The `nextIn` method has to go through several nodes, starting from the root of the tree and moving down to the current position, in order to move on to the next position. It would be helpful if we could avoid the time it takes to execute that loop. One way is to have each node keep track of its parent. That is, we put another instance variable in the TreeNode class in addition to `itsLeft`, `itsRight`, and `itsData`:

```
private TreeNode itsParent = null;  // add to TreeNode
```

Whenever you consider storing additional information in an object, you should consider two things: What do you gain and what do you lose? This is like decisions in a business venture: What is your added revenue and what are your added costs? You need to make sure you will be making a profit if you make a change.

What you gain by adding parent information is a faster implementation of the Iterator `next` method. None of the other methods in the BintMap class profit from having the parent information. What you lose is the space required to store the added information and the time required to update the information when needed.

Parent information changes whenever you add or remove a node in the binary tree. So for the `put` logic in Listing 17.6, set the parent of the root node to null and add the following statements directly after the two statements that mention `new TreeNode` (lines 16 and 22):

```
this.itsLeft.itsParent = this;   // after line 16
this.itsRight.itsParent = this;  // after line 22
```

For the `remove` logic in Listing 17.7, you need to change the record of the parent in the nodes directly below the one whose data you are changing. Specifically, you add these three lines right after deleting a data value whose right subtree is empty:

```
if (this.itsData != null)    // after line 12
{  this.itsLeft.itsParent = this;
   this.itsRight.itsParent = this;
}
```

The only other change required in Listing 17.7 is that each node that is moved up into another node's position has to have its parent information corrected. So have the following statement after deleting the node to the right of `this`. Its new right child may be empty, but it does not matter what an empty node thinks its parent is:

```
this.itsRight.itsParent = this;  // after line 16
```

Finally, put this statement after deleting a node to the left of p:

```
p.itsLeft.itsParent = p;         // after line 23
```

The only advantage of having this parent information is that you may now make the `nextIn` method execute faster, by replacing everything after the first two lines by the following coding to handle the case when `this.itsRight` is empty:

```
Node p = this;
while (p.itsParent != null && p.itsParent.itsLeft != p)
   p = p.itsParent;
return p.itsParent;
```

Implementing an Iterator with a threaded binary search tree

Adding a parent pointer as just described saves execution time, but even more time would be saved if we had an `itsNext` value stored in each node, instead of an `itsParent` value, where `itsNext` tells the node containing the very next value in sequence in the tree. Then we could replace the entire body of the `nextIn` method in Listing 17.8 by one statement:

```
itsPos = itsPos.itsNext;  // new body of nextIn
```

On the other hand, we need to go to more trouble to adjust the `itsNext` values each time we add or remove a node in the tree. Specifically, we declare a new instance variable in the TreeNode class as follows (instead of having `itsParent`):

```
private TreeNode itsNext = null;  // add to TreeNode
```

Then we assign it the correct value when we put a new data value in the tree. For the `put` logic in Listing 17.6, we have to make some adjustments around lines 16 and 22. The adjustment for line 16 is left as an exercise; the adjustment for line 22 is to add the following two statements after it:

```
itsRight.itsNext = this.itsNext;  // after line 22
this.itsNext = itsRight;
```

Some adjustment also has to be made for the `remove` logic in Listing 17.7, at lines 10, 16, and 20. Line 10 is left as an exercise. The adjustment for the other two is straightforward – just add the following statement at line 16 and also at line 19:

```
this.itsNext = this.itsNext.itsNext;  // in removeData
```

Historical Note This kind of link is called a **thread** through the tree. Back in the days when RAM was expensive, it was standard to put the `itsNext` values and the `itsRight` values in the same storage space and just add a boolean value to each node to tell what kind of value was in that storage space (and similarly for an `itsPrevious` value sharing space with the `itsLeft` value). They would however only store something in `itsNext` when its right subtree was empty.

Exercise 17.26 Write statements that delete a TreeNode node `toDelete` from a binary tree when `toDelete` is not empty and `toDelete.itsRight` is empty and `toDelete.itsParent != null`. Assume you have the `itsParent` values.
Exercise 17.27 (harder) Write a private recursive TreeNode method so that you can replace everything in `nextIn` starting from `TreeNode lastLeftTurn` by the statement: `return ancestor.lastLeft (null, id);` and still do the same thing.
Exercise 17.28 (harder) Determine the amount of execution time saved by adding the parent information described in this section for a well-balanced tree.
Exercise 17.29* Essay: What are the advantages and disadvantages of having each MapIt iterator create a stack and put on it all of the TreeNodes at which the iterator made a left turn on its way to the current position `itsPos`? Compare this approach with using the `itsNext` instance variable.
Exercise 17.30** Revise the `put` logic at line 16 in Listing 17.6 to allow for the `itsNext` value.
Exercise 17.31** Revise the `remove` logic at lines 10 through 12 in Listing 17.7 to allow for the `itsNext` value.
Exercise 17.32** Implement the Iterator's `remove` method for Listing 17.8. Hint: You cannot simply call TreeNode's `removeData` method in Listing 17.7.

Part B Enrichment And Reinforcement

17.6 More Tree Traversals

If you want to put all the data values from a binary tree on a queue for later use, you have a choice of six standard depth-first traversal processes. You can process all the data in the left subtree either before or after all the data in the right subtree (left-to-right or right-to-left). Whichever of these two you choose, you can process the data in the root either before all subtree data (preorder traversal), or after all subtree data (postorder traversal), or in between the data of the two subtrees (inorder traversal).

Figure 17.6 shows three of these standard traversals. Start at the down-arrow at the top of one of the figures and follow the dotted line. Process each of the seven data values when you come to the little rectangle pointing to it. For the left-to-right preorder traversal, the order of processing is 17, 13, 11, 15, 22, 20, 25. For the left-to-right inorder traversal, the order of processing is 11, 13, 15, 17, 20, 22, 25. For the right-to-left postorder traversal, the order of processing is 17, 22, 25, 20, 13, 15, 11.

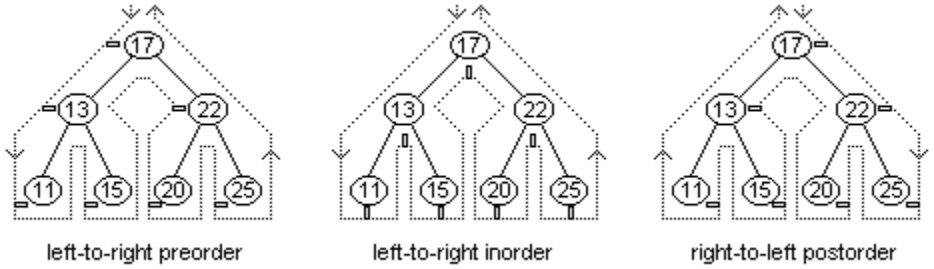


Figure 17.6 Three of the six standard traversals (start at the down-arrow)

The following right-to-left postorder traversal method, illustrated in the middle of Figure 17.6, could be in the TreeNode class, with the parameter being any QueueADT object to which you want to add all the data values:

```
public void postorderTraverseRL (QueueADT queue)// in TreeNode
{  if (isEmpty())
      return;
   itsRight.postorderTraverseLR (queue);
   itsLeft.postorderTraverseLR (queue);
   queue.enqueue (itsData);
}  //=======================
```

There are five more analogous methods, one for each kind of standard depth-first traversal. Note that the queue produced by a preorder traversal of a binary search tree can be used to produce an exact copy of that tree, with the same parent-child relations.

The most commonly-used traversal of a binary tree is left-to-right inorder traversal (as shown in the middle of Figure 17.6). This produces the data values in the **natural order** (i.e., using `compareTo`) if the tree is a binary search tree. An iterator could be made for BintMap using this traversal: When the iterator is constructed, it traverses the entire tree recursively and puts the data values on a queue; each time `next` is called, the next data value is removed from the queue.

The coding for this **QueueMapIt** iterator class for BintMap is in Listing 17.9 (see next page). Note that `next` does not need an explicit throw statement, since the queue will throw a RuntimeException if anyone tries to take a data value from it when it is empty.

Listing 17.9 The QueueMapIt class, providing iterators for BintMap

```
   public java.util.Iterator iterator()    // member of BintMap
   {  return new QueueMapIt (this.itsRoot);
   }  //=====================


   public class QueueMapIt implements java.util.Iterator
   {
      private QueueADT itsQueue = new NodeQueue();


      public QueueMapIt (TreeNode given)
      {  given.inorderTraverseLR (itsQueue);
      }  //=====================

      public boolean hasNext()
      {  return ! itsQueue.isEmpty();
      }  //=====================

      public Object next()
      {  return itsQueue.dequeue();
      }  //=====================

      public void remove()
      {  throw new UnsupportedOperationException ("no remove!");
      }  //=====================
   }



// The following method is added to the TreeNode class

   /** Add to the given queue all data values in the standard
    *  left-to-right inorder traversal. */

   public void inorderTraverseLR (QueueADT queue)
   {  if (isEmpty())
         return;
      itsLeft.inorderTraverseLR (queue);
      queue.enqueue (itsData);
      itsRight.inorderTraverseLR (queue);
   }  //=======================
```

This implementation of Iterators can be more efficient than the three kinds of implementations described in the preceding section. For one thing, you do not have to update parent pointers or threading information for each insertion or deletion of data. However, if users often progress only a little way through most iterations, you waste the time spent making a complete traversal to fill in the queue.

Yet another implementation of BintMap's iterator

An alternative for iterating in left-to-right inorder traversal order is to have each iterator keep track of all the nodes at which it made a left turn on the way down the tree to its current position. Those nodes, plus any nodes in the right subtrees of any of those "left-turn nodes", are all the nodes that it has not yet returned from the `next` method. If we keep the left-turn nodes on a stack, we can easily move back up the tree from one TreeNode to another. Listing 17.10 has the coding (see next page).

Listing 17.10 The StackMapIt class, providing iterators for BintMap

```
public java.util.Iterator iterator()    // member of BintMap
{  return new StackMapIt (this.itsRoot);
}  //====================


public class StackMapIt implements java.util.Iterator
{
   private StackADT itsStack = new NodeStack();


   public StackMapIt (TreeNode given)
   {  pushAllLefties (given, itsStack);
   }  //====================

   public boolean hasNext()
   {  return ! itsStack.isEmpty();
   }  //====================

   public Object next()
   {  TreeNode pos = (TreeNode) itsStack.pop();
      pushAllLefties (pos.getRight(), itsStack);
      return pos.getData();
   }  //====================

   public void remove()
   {  throw new UnsupportedOperationException ("no remove!");
   }  //====================

   /** push all nodes from the "left edge" of this tree. */

   private void pushAllLefties (TreeNode node, StackADT stack)
   {  while ( ! node.isEmpty())
      {  stack.push (node);
         node = node.getLeft();
      }
   }  //====================
}
```

The key concept for this **StackMapIt** class is that, at any given time, the `next` method has yet to return the data values (a) in any node in the stack or (b) in any node in the right subtree of any node in the stack. When the iterator is created, it pushes onto the stack all the non-empty nodes down the "left edge" of the tree. When `next` is called, it pops the top node (which, of all the nodes on the stack, is the node furthest down the tree). It then pushes onto the stack all the non-empty nodes down the "left edge" of the popped node's right subtree. Finally, it returns the data in the popped node.

Exercise 17.33 Describe the effect of replacing "Left" by "Right" and vice versa throughout Listing 17.10.
Exercise 17.34 Write the method `public void preorderTraverseRL (QueueADT queue)` analogous to the method in the lower part of Listing 17.9.
Exercise 17.35* Write the other three methods analogous to `inorderTraverseLR`.
Exercise 17.36* Rewrite the entire Listing 17.9 to use a NodeStack (methods `push` and `pop` instead of `enqueue` and `dequeue`) instead of a NodeQueue: Construction of an iterator puts all of the data values on a stack so that the `next` method produces left-to-right inorder traversal and has only one statement.
Exercise 17.37* How would you use the logic of Listing 17.3 to modify the constructor in Listing 17.9 to obtain an iterator that goes in breadth-first order?

17.7 Red-Black And AVL Binary Search Trees

One problem with binary search trees is that they may become unbalanced as values are put into them and taken out of them. We define a **full tree** to be a tree where every node with less than two subtrees is on the lowest level (and thus is a leaf with no subtrees). So the full tree with 3 nodes has 2 levels; the full tree with 7 nodes has 3 levels; and the full tree with 15 nodes has 4 levels. In general, a full tree with 2^k - 1 nodes has k levels. Figure 17.4 had an example of the full tree with 31 nodes. Figure 17.7 shows full trees with 1 node, 3 nodes, 7 nodes, and 15 nodes.

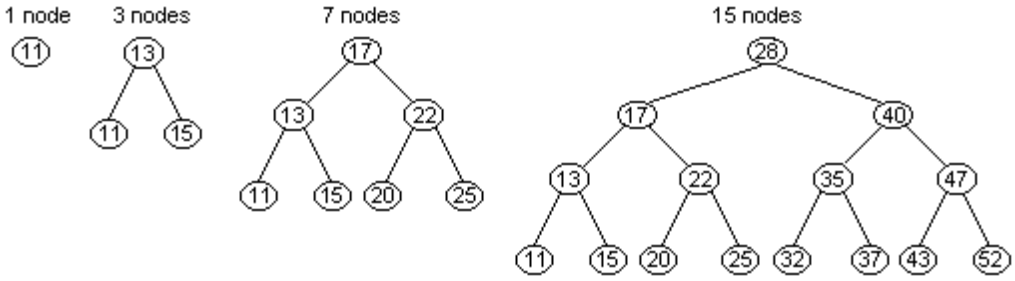


Figure 17.7 Four full trees

This book defines a tree to be **near-full** if it has the minimum possible number of levels for the number of data values in the tree. For instance, any tree with at least 4 data values has to have at least 3 levels; any tree with at least 8 data values has to have at least 4 levels; and any tree with at least 16 data values has to have at least 5 levels. In general, a tree with anywhere from 2^{k-1} data values up to and including 2^k-1 data values is near-full if it has only k levels.

In a near-full tree, the worst-case execution time to find a single value (execution time for `containsKey` or `get`) is big-oh of log(N). The reason is that the search process requires looking at only one value on each level of the binary tree. For example, since log2(32) is 5, if N is in the range from 16 to 31, log2(N) ranges from 4 to 5, and a near-full tree with N data values has 5 levels. So it takes at most 5 comparisons to find a key you are searching for. In general, you need at most log2(N+1) comparisons to find a data value in a near-full tree with N data values. By contrast, a badly-balanced tree could require up to N comparisons. Reminder: This book uses **log2(x)** to mean the number of times you have to halve x (i.e., execute `x /= 2.0`) to get 1 or less. In Java coding, log2(x) is the same as Math.ceil (Math.log(x) / Math.log(2)).

The **average search time** for a non-empty data structure is the total search time divided by the number of data values in the data structure. The total search time is the number of data values you have to look at before you find D, summed over all data values D in the data structure.

Example: The average search time for the first three full trees in Figure 17.7 is 1/1 for the first, 5/3 for the second, and 17/7 for the third. The 17 in the value 17/7 is the sum of 1 for the root node, 2 for each of the 2 second-level nodes, and 3 for each of the 4 third-level nodes: 1 + 2*2 + 3*4 = 17. A probabilistic analysis would show that, if you insert a random sequence of data values into a binary search tree, the average search time is 1.38 * log2(N) where N is the number of data values in the tree.

Balancing an existing binary search tree

If you have any Mapping object that contains mutually Comparable objects in ascending order, you can create a new BintMap object with the same data values in the same order but in a near-full tree. The BintMap constructor in Listing 17.11 (see next page) does this.

Listing 17.11 The balancing constructor in BintMap, with a method in TreeNode

```
/** Precondition:  par has MapEntries in ascending order. */

public BintMap (Mapping par)
{  itsRoot = (par == null || par.isEmpty())              //1
         ? TreeNode.ET                                   //2
         : TreeNode.balanced (par.size(), par.iterator());//3
}  //=====================

/** Return a near-full binary tree with the same ordering.
 *  Precondition:  it has at least numNodes >= 1 values.  */

public static TreeNode balanced (int numNodes, Iterator it)
{  if (numNodes == 1)                                    //4
      return new TreeNode (it.next());                   //5
   TreeNode leftSide = balanced (numNodes / 2, it);      //6
   TreeNode root = new TreeNode (it.next());             //7
   root.itsLeft = leftSide;                              //8
   int n = numNodes – 1 - numNodes / 2;                  //9
   root.itsRight = (n == 0) ?  ET  :  balanced (n, it);  //10
   return root;                                          //11
}  //=====================
```

If for instance the given Mapping object has 20 or 21 data values, the `makeBalanced` method iterates through the first 10 data values and constructs a near-full BintMap named `leftSide`. Then it makes a tree with the next (eleventh) data value in its root and `leftSide` as its left subtree. Finally, it iterates through the remaining 9 or 10 data values and puts them in a near-full BintMap as the right subtree of the root.

AVL trees

It would be best if each call of `put` or `remove` for a binary search tree would check the tree to be sure that the change it makes does not make the tree too far out of balance. If it would go too far out of balance, the method should make a small adjustment (on the order of big-oh of 1). This can be done, but it requires a somewhat looser concept of balance: This book defines a **decently-balanced binary tree** to be one with not more than twice as many levels as a near-full tree of the same number of nodes.

Some computer scientists named Adelson-Velskii and Landis figured out a way to do this, so the result is called an AVL tree. They add one more instance variable to the TreeNode class. This number is the number of levels in the left subtree of a node minus the number of levels in the right subtree of that same node:

```
private int leftMinusRight = 0;
```

The objective of the AVL algorithm is to keep this `leftMinusRight` number equal to 0, 1, or -1 for every node in the tree at all times. A binary search tree for which this property is true is called an **AVL tree**. So each time `put` or `remove` is called, it must see whether any node has had this value changed to an unacceptable value and, if so, make some adjustment to bring the tree back to having the AVL property.

When `put` is called for some AVL tree, it adds one node. It should be clear that the `leftMinusRight` number only changes for nodes that are on the path from the root down to the added node. If the addition of a node X loses the AVL property, consider the node P that is furthest down on the path from the root to X and has the wrong `leftMinusRight` number. That number will clearly be 2 if X is in the left subtree of P and will be -2 if X is in the right subtree of P.

To restore the AVL property, perform a **rotation** (which consists of moving a node from the side of P with more levels to the side with fewer levels). In this description and in Figure 17.8, C denotes the child of P and G denotes C's child on the side of C that has more levels (the "grandchild" G will be X or an ancestor of X):

1. Move node G to the other side of P (opposite from C) to be P's other child.
2. Swap the data in G, P, and C around to get them in increasing order left to right.
3. Attach the four subtrees of G, P, and C to those three nodes in the way that keeps their original order left to right.

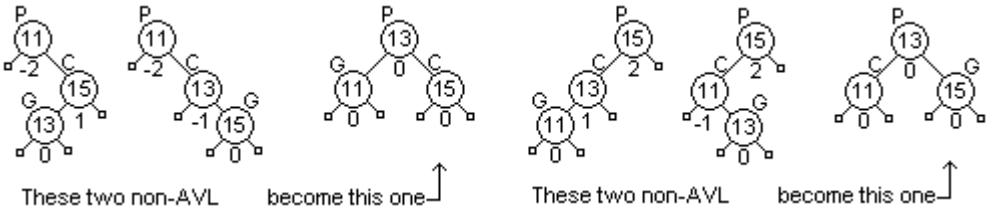


Figure 17.8 Rotations to correct an AVL tree, showing leftMinusRight values

The coding is left as a major programming problem. The next section proves logically that an AVL tree (also known as a **height-balanced tree**) is decently-balanced. The **height** of a binary tree is the number of nodes on the longest path from the root to a leaf. It is therefore the number of levels in the binary tree. But first we discuss another kind of tree, a red-black tree. We will also prove logically that a red-black tree is decently-balanced (so the worst-case execution time for a search is never more than twice as much as in a near-full tree with the same number of nodes).

Red-black trees

A **red-black tree** is a binary search tree in which each node is considered to be colored either red or black. We add a new instance variable to each node, as follows:

```
private boolean isRed = true;
```

This says that each node is created red by default; we change its color to black when needed (by executing `isRed = false`). A red-black tree must satisfy three properties:

1. The root node is black, and external nodes are considered to be black.
2. No red node is the child of a red node.
3. The number of black nodes on each path from the root to some external node is the same as on any other path from the root to some external node.

The first data value we put into a red-black tree is black, since it goes at the root. Each additional data value we put in goes in a red node X. There are three possible cases:

1. If X is the child of a black node, the tree maintains the three red-black properties.
2. If X is the child of a red node whose parent P has a black node on the other side, then perform a rotation as described previously for AVL trees.
3. If X is the child of a red node whose parent P has a red node on the other side, then color both children of P black. If P is not the root node of the whole tree, then color P red and apply this same logic (steps 1 through 3) to P in place of X.

Figure 17.9 illustrates what happens if you add one data value at a time to an AVL tree or a red-black tree, each value larger than the one before. When the 15 is added, it makes the tree unacceptable, which causes a rotation: The node containing 15 becomes the left child of the root node. Then the data values are moved around: 15 where 13 was, 13 in the root, and 11 to the left of the root.

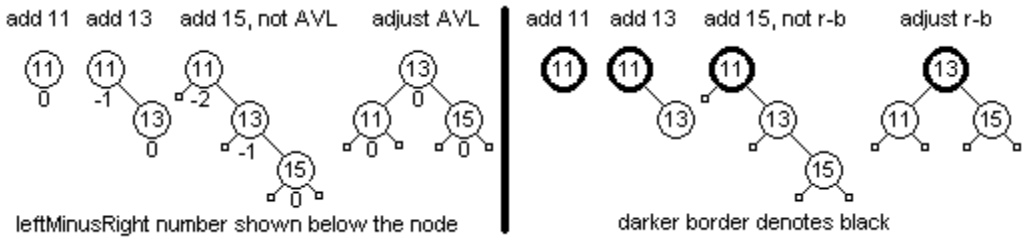


Figure 17.9 Adding ever larger values to an AVL tree and to a red-black tree

If you then added 17 to either of the balanced 3-node trees in Figure 17.9, it would be acceptable on the right of the 15-node in the AVL tree, but the red-black tree would require that you change the color on the 11-node and the 15-node to black.

Coding the red-black insertion algorithm

A partial coding of this algorithm for red-black trees is in Listing 17.12. The line numbers from the `put` logic in Listing 17.6 have been kept so that you can see that there are only two kinds of changes: (a) Lines 17, 20f, 23, and 26f return `ET` instead of null to signal the problem that the current node is red and has a red child; (b) Lines 20a-20e and lines 26a-26e call special private methods `fixLeftRedRed` and `fixRightRedRed` to correct the problem when the current node is a black node with a red child that itself has a red child. The `putRecursive` coding is the same as in Listing 17.6.

Listing 17.12 The put logic in the TreeNode class, revised for red-black trees

```
private Object putInLeftSubtree (Comparable id, Object val)
{  if (itsLeft.isEmpty())                                    //15
   {  itsLeft = new TreeNode (new MapEntry (id, val));       //16
      return this.isRed ? ET : null;                         //17'
   }                                                         //18
   else                                                      //19
   {  Object e = itsLeft.putRecursive (id, val);             //20'
      if ( ! this.isRed && e == ET)                          //20a
      {  fixLeftRedRed();                                    //20b
         return null;                                        //20c
      }                                                      //20d
      else                                                   //20e
         return this.isRed && itsLeft.isRed ? ET : e;        //20f
   }                                                         //20g
}  //=====================

private Object putInRightSubtree (Comparable id, Object val)
{  if (itsRight.isEmpty())                                   //21
   {  itsRight = new TreeNode (new MapEntry (id, val));      //22
      return this.isRed ? ET : null;                         //23'
   }                                                         //24
   else                                                      //25
   {  Object e = itsRight.putRecursive (id, val);            //26'
      if ( ! this.isRed && e == ET)                          //26a
      {  fixRightRedRed();                                   //26b
         return null;                                        //26c
      }                                                      //26d
      else                                                   //26e
         return this.isRed && itsRight.isRed ? ET : e;       //26f
   }                                                         //26g
}  //=====================
```

These two private methods are left as exercises. For instance, `fixLeftRedRed()` fixes the situation where the problem is on the left of the executor. The overall logic is:

```
if (itsLeft.itsLeft.isRed && itsLeft.itsRight.isRed)
   // make it red but make its two children black
else if (itsLeft.itsLeft.isRed)
   // rotate that grandchild to itsRight and readjust
else  // itsLeft.itsRight is red
   // rotate that other grandchild to itsRight and readjust
```

This requires one other change: The `put` method at the top of Listing 17.6 must make the root black again if it became red. This is an exercise too. The `remove` method for red-black trees is left as a major programming project.

When the executor of `putInLeftSubtree` is a red node and the key is not already in the tree, then that executor acts as follows: If its left subtree is empty, it creates a new red node to its left for the data and returns `ET`, which signals a problem. If its left subtree is not empty, it asks the black node Z on its left to insert the data and then, if Z turns red as a consequence, returns `ET` to signal a problem. Red nodes do not fix problems.

When the executor of `putInLeftSubtree` is a black node and the key is not already in the tree, then that executor acts as follows: If its left subtree is empty, it creates a new red node to its left for the data and returns null, which signals "no problem". If its left subtree is not empty, it asks the node Z on its left to insert the data; if Z returns `ET` (signaling a problem) it fixes the problem and returns null, otherwise it returns what Z returned. But fixing the problem may include turning itself red, which might cause a problem for its parent.

Exercise 17.38 Calculate the average search time for the 15-node full tree.
Exercise 17.39 Draw or precisely describe the trees of 4 or 5 nodes that `makeBalanced` produces.
Exercise 17.40 Draw or precisely describe the trees of 6 or 7 nodes that `makeBalanced` produces.
Exercise 17.41 (harder) Revise the `put` method in the upper part of Listing 17.6 to make the root node black after each value is added to the tree.
Exercise 17.42 (harder) If you put five data values into an AVL tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others?
Exercise 17.43 (harder) If you put five data values into a red-black tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others?
Exercise 17.44* If you put seven data values into an AVL tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others? Hint: Continue Exercise 17.42.
Exercise 17.45* If you put seven data values into a red-black tree, each larger than the one before, what does the tree look like? What does it look like if you then add one more data value larger than all the others? Hint: Continue Exercise 17.43.
Exercise 17.46* Write a recursive TreeNode method `public int height()`: The executor returns the height of its subtree; it returns 0 if empty.
Exercise 17.47* Write a recursive TreeNode method `public int howFar()`: The executor tells the shortest distance to an external node (it is 1 if either subtree is empty).
Exercise 17.48** Write the `fixLeftRedRed` and `fixRightRedRed` methods called by Listing 17.12 (they are identical except for swapping "Left" and "Right").
Exercise 17.49** Write the `fixLeftRedRed` and `fixRightRedRed` methods called by Listing 17.12, but instead of actually moving the grandchild node, create a new node on the other side of P from C and later discard the grandchild node. This takes fewer statements than the method described in the text. Is it more efficient?

17.8 Inductive Reasoning About Binary Trees

The definition of a binary tree is recursive, so most of the logic for binary trees is also recursive. Let us review the definition of a binary tree (parts 1 and 2 are the reason for having ET as the external node instead of using null to represent an external node):

1. There are two kinds of binary trees: empty ones and non-empty ones.
2. An empty binary tree has no data and no subtrees.
3. A non-empty binary tree has one data value and two subtrees (which are are also binary trees), called its left and right subtrees.

Each binary tree corresponds in a fundamental way to its root node, which is external if the tree is empty and internal if the tree is not empty.

An example of recursive logic for binary trees is the proof that, **in every binary tree X, the number of external nodes is one more than the number of internal nodes**:

1. Basis Step If X is empty, then X has one external node (its root node) and zero internal nodes, so the difference is 1 as the assertion says.
2. Inductive Step Say X is some non-empty tree and say we have already proven the assertion for every tree with fewer nodes than X. Then each of X's two subtrees has 1 more external node than internal nodes, since each has fewer nodes than X. So together X's two subtrees have 2 more external nodes than internal nodes. Since the root node is internal, that brings the difference for X down to 1 more external node than internal nodes.
3. Conclusion The truth of the assertion for all binary trees follows by the Induction Principle from the Basis Step and the Inductive Step.

When some people first see this kind of logic, they think that it is circular logic, since we prove something about a non-empty tree X by assuming we have already proven it about each of its subtrees. But this is not circular logic, it is **inductive logic**, because each of the subtrees of X has fewer data values than X has. The Basic Step of the proof shows that the assertion is true for trees with 0 data values (empty trees). The Inductive Step of the proof shows that the assertion is true for any tree X with a positive number of data values as long as the assertion is true for all trees that have fewer data values than X has. **So for any n > 0, the Inductive Step deduces that the assertion is true for any n-node tree from the assertion being true for all trees with less than n nodes.**

How do you know that the assertion is true for any binary tree X? Because it could be proven for each number of data values 0, 1, 2, 3, 4, 5, etc. in that order:

- If X has 0 data values, the Basic Step shows the assertion is true for X.
- If X has 1 data value, then its subtrees both have 0 data values, so the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 2 data values, then each of its subtrees has at most 1 data value, so the two preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 3 data values, then each of its subtrees has at most 2 data values, so the three preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 4 data values, then each of its subtrees has at most 3 data values, so the four preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- If X has 5 data values, then each of its subtrees has at most 4 data values, so the five preceding points show that the assertion is true for each of its subtrees, so the Inductive Step shows the assertion is true for X.
- etc.

Minimum size of a red-black tree

We define the **black-height** of a red-black tree to be the number of black nodes on each path from the root to an external node. Figure 17.10 shows a red-black tree with a black-height of 3. We can prove that **every non-empty red-black tree X with black-height k has at least 2^k-1 internal nodes** using induction on k as follows:

1. Basis Step If X has black-height 1, it has at least 1 node, and 1 is at least 2^1-1.
2. Inductive Step Say X has black-height h+1 where h is 1 or more, and say we have already proven the assertion for each red-black tree with black-height h. We could then prove the assertion for X as follows: The tree has a black root node R with two children. A black child of R is the root of a red-black tree with black-height h, and a red child of R contains two subtrees that are red-black trees with black-height h. So the whole tree contains at least two separate subtrees each with at least 2^h-1 nodes, besides the root itself, and therefore has at least 2*(2^h-1)+1 nodes, which is 2^{h+1}-1.
3. Conclusion The truth of the assertion for all non-empty red-black trees follows by the Induction Principle from the Basis Step and the Inductive Step.

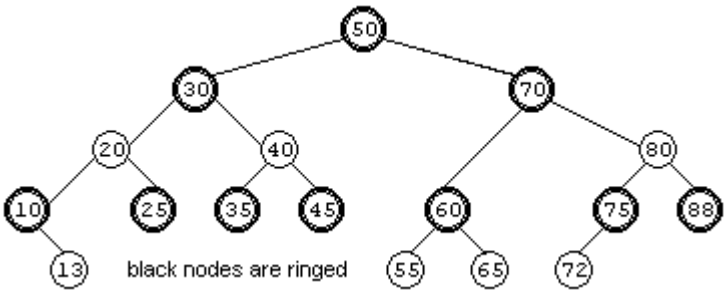


Figure 17.10 A red-black tree with black-height 3 and 5 total levels.

Worst-case execution time for a search of a red-black tree

Say you have a red-black tree with a positive number of nodes N. Let k denote the black-height of the tree. Then the preceding proof showed that N is 2^k-1 or more, so log2(N+1) is k when N is 2^k-1 and is at least k+1 otherwise. This tree with black-height k has at most 2*k levels, since you cannot have two red nodes in a row. And in the special case when N is exactly 2^k-1, the tree has only k levels. Now 2*log2(N+1)-1 is 2*k-1 in the special case (when the number of levels is k) and is at least 2*k+1 otherwise (when the number of levels cannot be more than 2*k). Conclusion: The number of levels, and therefore the worst-case time for a search in a red-black tree, is at most 2*log2(N+1) - 1. For instance, if N is 16 to 31, then log2(N+1) is 5 and so the worst-case search time is 2 * 5 - 1 = 9; the red-black tree cannot have 10 levels.

Worst-case execution time for a search of an AVL tree

We define **size(h)** to be the minimum possible number of data values in an AVL tree with h levels. An AVL tree with 1 level has 1 data value, so size(1) is 1. An AVL tree with 2 levels must have either 2 or 3 data values, so size(2) = 2 (the minimum).

If an AVL tree has h levels and h is more than 2, then one of its root's two subtrees must logically have h-1 levels, though the other subtree could have either h-1 or h-2 levels (not less, otherwise it would not have the AVL balance property). So the minimum possible number of data values in the subtree with level h-1 is size(h-1) and the minimum possible in the other subtree is size(h-2). Therefore, the minimum possible number of data values in the whole AVL tree is 1 + size(h-1) + size(h-2). Clearly, size(h-1) is larger than size(h-2). So it follows that size(h) is larger than 1 + 2 * size(h-2). This fact can be used to prove inductively that **size(2*k) >= 2^k for every positive k**:

1. Basic Step size(2) = 2 implies that size(2*1) >= 2^1, so the assertion is true for k==1.
2. Inductive Step To show that size(2*k) >= 2^k in situations where k > 1 and we know that the formula is true for k-1, i.e., that size(2*(k-1)) >= 2^{k-1}, we reason as follows: size(2*k) > 1 + 2 * size((2*k)-2) = 1 + 2 * size(2*(k-1)) > 2 * 2^{k-1} = 2^k.
3. Conclusion The truth of the assertion for all positive integers k follows by the Induction Principle from the Basis Step and the Inductive Step. For instance:

- The Basic Step showed that size(2*1) >= 2^1.
- size(h) > 2 * size(h-2) implies that size(4) >= 2 * size(2), i.e., size(2*2) >= 2^2.
- size(h) > 2 * size(h-2) implies that size(6) >= 2 * size(4), i.e., size(2*3) >= 2^3.
- size(h) > 2 * size(h-2) implies that size(8) >= 2 * size(6), i.e., size(2*4) >= 2^4, etc.

To find the longest path from the root to an external node in an AVL tree with N data values, define k to be log2(N+1), so that N < 2^k. Since we have proven that size(2*k) is at least 2^k, the AVL tree must have less than 2*k levels, thus less than 2*log(N+1) levels. So the worst-case search time in an AVL tree is 2*log2(N+1) - 1. For instance, if N is 16 to 31, then log2(N+1) is 5 and so the worst-case search time is 2 * 5 - 1 = 9; the tree cannot have 10 levels. This is the same result as for red-black trees.

Exercise 17.50 Calculate size(3) and size(4). Justify your answer.
Exercise 17.51 (harder) There is 1 binary tree with 1 data value; there are 2 binary trees with 2 data values and 5 binary trees with 3 data values. Calculate the number of binary trees with 4 data values and the number with 5 data values. Justify your answer.
Exercise 17.52 (harder) There are 4 red-black trees with a black-height of 1. Calculate the number of red-black trees with a black-height of 2. Justify your answer.
Exercise 17.53* Define rbsize(h) to be the minimum possible number of data values in a red-black tree with h levels, so rbsize(2) is 2 and rbsize(4) is 6. Find rbsize(6), rbsize(8), and rbsize(10). Find a formula for rbsize(2*n). Also calculate size(4), size(6), size(8), and size(10). Which kind of tree can be more unbalanced for a given number of levels, red-black or AVL?
Exercise 17.54* Prove by induction on the number of nodes that left-to-right inorder traversal of a binary search tree produces the data values in ascending order (each less than or equal to the next).
Exercise 17.55* Prove by induction on the number of nodes that size(2*k) >= 2^{k+1} - 2 for k >= 1.
Exercise 17.56* Prove by induction on k that a full tree with k levels has 2^k - 1 nodes.
Exercise 17.57** Prove by induction on h that when h>= 7, size(h) <= 1.66*size(h-1) and size(h) >= 1.60*size(h-1). Can you discover its relation to the golden mean 1.61803...?
Exercise 17.58*** Write a method that calculates the number of binary trees with n data values for any given positive int n. Hint: See Exercise 17.51. Store intermediate results in an array (this use of an array, here and in the next exercise, is called dynamic programming).
Exercise 17.59*** Write a method that calculates the number of red-black trees with a black-height of n for any given positive int n. Hint: See Exercise 17.52. Store intermediate results in an array.

17.9 2-3-4 Trees And B-Trees

In a drawing of a red-black tree, draw a big circle around each internal black node so that the big circle includes its red subtrees (of which there are either 0 or 1 or 2). So each big circle has either 2 or 3 or 4 subtrees pointing down from it. Define a class of objects that represent these big circles in an implementation of Mapping: up to 3 data values and up to 4 subtrees. You could define such an object class as follows:

```
public class BigCircle
{  private final int MAX = 4; // maximum number of subtrees
   private MapEntry [] itsData = new MapEntry[MAX - 1];
   private BigCircle[] itsSubtree = new BigCircle[MAX];
}
```

We store the 1 or 2 or 3 data values of one big circle in increasing order in the MapEntry array, leaving the unused components equal to null. That means:

- If a black node X has 0 red subtrees, fill in only `itsData[0]` with the black node's data. Make `itsSubtree[0]` and `itsSubtree[1]` refer to the two subtrees in order left to right, except use null if the subtree is empty.
- If the black node X has a black left subtree and a red right subtree, fill in `itsData[0]` with X's data and `itsData[1]` with the red node's data. Make `itsSubtree[0]` the left subtree of X, `itsSubtree[1]` the left subtree of the red node, and `itsSubtree[2]` the right subtree of the red node.
- If the black node X has a red left subtree and a black right subtree, fill in `itsData[0]` with the red node's data and `itsData[1]` with X's data. Make `itsSubtree[0]` the left subtree of the red node, `itsSubtree[1]` the right subtree of the red node, and `itsSubtree[2]` the right subtree of X.
- If the black node X has two red subtrees, fill in `itsData[0]` with the left red node's data and `itsData[1]` with X's data and `itsData[2]` with the right red node's data. Make `itsSubtree` contain the four subtrees of the red nodes in the proper order.

Now we write coding to do exactly what the red-black tree does to implement a Mapping, except we use the array representations of BigCircles instead of the left-right representations of nodes. Each of the BigCircle objects contains either 2 or 3 or 4 subtrees, so this implementation of a Mapping is called a **2-3-4 tree**. Note that it is not a binary tree, because a BigCircle does not have just two subtrees and because it has more than one data value. Figure 17.11 contains an example of a 2-3-4 tree with three levels, the exact analog of the red-black tree in the earlier Figure 17.10.

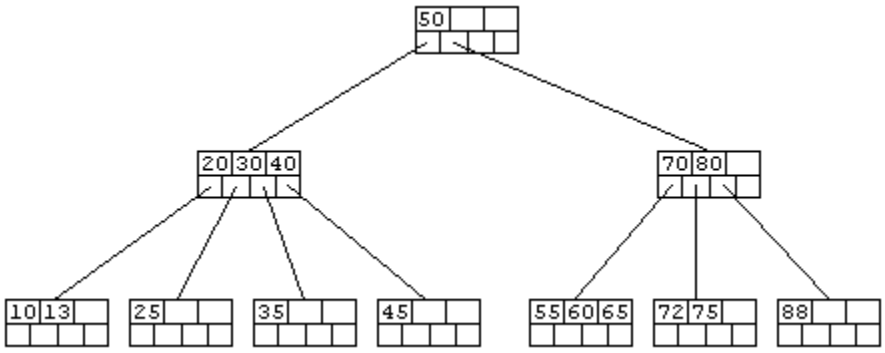


Figure 17.11 A 2-3-4 tree equivalent to the red-black tree of Figure 17.10

The get method for 2-3-4 trees

The `get` Mapping method is to search through the data structure for the MapEntry containing a given `id` and return the corresponding value. It is to return null if the `id` is not in the data structure. The coding of that `get` method for the BigCircle implementation could be as shown in Listing 17.13. The completion of this BigCircle class is left as a major programming problem.

Listing 17.13 The get method for the BigCircle class of 2-3-4 trees

```
public class BigCircle implements Mapping
{
   private final int MAX = 4;  // maximum number of subtrees
   private MapEntry[] itsData = new MapEntry[MAX - 1];
   private BigCircle[] itsSubtree = new BigCircle[MAX];

   /** Return the value for this id, or null if not there.
    *  Throw a RuntimeException if id is not Comparable. */

   public Object get (Object id)
   {  int k = 0;
      for ( ;  k < MAX - 1 && itsData[k] != null;  k++)
      {  int comp = ((Comparable) id).compareTo
                                 (itsData[k].getKey());
         if (comp <= 0)
            return (comp == 0)  ?  itsData[k].getValue()
                   : (itsSubtree[k] == null)  ?  null // a leaf
                     : itsSubtree[k].get (id);
      }
      return (itsSubtree[k] == null)  ?  null        // a leaf
             : itsSubtree[k].get (id);
   }  //====================
}
```

Another data structure sometimes used for a Map is a **2-3 tree**, which is halfway between a binary search tree and a 2-3-4 tree: Each node has 1 data value and 2 subtrees or else 2 data values and 3 subtrees, with the same ordering properties as for 2-3-4 trees, and all leaves are on the same level.

B-trees

When an operating system reads information from a data file, it normally gets the information in chunks of a particular number of bytes, usually a power of two. For instance, an operating system may read in chunks of 4096 bytes. Even if you only want a few hundred bytes of information, each read operation gets 4096 bytes.

When you are reading information for a Mapping object from a file, it is most efficient if you can use most or all of the 4096 bytes you will get from each read operation. To this end, you could have BigCircle objects with lots more than 4 subtrees. Make them large enough that they will barely fit within 4096 bytes.

Say you know that each record (one object's information) is 190 bytes long. A reference to a subtree takes up 4 bytes of space, for a total of 194 bytes. 194 divided into 4096 is 21. So you could have as many as 21 data values in each BigCircle object, with the data values in increasing order of key values. You would thus have up to 22 subtrees. Then the `get` method for these really big circles could be the same as the one in Listing 17.13, except that MAX would be 22 instead of 4. Even better, you could use binary search to find the desired index.

This data structure is called a B-tree of order 22. In general, a **B-tree of order N** has up to N subtrees in each node for some N, always with one less data value than it has subtrees. A B-tree has two additional restrictions: All leaf nodes must be on the bottom level, and all nodes except the root must have at least N/2 subtrees (rounded up if N is odd). A 2-3-4 tree is just a B-tree of order 4.

When you add a new data value to a B-tree structure, you put it in the appropriate leaf node on the bottom level. There will only be one such place it can go in, according to the restrictions on ordering. However, if the node where it goes is full (already has N-1 data values), there is no room for the additional data value. In that case, you split the full leaf node into two nodes, with at least N/2 data values in the first node, the next (middle) data value moved up into the parent node, and the rest of them in the second node. Both of these nodes become children of the parent node. Of course, if that makes the parent node full, you have to split again, possibly going as far as splitting the root node.

A B-tree of order 22 that organizes a data file with 400 million records (more than the population of the United States) would have at most nine levels: It would have at least 1 data value in the root, therefore at least 2 nodes on the second level with at least 11 subtrees of each. The third level would then have at least 22 nodes, so the fourth level would have at least 22*11 nodes, etc. Each level multiples by 11, so the ninth level would have at least $22*11^6$ nodes, about 39 million, with at least 10 data values in each of those leaf nodes. Since you would always keep the root node in memory, you can find a data value in only 8 disk accesses, instead of the 29 that even a near-full binary tree would require.

Indexing

A popular alternative is to use B-trees to hold an index to a random-access data file instead of the data itself. Only the keys are stored in the B-tree; the full data record is only in a disk file. Each leaf node contains the key for one record plus the file position telling where the full data record is in the data file. All keys are in the leaf nodes. In each non-leaf node, `itsData[k]` is the first key of `itsSubtree[k+1]`.

Example If keys are 10 bytes (e.g., Social Security Numbers), and file pointer values are 4 bytes, then you only need 14 bytes per record. Instead of the B-tree of order 22 just described, you would have room for 4096 / 14 = 292 subtrees in each "big circle" node. So you could use a B-tree of order 292 for the index. So six levels would store a minimum of 292 * 146 * 146 * 146 > 900 million keys and file pointers on the sixth level. Even allowing for the extra disk access to retrieve the record itself, it would take only six disk accesses to find a data record in the random-access file, assuming you keep the root node in memory at all times.

Exercise 17.60 How many 296-byte records could you store in a B-tree node if it is to barely fit into a disk block of 8192 bytes? Show why.
Exercise 17.61 (harder) How many disk accesses (at most) would it take to find a single data value in a B-tree of order 100 used to store ten million records? Show why.
Exercise 17.62* How many disk accesses (at most) would it take to find a single data value in a B-tree of order 200 used to store 60,000 records? Show why.
Exercise 17.63* Rewrite the BigCircle `get` method in Listing 17.13 to first look at the middle one of three data values in the BigCircle. Does this execute faster?

17.10 Data Flow Diagrams

A small manufacturing company wants to hire you to create software to automate some of their operations. A scenario of part of what this company does could go as follows:

A customer places an order. You verify that the information about the customer (address, references, preferences, etc.) is already in your files. You check with your accounting department that the customer's credit is good. You record the order, manufacture the product wanted, and ship it to the customer along with an invoice. The customer sends you a check in payment, which you use to pay investors' dividends.

The preceding paragraph is called a **use case**; it is a scenario that describes a possible sequence of events using the system you are to design. If you are to model the entire business in software, then the business is the system and everything outside the system is its **environment**. Figure 17.12 is a Data Flow Diagram that graphically shows the elements mentioned in this particular use case.

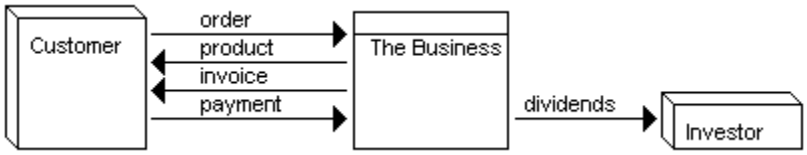


Figure 17.12 Data Flow Diagram for one use case

A **Data Flow Diagram** (**DFD**) has four basic kinds of elements. The first three in this list called the **nodes** of the diagram:

1. A **process**, which is part of the system to be modeled, shown as a rectangle divided horizontally. You put the name and/or description of the system part in the lower section. The middle node of Figure 17.12 is a process.
2. An **environment entity** that interacts with the system, shown as a 3D-box-like figure. You put the name of the entity on the front of the box. The nodes on the left and right of Figure 17.12 are environment entities.
3. A **data store**, which is part of the system to be modeled, shown as a rectangle divided vertically and with the right side containing its name: It stores a collection of information of a particular kind. No data stores appear in Figure 17.12.
4. A **data flow**, shown as an arrow between two nodes, of which one is almost always a process. A data flow indicates information flowing from one node to another. Its name is above or beside the arrow. Figure 17.12 has five data flows.

Two additional use cases for the manufacturing company are as follows:

You check the amount you have on hand of various parts you use to manufacture your product. You see that you are short on one kind of part. You place an order with a supplier. The supplier sends you boxes of parts and an invoice. You send a check to pay the invoice.

Investors contribute more capital to the operation, for which you issue stock. Every three months, you calculate your profits. Based on these calculations, you send a tax payment to the IRS and also dividend checks and a quarterly report to your investors.

When you look over these and other use cases, including entering a description of the customer (address, etc.) in your files, you might come up with the Data Flow Diagram in Figure 17.13 (see next page) to describe the major interactions of the company with its environment.

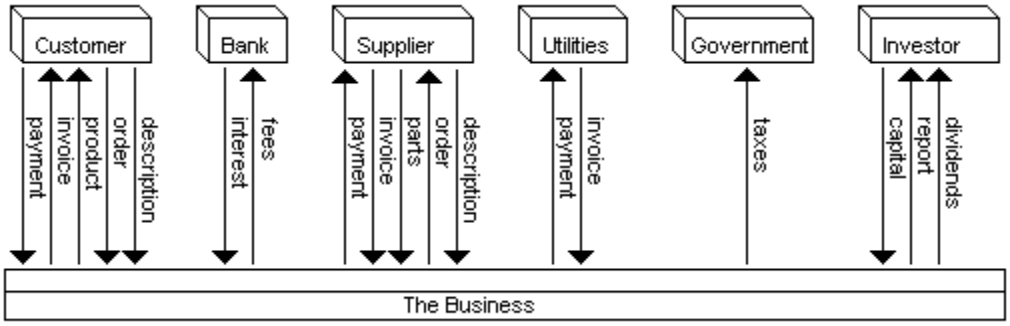


Figure 17.13 Data Flow Diagram for the entire business

The OrderHandling system

Your client wants you to implement in software only the part of the operation that handles orders from customers and orders to suppliers. So you can ignore some of the data flows described so far, such as dividends and payment of utility bills and taxes. Start with some examples of use cases for this OrderHandling system:

Use Case 1 A person wants to order products from your company. You look up the name in your records and see that the customer is not recorded there. So you get the full description of the customer (name, address, references, etc.) and store it in your data store of customers. You then take the order and store it in your data store of orders. You check your inventory and see that the product is there. So you ship the product to the customer along with an invoice for payment, and at the same time notify the accounting department to expect payment since you have sent an invoice.

Use Case 2 A person places an order. You look up the name in your records and retrieve the customer's information from your customer data store (since it was previously recorded). You then take the order and store it in your data store of orders. You check inventory and see that the product is not there. So you order it from the appropriate supplier (which in many cases will be a manufacturing department in the rest of the business, but treat it as a supplier for now).

Use Case 3 A supplier ships parts to you along with an invoice. You store it in your inventory and notify the accounting department to send payment to the supplier.

Use Case 4 A manufacturing department sends you finished product. You store it in your inventory.

Use Case 5 You check your data store of orders not yet filled and find one for which you now have the required product. So you ship the product to the customer along with an invoice for payment, and at the same time notify the accounting department to expect payment since you have sent an invoice.

Further thought about the software and discussion with the client reveals that the customer will sometimes ask for the status of all of the customer's orders currently being processed, to verify its own records. In addition, the rest of the business operation may query the system from time to time for a list of all orders pending, of all orders filled in the past year, of inventory changes during the year, etc.

A very effective first step in the development of a software system is to describe its interactions with its environment, that is, with everything that is not the system to be developed. From this point of view, the rest of the business is part of the environment. These interactions usually take the form of input to the system from external entities and output from the system to external entities, both of which can be expressed as data flows.

Figure 17.14 is the only part of the overall operation of the company that concerns the OrdereHandling system. Note that the system is split off from the rest of the business in this Data Flow Diagram. The names of the data flows have been made more precise: CustDescr is a description of one customer, SuppOrder is one order sent to a supplier, etc. The reporting functions have been added to the Data Flow Diagram. Minor "backflows" along certain arrows have been omitted, such as a confirmation message to the customer that the description or order has been received and a query for a report.

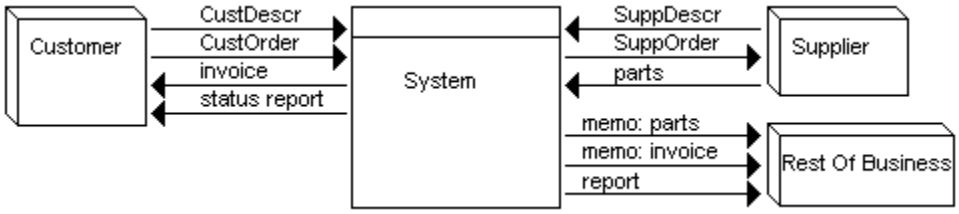


Figure 17.14 Data Flow Diagram for the OrderHandling software

Detailed Analysis

The next step in the analysis is to specify the exact form of the inputs and the outputs, the conditions under which they will occur, the sequence in which they occur, etc. This information fleshes out the DFD by saying what happens. Avoid saying how it happens; that is almost always a matter of design, so it comes later. Treat the system as a "black box" whose contents and workings are a mystery.

Analysis for this problem requires a full and precise description of the number and types of values entered, e.g., first and last name are strings of characters, the ZIP code is entered separately from the city and state, and the number of items ordered is a positive integer. It also requires a full and precise description of the form of the reports. This **Requirements Specification Document** should include examples and specifications for these things along with the Data Flow Diagram. Test data should also be developed at this point. We will not try to do it here for this problem.

Design

Now you start to divide up the proposed system into components. Design is a matter of deciding which components you want to have and how they will interact. For this OrderHandling software, some obvious choices are classes of CustomerOrder objects, SupplierOrder objects, Invoice objects, CustomerDescription objects, and SupplierDescription objects.

You also need a data structure for long-term storage of the list of CustomerDescription objects for all current customers, the list of CustomerOrder objects still pending, and the analogous lists of SupplierDescription and SupplierOrder objects. Reports are presumably simply long Strings of characters, and you already have a String object class.

The two main processes of the system are managing the customer relations and managing the supplier relations. These considerations lead to the Data Flow Diagram shown in Figure 17.15 (see next page). The abbreviations CustOrder, SuppOrder, CustDescr, and SuppDescr make the diagram more compact.

Number the process nodes 1, 2, 3, on up in whatever order you like, in the upper section of the node. Number data store nodes D1, D2, D3, on up also. We often append DS to the name to suggest a Data Store (also known as a Data Structure). Use these numbers for cross-referencing with more detailed Data Flow Diagrams and with narrative specifications.

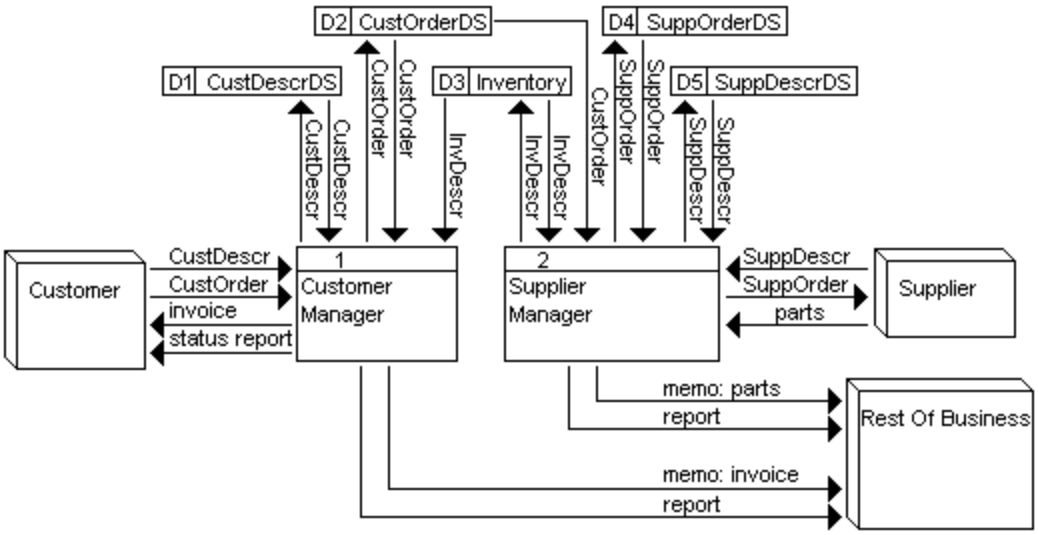


Figure 17.15 Data Flow Diagram for second level of OrderHandling

The middle and upper parts of Figure 17.15 are an **explosion** of the System node in the earlier Figure 17.14, since they are a more detailed replacement for that System node. You need to verify that all the data flows between the System node and its environment in Figure 17.14 are in the exploded view, and no others have been added. A close check shows that the ten data flows of the earlier figure are present in the later one, although "report" occurs twice in the later one (since the two separate processes have separate kinds of reports).

CRC cards

A useful device for a preliminary specification is a Class/Responsibilities/Collaborations card, a **CRC** card for short. You use an index card (no bigger than 4-by-6) on which you write the name of the class of objects at the top. Then you list on the left side of the card the major responsibilities of that class -- what messages it can be sent and what action it should take in response. You list on the right side of the card the collaborations of that class -- which other classes it sends messages to or otherwise uses.

The CustomerManager kind of object (process #1) has four major responsibilities:

1. Given a CustomerOrder object containing a CustomerID, see whether the CustomerDescription object is in the CustDescrDS. If not, create one from from data the customer provides and store it in the CustDescrDS. Then store the CustomerOrder object in the CustOrderDS.
2. Search the CustOrderDS for orders not yet filled. For each one, check whether all products ordered are currently in the Inventory. If so, create an Invoice object, send it to the customer along with the product ordered, and send a memo of the Invoice to the accounts-receivable department.
3. Given a request from the customer for a status report, create and return it.
4. Given a request from the rest of the business for any one of several kinds of summary reports, create and return it.

This process #1 has therefore collaborations with three data stores (CustDescrDS, CustOrderDS, and Inventory), with three passive data flows (CustomerOrder, CustomerDescription, and InventoryDescription objects) and with the rest of the business. Now stubbed documentation for the CustomerManager class of objects can be developed as shown in Listing 17.14.

Listing 17.14 CustomerManager class of objects, stubbed documentation

```
public class CustomerManager           // stubbed
{
   /** Verify that the customer's description is on file.
    *  If not, obtain and file it.  Then file the order.  */

   public void storeOrder (CustomerOrder order)              { }


   /** Find all orders that are currently unfilled.
    *  Fill any such order for which all product is available. */

   public void fillAllPossibleOrders()                       { }


   /** Return a description of all orders from this customer
    *  that are currently unfilled. */

   public String getStatus (String customerID)   { return null; }


   /** Return a description of all orders currently unfilled,
    *  in order of customer ID. */

   public String getOrdersByID()                 { return null; }


   /** Return a description of all orders currently unfilled,
    *  in order of dates entered. */

   public String getOrdersByDate()               { return null; }


   // many other report functions would go here
}
```

You develop CRC cards by walking through the scenarios given by use cases and seeing which classes and operations a software system should have. A large number of people find that developing CRC cards is a far better way to develop an object design than data flow diagrams, since they have a concrete object (the 4-by-6 card) to help them anchor their thoughts. Other people prefer developing data flow diagrams. You need to attend a 2- or 3-hour workshop on the use of CRC cards in order to truly appreciate them. Reading about them would not adequately convey their advantages.

Exercise 17.64* Write out two more use cases for the OrderHandling software.

17.11 Review Of Chapter Seventeen

- A **general tree** has one or more data values and zero or more subtrees.
- A **binary tree** is either (a) empty, in which case it has no data and no subtrees, or (b) non-empty, in which case it has one data value and two subtrees. The data value is called its **root value**. The two subtrees are called its **left and right subtrees**.
- An **internal node** represents a non-empty binary tree and has a minimum of three attributes (instance variables): a data value stored there plus references to the root nodes of the left subtree and the right subtree. If both of the subtrees of an internal node are empty, it is called a **leaf node**. An **external node** has no data value and no subtrees; it represents the empty binary tree.
- A **path** in a tree from one node to another is a sequence of nodes beginning with the first and ending with the second such that, for each time node Y follows node X in the sequence, node Y is the root of a subtree of node X. A binary tree cannot have two different paths from one node to another, nor any path from a node to itself.
- **Breadth-first search** of a binary tree means that you first look at the data at the root, then at the data one step from the root, then at the data two steps from the root, etc. **Depth-first search** means that, after looking at the data in a particular node, you look at all the data in the subtree it is the root of before you look elsewhere.
- **Left-to-right preorder traversal** is a depth-first search that, after it looks at a node X, looks at all the nodes in the left subtree of X, then looks at all the nodes in the right subtree of X. "Traversal" means you visit each node in the tree (potentially). "Left-to-right" means that you visit all nodes in the left subtree before you visit any node in the right subtree; the alternative is right-to-left traversal. "Preorder" means that you visit the root node of each subtree before visiting any of the nodes in its subtrees. The alternatives to preorder traversal are **postorder traversal** (visit the root node after visiting all the nodes of the subtrees) and **inorder traversal** (visit the root node in between visiting the nodes of the subtrees).
- In a **binary search tree**, the Comparable keys have a special relationship: Every key in a left subtree is less than the key for the root and every key in a right subtree is greater than the key for the root. Some applications (not Mappings) allow several data values that are equal or the equivalent. In that case, a data value can be equal to another one in its right subtree.
- If a node X represents a non-empty binary tree, the nodes at the roots of the subtrees of X are called the **left child** and the **right child** of X. X is their **parent**.
- A tree is full if every node with less than two subtrees is on the lowest level. So the **full tree** with 3 nodes has 2 levels; the full tree with 7 nodes has 3 levels; and the full tree with 15 nodes has 4 levels. In general, a full tree with 2^k - 1 nodes has k levels. This book defines a tree to be **near-full** if it has the minimum possible number of levels for the number of data values in the tree. A binary tree can be considered to be **decently-balanced** if it does not have more than twice as many levels as a near-full tree of the same number of nodes.
- The **average search time** for a non-empty data structure is the total search time divided by the number of data values in the data structure. The total search time is the number of data values you have to look in order to find D, summed over all data values D in the data structure.
- An **AVL tree** is a binary search tree in which, for any node X, the number of levels in the right subtree of X differs from the number of levels in the left subtree of X by at most 1. An AVL tree is always decently-balanced.
- A **red-black tree** is a binary search tree in which the root node is black, no red node is the child of a red node, and the number of black nodes on each path from the root to some external node is the same as on any other path from the root to some external node. A red-black tree is always decently-balanced.
- A **B-tree of order N** has up to N subtrees in each node for some fixed N, always with one less data value than it has subtrees. A B-tree has two additional restrictions: All leaf nodes must be on the bottom level, and all nodes except the root and the leaves must have a minimum of 0.5*N subtrees. A **2-3-4 tree** is just a B-tree of order 4.

Answers to Selected Exercises

17.3 The fifth tree: A, B, C. The sixth tree: C, B, A. The seventh tree: A, B, C.

17.4 c.size() gets 0 from its call of c.itsLeft.size() and get 4 from its call of d.size(), so it returns 5.

17.5 b.deleteLastNode() would see that h.itsRight is not empty, so it would call h.deleteLastNode(), which would see that i.itsRight is not empty, so it would call i.deleteLastNode(), which would see that k.itsRight is empty, so it would set i.itsRight to be the empty tree.

17.6
```
public TreeNode lastNode()
{   return itsRight.isEmpty()  ?  this  :  itsRight.lastNode();
}
```

17.7 It is the same as deleteLastNode except interchange the words "Right" and "Left".

17.8 printInOrder executes in O(N) time, since it visits every node in the tree. deleteLastNode executes for the average binary tree in O(log(N)) time, since it navigates only one path down the tree.

17.11 A, C, G, F, B, E, D. A's left is C, C's left is G, A's right is B.

17.12 A, E, B, G, F, D, C. A's left is B, B's left is C, A's right is E.

17.13 The tree could have 7 levels with 1 node on each level. So no node has two children.

17.14 The level of a data value is equal to the difference between the number of left parentheses that come before it and the number of right parentheses that come before it. In other words, read the output from left to right, counting +1 for each left parenthesis and -1 for each right parenthesis, and when you come to a data value, the net count so far is the level of that data value.

17.17
```
public TreeNode lookUp (Comparable id)
{   int comp = compare (id, itsData);
    if (comp < 0)
        return itsLeft.isEmpty()  ?  null  :  itsLeft.lookUp (id);
    if (comp > 0)
        return itsRight.isEmpty()  ?  null  :  itsRight.lookUp (id);
    return this;
}
```

17.18 Insert the following just before the "else if" in line 14:
```
else if (itsLeft.isEmpty())
{   itsData = itsRight.itsData;
    itsLeft = itsRight.itsLeft;    // not the opposite order
    itsRight = itsRight.itsRight;
}
```
This executes faster on average only if there are a significant number of nodes in the tree with an empty left subtree and a right subtree of 2 or more levels. So it is slower if the tree is reasonably balanced, but faster if the tree is quite unbalanced.

17.19
```
public TreeNode copy()
{   TreeNode root = new TreeNode (itsData);
    root.itsLeft = itsLeft.isEmpty()  ?  ET  :  itsLeft.copy();
    root.itsRight = itsRight.isEmpty()  ?  ET  :  itsRight.copy();
    return root;
}
```

17.20
```
public int under (Comparable id)
{   return this.isEmpty()  ?  0
            :  compare (id, itsData) <= 0  ?  itsLeft.under (id)
                    :  itsLeft.size() + 1 + itsRight.under (id);
}
```

17.26
```
if (toDelete.itsParent.itsLeft == toDelete)
    toDelete.itsParent.itsLeft = toDelete.itsLeft;
else
    toDelete.itsParent.itsRight = toDelete.itsLeft;
toDelete.itsLeft.itsParent = toDelete.itsParent;
```

17.27
```
private TreeNode lastLeft (TreeNode lastLeftTurn, Comparable id)
{   int comp = compare (id, itsData);
    return (comp == 0)  ?  lastLeftTurn
            : (comp > 0)  ?  itsRight.lastLeft (lastLeftTurn, id)
                    : itsLeft.lastLeft (this, id);
}
```

17.28 In a well-balanced tree with N values, a node X with no right subtree should only be in one of the two bottom levels. The successor of X will tend to be roughly two levels above X, computed as follows: There is a 50% chance X's parent is its successor (i.e., X is to the left of its parent); a 25% chance that X's "grandparent" is its successor; a 12.5% chance that X's "great-grandparent" is its successor, etc. That averages out to X's successor being about two levels above X. X will also be log(N)-1 nodes below the root node on average. So the parent information allows next to find the successor by going up about 2 levels instead of down log(N)-1 levels, so we save about log(N)-3 comparisons. Since about half of the nodes will have no right subtree, that implies a total of (N/2) * (log(N)-3) comparisons saved in traversing N nodes in a well-balanced tree.

17.33 The iterator would traverse the tree in a right-to-left inorder traversal (i.e., backwards).

17.34
```
public void preorderTraverseRL (QueueADT queue)
{     if (isEmpty())
            return;
      queue.enqueue (itsData);
      itsRight.preorderTraverseRL (queue);
      itsLeft. preorderTraverseRL (queue);
}
```

17.38 The total search time is 17 for the top three levels (already calculated in the text for that full tree) plus 8 times 4 for the fourth level, a total of 49. So the average search time is 49/15.

17.39 Start with the full tree with 3 nodes. Add a left child of the left child of the root for the 4-node tree. Add also a left child of the right child of the root for the 5-node tree.

17.40 The 7-node tree is the full tree with 7 nodes shown in Figure 17.7. Omit the bottom-right node (which contains 25 in Figure 17.7) to get the 6-node tree.

17.41 Add a makeBlack method to TreeNode that has the one statement isRed = false; then replace the last line of the put method (line 7) by the following three statements:

```
Object valueToReturn = itsRoot.putRecursive ((Comparable) id, value);
itsRoot.makeBlack();
return valueToReturn;
```

17.42 To the full tree with values 11, 13, and 15 in Figure 17.9, add 17 to the right of the 15 to get the 4-node tree. Adding 19 forces a rotation to put 17 in place of 15, 15 to 17's left, and 19 to 17's right. That gives the 5-node tree. Adding 21 then forces a rotation of 17 into the root, 13 as 17's left child, 11 as 13's left child, 15 as 13's right child, 19 as 17's right child, and 21 as 19's right child.

17.43 To the full tree with values 11, 13, and 15 in Figure 17.9, add 17 to the right of the 15 and color 11 and 15 black to get the 4-node tree. Adding 19 forces a rotation to put 17 in place of 15, red 15 to 17's left, and red 19 to 17's right. That gives the 5-node tree. Adding 21 then turns the 19-node black, the 15-node black, and the 17-node red without forcing a rotation.

17.50 size(3) = 1 + size(2) + size(1) = 1 + 2 + 1 = 4. size(4) = 1 + size(3) + size(2) = 1 + 4 + 2 = 7.

17.51 4 data values could mean 3 on the left side of the root (5 cases), or 3 on its right side (5 cases), or 2 on the left side of the root (2 cases) or 2 on its right side (2 cases), a total of 14 cases.
5 data values could mean 4 on the left side of the root (14 cases) or on its right side (14 cases), or 3 on the left side of the root (5 cases) or on its right side (5 cases), or 2 on each side (2*2 = 4 cases), a total of 42 cases for the 5-node tree.

17.52 The root could have 2 black children, each rooting one of 4 red-black trees with a black-height of 1, which makes 4 * 4 = 16 cases. Or it could have one red child, on the left, which will have 16 possible subtrees, with 4 possible subtrees on the right, which makes 16 * 4 = 64 more cases. Or it could have one red child, on the right, for 64 more cases. Or it could have two red children, each with 64 possible subtrees, thus 64 * 64 = 4096 more cases. The total is 4240 cases.

17.60 296 + 4 = 300. 8192 / 300 = 27 with 92 left over, so you could store 27 records in each node.

17.61 The root could have only 1 record, but still the second level would have at least 2 nodes with at least 50 subtrees each, so the third level would have at least 100 nodes with at least 50 subtrees each, so the fourth level would have at least 5000 nodes with at least 50 subtrees each, so the fifth level would have at least 250,000 nodes with at least 50 subtrees each, which is over 12 million. Therefore four disk accesses would be enough, assuming you keep the root node in memory.

18 Priority Queues And Heaps

Overview

You need to study at least the first part of Chapter Seventeen (Trees) before you study this chapter:

- Section 18.1 introduces the concept of a priority queue and discusses situations in which they can be useful, particularly file compression. A priority queue has the operations `add (Object)`, `removeMin()`, `peekMin()`, and `isEmpty()`.
- Section 18.2 presents the Comparator interface for generalizing ordering of data.
- Section 18.3 develops two implementations of the PriQue interface using arrays, one based on the insertion sort and the other based on the selection sort.
- Section 18.4 gives two corresponding implementations of PriQue using linked lists.
- Section 18.5 presents a fifth implementation that is more efficient when there are only a few different priority levels.
- Sections 18.6-18.7 describe how priority queues can be used directly for sorting. The TreeSort and HeapSort algorithms are discussed, as well as two implementations of PriQue based on binary trees and heaps. TreeSort is a variation of QuickSort.
- Sections 18.8-18.9 discuss the MergeSort algorithm for linked lists and go on to use a priority queue for sorting sequential files using the merging process. Two different implementations of sequential file sorting are presented.

18.1 File Compression Using Huffman Codes

All communications between spies employed by the ISPY Spy Company, Inc., are restricted to a vocabulary of 1024 very useful words. Each spy carries a dictionary in which each of the 1024 words has a 10-digit binary code, used in their digital messages (note that there are 1024 different 10-digit binary numbers, since $2^{10} = 1024$).

The company, having lost a few spies to the enemy due to messages taking too long to transmit, hires you to come up with a better plan for codes. You obtain a list of the frequency of use of each word in the last 10,000 messages and then develop a computer program that assigns binary codes to the 1024 words so that the average length of a message decreases by 23%, thereby saving the company an average of 2.7 spy lives per year and earning for yourself a $54,000 bonus.

At least, that is what you would do if you knew about the **Huffman code algorithm**. This algorithm provides the best possible assignment of binary codes as measured by average message length. The idea is that the words that are used frequently in spy messages (such as "war" and "kill") are assigned binary codes shorter than 10 binary digits (bits), while words used rarely (such as "peace" and "love") are assigned binary codes longer than 10 bits. That lowers the weighted average.

The algorithm begins by constructing 1024 2-node binary trees, each with a word in the left leaf of the root and the frequency with which that word occurs in the root (so each tree is "left-handed", in that it has an empty right subtree). These 1024 trees are put into a priority queue data structure. Among the services offered by a priority queue to the public are methods with the following headings for adding a data value to the structure and for removing and returning the data value with the lowest frequency:

```
public void add (Object ob)
public Object removeMin()
```

Next you repeat the following process 1023 times:

1. Remove from the priority queue the two trees with the smallest frequencies.
2. Combine those two trees into a single tree whose root contains the sum of the two frequencies.
3. Add that new combined tree to the priority queue data structure.

Eventually, this leaves the priority queue with a single binary tree containing all the words. The binary codes to be assigned to each word are easily derived from that tree by a process described later in this section.

Comparison with stacks and queues

A priority queue can be used for storing many kinds of data values, not just for storing binary trees that have a number in the root node. A priority queue uses a method that compares two data values to determine which has higher priority for removal from the data structure; in the case of the binary trees, a lower frequency gives a higher priority.

Assume you keep a pile of jobs to do, adding each new job you get to the top of the pile. The following contrasts priority queues with stacks and queues in terms of the way you choose the next job to carry out:

- If you always take the job on top, you are following a **Last-In-First-Out** principle (LIFO for short): Always take the job that has been in the pile for the shortest period of time. A data structure that implements this principle is called a **stack**.
- If you always take the job on the bottom, you are following a **First-In-First-Out** principle (FIFO for short): Always take the job that has been in the pile for the longest period of time. A data structure that implements this principle is called a **queue**.
- If you always take the job that has the highest priority, you are following a **Highest-Priority** principle: Always take the job in the pile that has the highest priority rating (according to whatever priority criterion you feel gives you the best chance of not getting grief from your boss). A data structure that implements this principle is called a **priority queue**.

The interface for priority queues

We will develop several different implementations of priority queues in this chapter. We specify what is common to all by defining a **PriQue** interface. An interface describes what operations the object can perform but does not give the coding. So it describes an **abstract data type**. The PriQue interface is in Listing 18.1 (see next page). A PriQue object has an operation to add an element to the data structure, an operation to remove an element, and two query methods: one to see what would be removed if requested, and one to tell whether the data structure has any elements to remove.

As an example of how a priority queue is used, the following coding applies the Huffman logic to a priority queue initially containing many of the 2-leaf binary trees. It leaves the priority queue with the single combined binary tree:

```
public void combineIntoOneTree (PriQue par)
{  TreeNode one = (TreeNode) par.removeMin();
   while (! par.isEmpty())
   {  TreeNode two = (TreeNode) par.removeMin();
      two.combineHuffmanly (one);
      par.add (two);
      one = (TreeNode) par.removeMin();
   }
   par.add (one);
}  //======================
```

Listing 18.1 The PriQue interface

```
public interface PriQue              // not in the Sun library
{
   /** Tell whether the priority queue has no more elements. */
   public boolean isEmpty();


   /** Return the object removeMin would return without modifying
    *  anything.  Throw an Exception if the queue is empty. */
   public Object peekMin();


   /** Delete the object of highest priority and return it.
    *  Priority is determined by a Comparator passed to the
    *  constructor.  Throw an Exception if the queue is empty. */
   public Object removeMin();


   /** Add the given element ob to the priority queue, so the
    *  priority queue has one more element than it had before.
    *  Precondition: The Comparator can be applied to ob. */
   public void add (Object ob);
}
```

For the given coding to work, you need to add a `combineHuffmanly` method to the TreeNode class of Chapter Seventeen that attaches the left subtree of `two` as the right subtree of `one` and then makes `one` the new left subtree of `two`. The method must also store the sum of the two frequencies in the root of `two`. This method is left as an exercise.

When you are done, you will have a "left-handed binary tree" (i.e., the right subtree is empty) with all 1024 words at the leafs and frequency values in the other nodes. Assign to each word the binary code corresponding to the path from itsRoot.itsLeft to the word, using 0 for a turn to the left and 1 for a turn to the right. For instance, if a word is stored in the leaf at (itsRoot.itsLeft).itsLeft.itsRight.itsLeft.itsRight.itsRight, its binary code will be 01011. This is provably the best possible correspondence of words to binary codes.

Figure 18.1 shows a smaller situation with eight words and their frequencies out of 100 occurrences ("peace" occurs 4 times and "kill" appears 28 times). The first iteration of the Huffman algorithm combines the two rarest words, "peace" and "love", into one tree with frequency 9 and inserts it after the 7-tree and the 8-tree. The next iteration combines "push" and "hit" into one tree with frequency 15 and inserts it after the 10-tree. The third iteration would combine the 9-tree with the 10-tree.

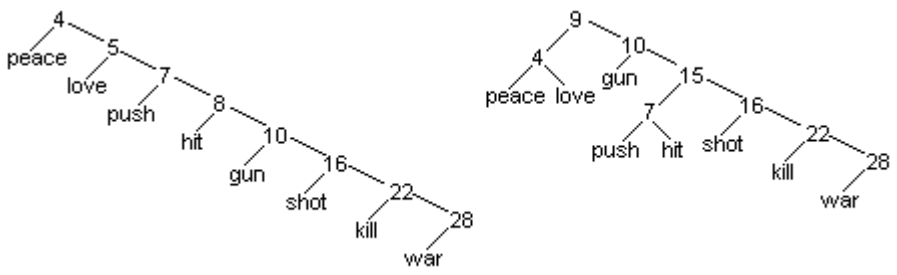
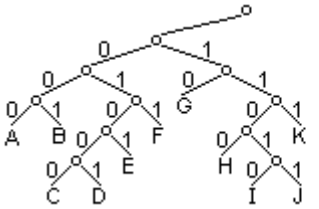


Figure 18.1 Huffman algorithm applied two times to 8 words

Figure 18.2 shows the binary codes that would be assigned to each of eleven 1-letter "words" if the final binary tree were what is shown on the left of the figure.



The binary codes for these 11 "words"

A 000	G 10
B 001	H 1100
C 01000	I 11010
D 01001	J 11011
E 0101	K 111
F 011	

Figure 18.2 A final Huffman tree and the resulting binary codes

File compression

Another situation in which Huffman coding is valuable is **file compression**. Consider a file of perhaps a million bytes of data. One **byte** is eight binary bits, so a byte ranges in value from -128 through 127. The UNIX file compression algorithm reads in the file and counts how many times each of the 256 byte values occurs. These frequencies are used in the Huffman coding algorithm to assign one sequence of bits to each byte value so that the more frequently-occurring byte values are assigned shorter bit sequences than the rarer byte values. This reduces the average bits per byte.

This process can lower the total size of the file by fifty percent or more. You can send the file, now perhaps less than half-a-million bytes, along with a 1K description of the encoding used, over the internet to another program that can decode the file using that description. The overall logic is in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for file compression

1. Read the given file, counting how many times each of the 256 different byte patterns occurs in that file. Each byte pattern is a "word" for purposes of this algorithm.
2. Put 256 different frequency+word binary trees into a priority queue, where higher priorities are given to trees with lower frequency.
3. Apply the Huffman process to obtain a single large binary tree that has 256 leaves, with one word in each leaf.
4. Calculate the binary code for each word from the path through the tree to that word. Some will be shorter than 8 bits and some longer but the average should be below 8.
5. Write a new file in which each byte pattern is replaced by its binary code.
6. Send the encoded file along with the short description of encoding.

Exercise 18.1 Write an independent method `public static void transfer (PriQue one, PriQue two)`: It transfers all elements from the first parameter into the second parameter. Precondition: Both parameters are non-null.
Exercise 18.2 After the third iteration of the Huffman algorithm for the example in Figure 18.1, what frequency is in the newly-inserted tree's root and what is in its left subtree?
Exercise 18.3 (harder) Write the `public void combineHuffmanly (TreeNode par)` TreeNode method as described in this section. Precondition: `par` is not null.
Exercise 18.4* Perform the fourth and fifth iterations of the Huffman algorithm for Figure 18.1 and show the sequence of binary trees.
Exercise 18.5* Perform the rest of the iterations of the Huffman algorithm for Figure 18.1 and show the sequence of binary trees. How much better is the average length of messages than the 3-bits-per-word length of messages without the coding?
Exercise 18.6** Write a program that applies the Huffman algorithm to any set of words and frequencies. Print a list of the words and binary codes, in their order left to right in the final binary tree. Use the TreeNode class, the String class for the words, and the Integer class for the frequencies. Note that this algorithm relies on the ability to store either of two kinds of Objects in a TreeNode.

18.2 Comparator Objects

A priority queue is appropriate when you have a number of jobs to do, each with its own priority. As time becomes available to do a job, you select the one with the highest priority. Whenever you receive another job to do, you add it to the list of jobs on hand. An example of this situation is a **printer queue**: A high-speed printer that serves a large number of users is continually receiving new print jobs to do, but prints those with highest priority first.

We assume that the priority is determined by an int-valued method named `compare` belonging to a **java.util.Comparator** object: For a given Comparator `test`, `test.compare(x,y)` returns a negative number if `x` has higher priority than `y`, a positive number if `x` has lower priority than `y`, and zero if they have the same priority. In other words, finding the value of higher priority corresponds to find the smaller value (so priority 1 is the highest priority). The Sun library Comparator interface is shown in the upper part of Listing 18.2. The middle part of this listing contains a description of the two constructors that any PriQue implementation should have. The bottom part defines the most commonly-used kind of Comparator, the one that corresponds to `compareTo`.

Listing 18.2 The Comparator interface and how PriQues use it

```
public interface java.util.Comparator  // in the Sun library
{
   /** Tell whether one has higher priority than two.  Throw
    *  a ClassCastException if they cannot be compared with
    *  each other.  */
   public int compare (Object one, Object two);
}
//###############################################################



// the two standard constructors for an implementation of PriQue
public class Whatever implements PriQue
{
   private java.util.Comparator itsTest;


   public Whatever (java.util.Comparator givenTest)
   {  itsTest = givenTest;
   }  //======================


   public Whatever()
   {  itsTest = new Ascendor();
   }  //======================
}
//###############################################################



public class Ascendor implements java.util.Comparator
{
   public int compare (Object one, Object two)
   {  return ((Comparable) one).compareTo (two);
   }  //======================
}
```

When you construct a new PriQue object, you usually supply to the constructor the Comparator object you want that priority queue to use. So one constructor has a Comparator parameter that is assigned to an instance variable of the priority queue.

An appropriate Comparator class for the Huffman tree situation, assuming roots of trees contain Integers, is as follows. Then you would create the PriQue object to hold Huffman trees using e.g. `PriQue queue = new ArrayOutPriQue (new HuffmanComp())` (ArrayOutPriQue will be defined shortly as an implementation of PriQue):

```
public class HuffmanComp implements java.util.Comparator
{
   public int compare (Object one, Object two)
   {  return ((Integer) ((TreeNode) one).getData()).intValue()
          - ((Integer) ((TreeNode) two).getData()).intValue();
   }  //======================
}
```

The `compare` method sounds almost like the standard `compareTo` method that any Comparable class of objects has. Sometimes we will use the standard `compareTo` method to determine priority. As a convenience, an implementation of PriQue should have a constructor with no parameters, so that the new priority queue will use the standard `compareTo` method that belongs to the Comparable objects being stored in the priority queue. For instance, if you want to store String objects in alphabetical order, you can use `PriQue queue = new ArrayOutPriQue()`. This uses an Ascendor object (defined in Listing 18.2) to do the comparing.

The reason for using some Comparator's `compare` method rather than the data values' `compareTo` method is that we sometimes want to prioritize a class of objects on one criterion and sometimes on another. Comparators allow us to pass as a parameter an object that brings with it the appropriate `compare` function. Such an object is called a **functor** or **function object**, since its only purpose is to supply a function. Functors have instance methods but usually do not have instance variables.

Relation to stacks and (ordinary) queues

The PriQue methods correspond to the stack and queue methods described in Chapter Fourteen:

- `isEmpty` is the same as StackADT's `isEmpty` and QueueADT's `isEmpty`.
- `add` corresponds to `push` and `enqueue`;
- `peekMin` corresponds to `peekTop` and `peekFront`;
- `removeMin` corresponds to `pop` and `dequeue`;

The only real difference is in the effect of `removeMin` and `pop` and `dequeue`: `pop` delivers the object that has been in the data structure for the shortest period of time, `dequeue` delivers the object that has been in the data structure for the longest period of time, and `removeMin` delivers the object that has the highest priority. A PriQue implementation is **stable** if, in case of ties in priority, the element that has been in the data structure the longest is always removed first.

Another example

Suppose you have several classes with a `getCost()` method, all declared to implement this Priceable interface:

```
public interface Priceable
{     public int getCost();
}
```

Then you could use `PriQue queue = new ArrayOutPriQue (new ByCost());` to create the priority queue, with higher costs indicating higher priority, if you have the following definition:

```
public class ByCost implements java.util.Comparator
{
   public int compare (Object one, Object two)
   {  return ((Priceable) two).getCost()
           - ((Priceable) one).getCost();
   }  //======================
}
```

Whatever Comparator you define, it should have the properties of a **total ordering**. This means that, given two objects referenced by `sam` and `sue` for which `compare(sam, sue)` does not throw an Exception, the following should be true:

- If `sam.equals(sue)` is true, then `compare(sam,sue)` should be zero.
- The value of `compare(sue,sam)` should be the negative of `compare(sam, sue)` (i.e., one is positive and the other is negative, or else both are zero).
- If `compare(sam,x)` is positive and `compare(x,sue)` is positive, then `compare(sam,sue)` should be positive too. This is **transitivity**.

If the `compare` method you define can have `compare(sam,sue)` be zero only when `sam.equals(sue)` is true, it is said to be **consistent with `equals`**. If the `compare` method you define is not consistent with `equals`, then instances of some standard library classes (such as SortedSet) may refuse to let you add certain objects to the data structure.

Exercise 18.7 The java.awt.RectangularShape class in the Sun standard library has several subclasses such as Rectangle2D and Ellipse2D. The RectangularShape class has two instance methods `getX()` and `getY()` to retrieve the x- and y-coordinates of its upper-left corner on a Graphics page. A Graphics page has its non-negative coordinates numbered from 0 up starting at the upper-left corner of the page. Create a Comparator for RectangularShape objects so that higher objects have higher priority or, if both objects are at the same height, the one that is further right takes priority.

Exercise 18.8* Essay: Explain why any `compare` method that has the properties of a total ordering will also have this property: If `compare(x,y)` is negative and `compare(y,z)` is negative, then `compare(x,z)` is negative too.

Exercise 18.9* For the BigCircle class shown in Listing 17.13, write a constructor that has a Comparator object as a parameter. Use that Comparator object in the coding of the `get` method for BigCircles.

18.3 Implementing Priority Queues With Arrays

One obvious way to manage the values in a priority queue is to store them in a partially-filled array in the order in which they will be taken out, so that the next value to remove is always at `itsItem[itsSize-1]` (analogous to ArrayStack in Listing 14.2). This plan almost completely determines the coding for the four methods. Listing 18.3 contains this **ArrayOutPriQue** class. Note that `isEmpty`, `peekMin`, and `removeMin` are coded exactly the same as `isEmpty`, `peekTop`, and `pop` for ArrayStack. Reminder: IllegalStateException is in the java.lang package.

Listing 18.3 An array implementation of PriQue, partially done

```
public class ArrayOutPriQue implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private java.util.Comparator itsTest;


   public ArrayOutPriQue (java.util.Comparator givenTest)
   {  itsTest = givenTest;                                    //1
   }  //======================

   public ArrayOutPriQue()
   {  itsTest = new Ascendor();                               //2
   }  //======================

   public boolean isEmpty()
   {  return itsSize == 0;                                    //3
   }  //======================

   public Object peekMin()
   {  if (isEmpty())                                          //4
         throw new IllegalStateException ("priority Q is empty");
      return itsItem[itsSize - 1];                            //6
   }  //======================

   public Object removeMin()
   {  if (isEmpty())                                          //7
         throw new IllegalStateException ("priority Q is empty");
      itsSize--;                                              //9
      return itsItem[itsSize];                                //10
   }  //======================

   public void add (Object ob)
   {  if (itsSize == itsItem.length)                          //11
      {  }  // left as an exercise                           //12
      int k = itsSize;                                        //13
      while (k > 0 && itsTest.compare (ob, itsItem[k - 1]) >= 0)
      {  itsItem[k] = itsItem[k - 1];                         //15
         k--;                                                 //16
      }                                                       //17
      itsItem[k] = ob;                                        //18
      itsSize++;                                              //19
   }  //======================
}
```

If the `add` method is called when `itsSize` is `itsItem.length`, you need to increase the size of the array. This part of the coding of `add` is left as an exercise. In any case, when the `add` method is called, you move values up until you open up a spot where the new value goes to keep all values in order; it generally goes at an index `k` such that `itsTest.compareTo (ob, itsItem[k - 1]) < 0`.

Figure 18.3 illustrates the process (small numbers indicate high priority): Adding an element of priority 5 requires shifting the three elements of higher priorities 4, 2, and 1 further toward the rear of the array to make room for the 5 between the 6 and the 4.

indexes of components:	0	1	2	3	4	5	6
current status of the array:	8	6	4	2	1	no data	no data
call add(5):	8	6	**5**	4	2	1	no data
call removeMin():	8	6	5	4	2	no data	no data

Figure 18.3 For ArrayOutPriQue: Call add(5), then call removeMin()

Internal invariant for ArrayOutPriQues

- The int value `itsSize` is the number of elements in the abstract priority queue. These elements are stored in the array of Objects `itsItem` at the components indexed 0 up to but not including `itsSize`.
- If `k` is any positive integer less than `itsSize`, then the element at index `k` has either equal or higher priority than the element at index `k-1`; and if their priorities are equal, the element at index `k` has been in the queue longer. In particular, the element at index `itsSize-1` has the highest priority of all elements in the priority queue.

Storing the elements in the order they come in

An alternative implementation for a priority queue is to just put the next value added in the array at index `itsSize`. That way, data values remain in the order they are put in the structure. This means that `isEmpty` and `add` are coded the same as are `isEmpty` and `push` for ArrayStack.

When the `peekMin` method is called, you have to search through the array to find the value of highest priority and return it. Coding for the `peekMin` method of this **ArrayInPriQue** class of objects could be as shown in the upper part of Listing 18.4 (see next page).

If two elements are tied for the highest priority, this `peekMin` method does not necessarily return the one that has been on the queue for the longest time, so it is not a stable PriQue implementation (whereas ArrayOutPriQue is). Correcting this defect is left as an exercise. Notation: "ArrayIn" reminds you they are stored in an array in the order of input to the data structure; "ArrayOut" reminds you they are stored in an array in the order of output from the data structure.

The call of `itsTest.compare` can invoke the `compare` method that uses a String's `compareTo` method or the `compare` method that uses a Priceable object's `getCost` method or any of a number of other possible codings. It is therefore a polymorphic method call.

Listing 18.4 The ArrayInPriQue class of objects, partially done

```
public class ArrayInPriQue implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private java.util.Comparator itsTest;


   // the two constructors and isEmpty are as for ArrayOutPriQue
   // the add method is left as an exercise


   public Object peekMin()
   {  return itsItem[searchMin()];                              //1
   }  //======================


   private int searchMin()
   {  if (isEmpty())                                            //2
         throw new IllegalStateException ("priority Q is empty");
      int best = 0;                                             //4
      for (int k = 1;  k < itsSize;  k++)                       //5
      {  if (itsTest.compare (itsItem[k], itsItem[best]) < 0)//6
            best = k;                                           //7
      }                                                         //8
      return best;                                              //9
   }  //======================


   public Object removeMin()
   {  int best = searchMin();                                   //10
      itsSize--;                                                //11
      Object valueToReturn = itsItem[best];                     //12
      itsItem[best] = itsItem[itsSize];                         //13
      return valueToReturn;                                     //14
   }  //======================
}
```

Exercise 18.10 Write the missing part of the `add` method for ArrayOutPriQue.
Exercise 18.11 Write the entire `add` method for ArrayInPriQue.
Exercise 18.12 Revise the `removeMin` method for ArrayInPriQue so it always returns elements of equal priority in first-in-first-out order (i.e., maintain stability). Do so by replacing the next-to-last statement by coding that moves elements down while keeping their original order.
Exercise 18.13* Essay: Explain why you may delete both two-line if-statements beginning `if(isEmpty())` in Listing 18.3 without violating the specifications for PriQue, but you may not delete the two-line if-statement in Listing 18.4.
Exercise 18.14* Write out the internal invariant for the ArrayInPriQue class.
Exercise 18.15* Write a complete implementation of PriQue for which the element of highest priority is always kept in `itsItem[itsSize-1]` and the rest are in the order in which they were entered. Note that this makes the `peekMin` method execute very quickly. Do not try to make this implementation stable.
Exercise 18.16* A while-condition that makes two tests (as for the while-condition in the `add` method of Listing 18.3) is moderately slower than a loop that makes one test. Rewrite that loop to first test `itsTest.compare (ob, itsItem[0])` and then execute one of two different loops, each with a condition that only makes one test.

18.4 Implementing Priority Queues With Linked Lists

You can store the elements of a priority queue in a linked list instead of in an array. This section discusses two different ways to do this, analogous to the two ways implemented in the previous section. We put a private nested Node class in each of these linked list implementations, to maintain encapsulation.

Implementing PriQue with a linked list in the order elements go out

You could keep the data in order of priority, with the highest-priority data in the first Node of the linked list. Then when you want to remove it or just get it, you have it immediately available. This **NodeOutPriQue** class needs an instance variable to record the first Node on the linked list; call it `itsFirst`. The only tricky part is the `add` method, since it has to put the given element into the linked list after every value of equal or higher priority.

The coding of `add` is much easier if you always keep an extra Node at the end of the linked list with null data in that **trailer node**. So a NodeOutPriQue priority queue is empty whenever `itsFirst.itsNext` is null, not when `itsFirst` is null. The coding for `isEmpty` and `removeMin` is in the upper part of Listing 18.5 (see next page). We omit the coding for the two constructors (they are the same as usual) and `peekMin` (it is left as an exercise).

For the `add` method, you need to search through the linked list for the Node that contains the value that should come after the given element in the list. Have a variable `p` refer to that Node. The statement `p = p.itsNext` moves `p` from whichever Node it is on to the next Node. The loop stops at the trailer node if not before, i.e., when `p.itsNext == null`. Then you can insert the given element `ob` in that Node `p` and make a new Node after `p` to contain the data value that was in `p`, as follows:

```
p.itsNext = new Node (p.itsData, p.itsNext);
p.itsData = ob;
```

This would not work if `ob` is to be added at the end of a standard linked list, since then `p` would be null. But with the trailer node, when `ob` is to be added at the end, the statement above copies the information from the trailer node into a new trailer node and has `ob` replace the null data in the old trailer node. The coding is in the lower part of Listing 18.5. Figure 18.4 shows stages in the execution of the `add` method, where the numbers indicate the priority of the data value.

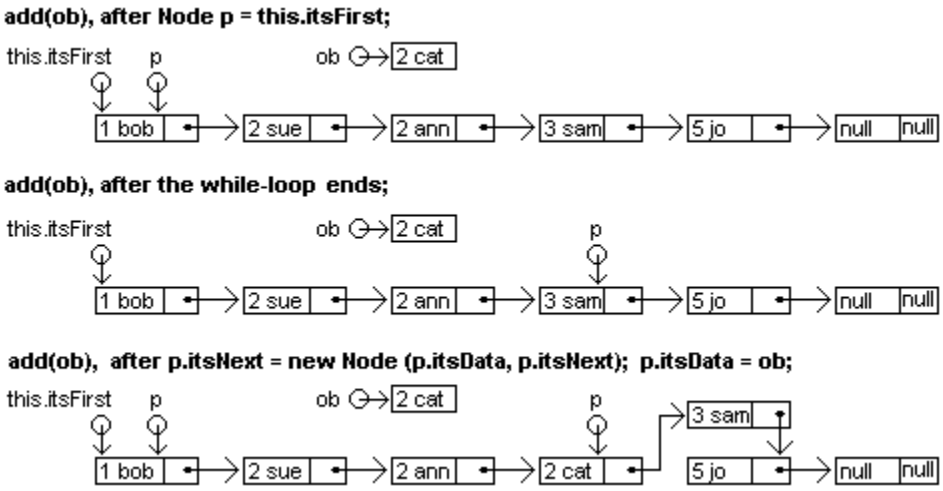


Figure 18.4 Adding a new element of priority 2 in NodeOutPriQue

Listing 18.5 The NodeOutPriQue class of objects, partially done

```
public class NodeOutPriQue implements PriQue
{
   private Node itsFirst = new Node (null, null); // trailer node
   private java.util.Comparator itsTest;


   public boolean isEmpty()
   {  return itsFirst.itsNext == null;                        //1
   }  //======================


   public Object removeMin()
   {  if (isEmpty())                                          //2
         throw new IllegalStateException ("priority Q is empty");
      Node toDiscard = itsFirst;                              //4
      itsFirst = itsFirst.itsNext;                            //5
      return toDiscard.itsData;                               //6
   }  //======================


   public void add (Object ob)
   {  Node p = this.itsFirst;                                 //7
      while (p.itsNext != null                                //8
                  && itsTest.compare (ob, p.itsData) >= 0) //9
      {  p = p.itsNext;                                       //10
      }                                                       //11
      p.itsNext = new Node (p.itsData, p.itsNext);            //12
      p.itsData = ob;                                         //13
   }  //======================


   private static class Node
   {
      public Object itsData;
      public Node itsNext;

      public Node (Object data, Node next)
      {  itsData = data;                                      //14
         itsNext = next;                                      //15
      }
   }  //======================
}
```

Implementing PriQue with a linked list in the order elements come in

The next implementation is **NodeInPriQue**, which implements a priority queue similar to ArrayInPriQue: Add each element to the front of the linked list as it comes in; when you need to produce the element of highest priority, search through the list to find it. Each NodeInPriQue object is to have an instance variable `itsFirst` to note the first Node on its linked list. Like NodeOutPriQue, the linked list is made with a private nested Node class and a trailer node.

The private `searchMin` method goes through the linked list to find the node containing the data value of highest priority. If the queue is empty, the `searchMin` coding throws an Exception when `p.itsNext` is evaluated, as it should. The `peekMin` method simply returns the data value in the node that `searchMin` finds. This coding is in the middle part of Listing 18.6 (see next page).

Listing 18.6 The NodeInPriQue class of objects, using trailer nodes

```
public class NodeInPriQue implements PriQue
{
   private Node itsFirst = new Node (null, null); // trailer node
   private java.util.Comparator itsTest;


   // the 2 constructors are the same as usual (see Listing 18.2)
   // the private Node class is the same as for NodeOutPriQue


   public boolean isEmpty()
   {  return itsFirst.itsNext == null;                          //1
   }  //======================


   public Object peekMin()
   {  return searchMin().itsData;                               //2
   }  //======================


   private Node searchMin()
   {  Node best = itsFirst;                                     //3
      for (Node p = itsFirst.itsNext;  p.itsNext != null;       //4
                                       p = p.itsNext)           //5
      {  if (itsTest.compare (p.itsData, best.itsData) <= 0) //6
            best = p;                                           //7
      }                                                         //8
      return best;                                              //9
   }  //======================


   public Object removeMin()
   {  Node best = searchMin();                                  //10
      Object valueToReturn = best.itsData;                      //11
      Node toDiscard = best.itsNext;                            //12
      best.itsData = toDiscard.itsData;                         //13
      best.itsNext = toDiscard.itsNext;                         //14
      return valueToReturn;                                     //15
   }  //======================


   public void add (Object ob)
   {  itsFirst = new Node (ob, itsFirst);                       //16
   }  //======================
}
```

An empty NodeInPriQue object is constructed with `itsFirst` being a reference to a trailer node with null for `itsData` and null for `itsNext`. Adding another data value just requires creating a new node to contain it and putting it at the beginning of the list:

```
itsFirst = new Node (ob, itsFirst);
```

The `removeMin` method does the same as `peekMin` except that it also deletes the value to be returned. You have to avoid changing the order of the remaining elements. To delete the element in a Node referenced by `best`, it is sufficient to replace it by the element in the Node following `best` and then delete the Node following `best`. This is

where the existence of the trailer node comes in handy. It guarantees that there will always be a Node following `best`, the trailer node if nothing else. The coding for the `removeMin` method is in the lower part of Listing 18.6.

Reminder Nesting only affects visibility: Methods inside NodeInPriQue can access the public variables of the Node class but outside methods cannot. So the principle of encapsulation is not violated by making Node's instance variables public.

An alternative to having a trailer node is to search through the list for the Node before the one that contains the highest-priority element and set that Node's `itsNext` value to the one after the one containing the highest-priority element. This is more complicated, so we leave that for a (hard) exercise. You could try it if you want, however, so you will understand why this book prefers to use a trailer node.

Exercise 18.17 Write the `peekMin` method for NodeOutPriQue.
Exercise 18.18 Rewrite the `removeMin` method for NodeInPriQue to not use a local variable for the Node to be deleted.
Exercise 18.19 How would you revise the `add` method for NodeOutPriQue if you were not to allow two elements with equal priority to be in the priority queue?
Exercise 18.20 Add a method `public int size()` to the NodeOutPriQue class: The executor counts and returns the number of values currently in the priority queue.
Exercise 18.21 Add a method `public String toString()` to the NodeOutPriQue class: The executor returns the concatenation of the string representation of each element currently in the queue with a tab character before each element, in order front to rear. This is very useful for debugging purposes.
Exercise 18.22 Write a method `public void removeAbove (Object ob)` that could be added to NodeOutPriQue: The executor removes all values from the stack that have higher priority than the parameter. Precondition: `itsTest` applies to `ob`.
Exercise 18.23 (harder) Same as the previous exercise, except for NodeInPriQue.
Exercise 18.24 (harder) Rewrite the `add` method for NodeInPriQue to add the given element in the second position of the linked list if the list is not empty and the element does not have higher priority than the data in the first Node. This will be used in the next exercise.
Exercise 18.25* Rewrite the `removeMin` method for NodeInPriQue so that, in conjunction with the `add` method of the preceding exercise, the element that is to be removed next is always `itsFirst.itsData` and the rest are in the order they were added. Note that this makes `peekMin` execute much faster.
Exercise 18.26* Explain why the condition in the private `searchMin` method of Listing 18.6 should not be written with `< 0` instead of `<= 0` for the comparison.
Exercise 18.27* Write out the internal invariant for the NodeOutPriQue class.
Exercise 18.28* Write a method `public void removeBelow (Object ob)` that could be added to NodeOutPriQue: The executor removes all values from the priority queue that have lower priority than the given element. Precondition: `itsTest` applies to `ob`.
Exercise 18.29* Same as the preceding exercise, but for NodeInPriQue.
Exercise 18.30* Rewrite the full implementation of NodeInPriQue so that each element is added to the end of a linked list but without looping (i.e., the reverse ordering). Hint: Add an instance variable that keeps track of the last node in the linked list.
Exercise 18.31* Essay: Explain why both of the linked list implementations of PriQue in this section are stable.
Exercise 18.32** Rewrite the full implementation of NodeOutPriQue without a trailer node, so that the number of Nodes equals the number of elements.
Exercise 18.33** Rewrite the full implementation of NodeInPriQue without a trailer node, so that the number of Nodes equals the number of elements.
Exercise 18.34** Write a method `public void add (NodeOutPriQue queue)` that could be in NodeOutPriQue: The executor interweaves `queue` 's elements in priority order and sets `queue` to be empty. Do not call on the existing `add` method.

Part B Enrichment And Reinforcement

18.5 Implementing Priority Queues With Linked Lists Of Queues

Some situations have only a few priority levels. For instance, you might have only five different priority levels with hundreds of elements at each level. If you add an element with low priority, you may have to go through many hundreds of elements to find the place where your given element is to be inserted.

In such a situation, the logic would execute substantially faster if you had just five regular queues, one for each priority level. Then you could go to the correct queue in at most five steps and quickly append the given element at the end of that queue. You could have a linked list of queues instead of a linked list of elements. You keep these queues in order of the priorities of the elements on them. The queue with the highest priority is first in the linked list.

This NodeGroupPriQue implementation uses a private nested Node class where `itsData` is a NodeQueue value instead of an Object value. Reminder: NodeQueue, an implementation of QueueADT, has these four methods for working with a FIFO queue:

- `q.isEmpty()` tells whether the queue has elements.
- `q.enqueue(ob)` adds ob to the rear of the queue.
- `q.dequeue()` removes and returns the element at the front of the queue.
- `q.peekFront()` returns the element at the front of the queue without removing it.

Each NodeGroupPriQue object has an instance variable `itsFirst` that keeps track of the linked list of queues; all of the queues on the list are to be non-empty. It is highly convenient to have a trailer node at the end of this linked list. Figure 18.5 shows roughly how this whole thing is structured: A NodeGroupPriQue's `itsFirst` is a Node whose `itsData` is a NodeQueue whose `itsFront` and `itsRear` are Nodes in a linked list of data (but the first kind of Node is a nested class; the second kind is stand-alone).

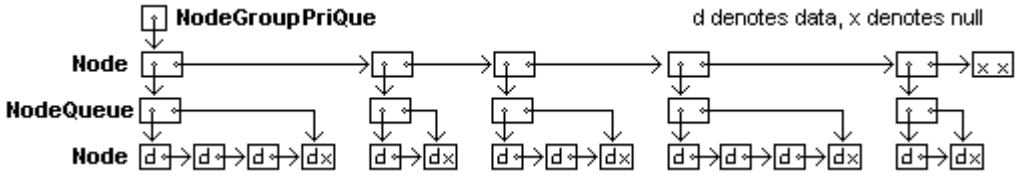


Figure 18.5 Linked list of Nodes p for which p.itsData is a NodeQueue

The isEmpty, peekMin, and removeMin methods

The `isEmpty` method only needs to check that `itsNext` for the first Node on the linked list is null. The element to be returned by the `peekMin` method is the front element of the queue with the highest priority. Since that is the first queue in the linked list, you return the `peekFront` value for that queue (since the queue is known to be non-empty). This coding is in the upper part of Listing 18.7 (see next page). Note that it will throw an Exception if there is no data, which is what it should do.

The `removeMin` method is about the same as the `peekMin` method except it returns `itsFirst.itsData.dequeue()`, which removes the front element on the queue of elements of highest priority. The call of `removeMin` might leave that first queue empty. Since the internal invariant of this implementation requires that you only store non-empty queues, you must discard the first queue if it has become empty as a consequence of dequeueing an element from it. This coding is in the middle part of Listing 18.7.

Listing 18.7 The NodeGroupPriQue class of objects

```
public class NodeGroupPriQue implements PriQue
{
   private Node itsFirst = new Node (null, null); // trailer node
   private java.util.Comparator itsTest;


   // the 2 constructors are the same as usual (see Listing 18.2)

   public boolean isEmpty()
   {  return itsFirst.itsNext == null;                          //1
   }  //======================

   public Object peekMin()
   {  return itsFirst.itsData.peekFront();                      //2
   }  //======================

   public Object removeMin()
   {  Object valueToReturn = itsFirst.itsData.dequeue();        //3
      if (itsFirst.itsData.isEmpty())  // after the dequeue     //4
         itsFirst = itsFirst.itsNext;  // lose the queue        //5
      return valueToReturn;
   }  //======================

   public void add (Object ob)
   {  Node p = this.itsFirst;                                   //6
      while (p.itsNext != null                                  //7
            && itsTest.compare (ob, p.itsData.peekFront()) > 0)
      {  p = p.itsNext;                                         //9
      }                                                         //10
      if (p.itsNext == null                                     //11
            || itsTest.compare (ob, p.itsData.peekFront()) < 0)
      {  p.itsNext = new Node (p.itsData, p.itsNext);           //13
         p.itsData = new NodeQueue();                           //14
      }                                                         //15
      p.itsData.enqueue (ob);                                   //16
   }  //======================


   private static class Node
   {
      public QueueADT itsData;
      public Node itsNext;

      public Node (QueueADT data, Node next)
      {  itsData = data;                                        //17
         itsNext = next;                                        //18
      }
   }  //======================
}
```

You should compare the coding throughout this Listing 18.7 with the coding for NodeOutPriQue in the earlier Listing 18.5 to see how similar they are. In fact, if no two elements can have the same priority, the NodeGroupPriQue implementation becomes the same as NodeOutPriQue except for an awful lot of extraneous creating and discarding of queue objects. Figure 18.6 (see next page) is the UML class diagram for the NodeGroupPriQue class.

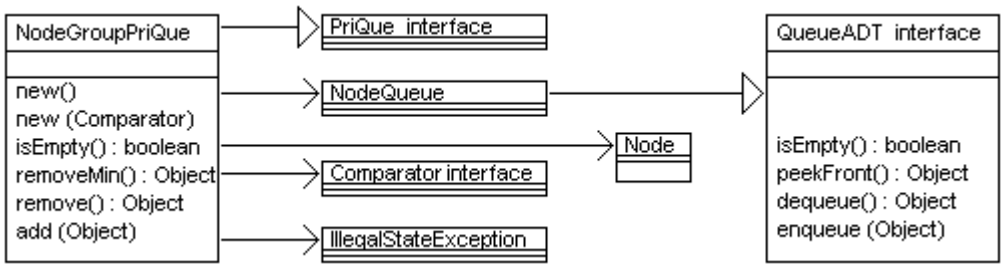


Figure 18.6 UML class diagram of the NodeGroupPriQue class

The add method

To add an element, you need to search for the first Node that contains a queue whose elements do not have higher priority than the given element. Of course, you do not go beyond the last Node in the linked list, which is the trailer node. So the condition for the while-statement is the logical equivalent of the following:

```
p.itsNext != null && ob > p.itsData.peekFront()
```

By contrast, the while-condition for NodeOutPriQue was the equivalent of the following, because you needed to go past not only elements of higher priority but also those of the same priority that had been on the linked list longer:

```
p.itsNext != null && ob >= p.itsData
```

Once you find the right Node `p`, see if it has a queue of elements with the same priority as the element you are supposed to add. If not (because the Node is empty or it has a queue of elements with lower priority), insert a new empty queue just before the queue in the Node you have found. In either case, you should now `enqueue` the given element onto that queue. This coding is in the lower part of Listing 18.7.

A variation

The implementation of PriQue just described works well as long as you know there are only a few different priority values, on the order of 5 to 20. You do not have to know what those values are.

Now if you know that the priority values are int values in the range from 1 to e.g. 100, you can have an implementation that uses an array of QueueADT objects, one for each int value. Adding a new element would involve finding its priority value `pv` and then storing that element on the queue at index `pv` in the array. And removing the element of highest priority would require searching the array for the first non-empty queue and dequeueing the front element on that queue. This is left as a major programming project.

Exercise 18.35 List the changes that Listing 18.7 would require if the `itsData` field of the Node class were declared as Object rather than as QueueADT.
Exercise 18.36* Write out the internal invariant for NodeGroupPriQue.
Exercise 18.37* Revise the Node class for Listing 18.7 to have a third instance variable `itsKey`, which is any element that is or has been on the queue stored in `itsData`. Use this revision to rewrite the `add` method to execute faster.

18.6 Sorting Using Priority Queues; The TreeSort Algorithm

A priority queue can be used to sort a large number of values. If you add them one at a time to the priority queue without ever calling `removeMin`, and then remove them one at a time until the priority queue is empty, they come out in the order determined by `compare`. If you have a Ascendor object (i.e., using `compareTo`), they come out in their natural ascending order. Specifically, the following method would sort the values in an ordinary queue, assuming that the given priority queue is initially empty:

```
public static void sort (QueueADT source, PriQue piq)
{  while ( ! source.isEmpty())                    // Loop #1
      piq.add (source.dequeue());
   while ( ! piq.isEmpty())                       // Loop #2
      source.enqueue (piq.removeMin());
}
```

In other situations, you might want a method to sort all the values in an array:

```
public static void sort (Object[] source, PriQue piq)
{  for (int k = source.length - 1;  k >= 0;  k--)  // Loop #1
      piq.add (source[k]);
   for (int k = 0;  k < source.length;  k++)       // Loop #2
      source[k] = piq.removeMin();
}
```

Either of the following statements does an InsertionSort, since the priority queue's `add` method inserts each value into its current list of values in ascending order and the `removeMin` method does virtually no work for these two "outie" implementations:

```
sort (source, new ArrayOutPriQue());
sort (source, new NodeOutPriQue());
```

Either of the following statements does a SelectionSort, since the priority queue's `removeMin` method runs through the entire list of values to select the minimum one and the `add` method does virtually no work for these two "innie" implementations:

```
sort (source, new ArrayInPriQue());
sort (source, new NodeInPriQue());
```

For the two InsertionSort cases, each call of `add` is a big-oh of N operation and each call of `removeMin` is big-oh of 1. So Loop #1 of the `sort` method takes big-oh of N^2 time, but Loop #2 takes big-oh of N time. For the two SelectionSort cases, each call of `removeMin` is a big-oh of N operation and each call of `add` is big-oh of 1. So Loop #2 of the `sort` method takes big-oh of N^2 time, but Loop #1 takes big-oh of N time.

When you use the priority queue implementation described at the end of the preceding section to sort values, for situations where priorities are int values in a limited range, you are using a BucketSort (which is more thoroughly described in Section 13.7).

Using the QuickSort logic

The QuickSort algorithm suggests another way to implement a priority queue that can execute much faster on average than the four elementary implementations mentioned above. You do it by keeping track of all the pivot elements used in sorting the elements so far. When you want to add another element `x`, you compare it with the first pivot (the first element that was added to the priority queue). If `x` has higher priority than the first pivot, it goes to the left of the first pivot; otherwise `x` goes to the right of the first pivot.

In the first case you compare `x` with the pivot that was used to split up the group of elements with higher priority than the first pivot. In the second case you compare `x` with the pivot that was used to split up the group of elements without higher priority than the first pivot. Either way, you put `x` to the left of that second pivot if `x` has higher priority than it, otherwise you put it to the right of that second pivot.

This continues until you see that `x` goes to one side of a pivot that does not already have any elements on that side. Then you put `x` in that position; it will be the pivot for future additions to the data structure at that point. However, if you use an array, you now have to move a lot of elements to make room for the new element, often more than half of them. This is a big-oh of N operation, which loses all the advantage of the QuickSort. Also, how are you going to keep track of all those pivot relationships?

Using TreeNodes

The standard solution is to use TreeNodes to store the values. In a TreeNode, `itsData` is the pivot for its group of elements. `itsLeft` refers to the TreeNode containing the pivot that was used for all elements that were compared with `itsData` and found to be of higher priority. `itsRight` refers to the TreeNode containing the pivot for all the elements that were compared with `itsData` and found to be of lesser or equal priority. A priority queue with this implementation could be called **TreePriQue**; see Listing 18.8.

Listing 18.8 The TreePriQue class of objects

```
public class TreePriQue implements PriQue
{
   private TreeNode itsRoot = TreeNode.ET;
   private java.util.Comparator itsTest;


   // the 2 constructors are the same as usual (see Listing 18.2)

   public boolean isEmpty()
   {  return itsRoot.isEmpty();
   }  //======================


   public Object peekMin()
   {  return itsRoot.firstNode().getData();
   }  //======================


   public void add (Object ob)
   {  if (itsRoot == TreeNode.ET)
         itsRoot = new TreeNode (ob);
      else
         itsRoot.add (ob, itsTest);
   }  //======================


   public Object removeMin()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      TreeNode[] newRoot = {itsRoot};
      Object valueToReturn = itsRoot.removeFirst (newRoot);
      itsRoot = newRoot[0];
      return valueToReturn;
   }  //======================
}
```

The first element added to the priority queue, with which you compare all other elements, is stored in a TreeNode named `itsRoot`. This is the only piece of information that the priority queue object has to keep track of. So you tell whether a priority queue is empty by seeing whether `itsRoot` value is `ET`. If you ask the priority queue to `get` the element of the highest priority, it asks `itsRoot` to find out and tell it (unless `itsRoot` is `ET`, in which case it throws an Exception). This coding is in the upper part of Listing 18.8. It calls the `firstNode` method given for the TreeNode class in Listing 17.2.

When you ask a TreePriQue object to add another element, there are two cases: If the TreePriQue object is empty, then the new element is added as the root value. Otherwise it asks `itsRoot` to add the element. This coding is in the middle part of Listing 18.8. It calls the `add` method in the TreeNode class, which is coded later in this section.

When you ask a TreePriQue object to remove the element of highest priority, it asks `itsRoot` to do so for it. But if it is empty, it throws an Exception. You could just code the `removeMin` method for a structure as `return itsRoot.removeFirst()` except for one problem: Sometimes it is the data in the root that is to be deleted, in which case you have to change `itsRoot` to be the root of whatever tree remains. So the TreeNode `removeFirst` method has to return two values: the `valueToReturn` and the new value of `itsRoot`. But a Java method can only return one value.

If you pass `itsRoot` as a parameter, any change that the method makes to the formal parameter does not change the value of the actual parameter `itsRoot`. A standard solution to the problem of getting two return values is to make one parameter be an array with one component. The method called can then change the value stored in that component if it needs to. The calling method can then use that new value to do what it has to do. This coding is in the lower part of Listing 18.8.

Figure 18.7 shows what we want to have happen when elements are added to and removed from the tree. Each little circle represents a TreeNode. The circle with the priority code 6 in it is `itsRoot`. The circle with the priority code 3 in it is `itsRoot.itsLeft` and the higher circle with the 8 in it is `itsRoot.itsRight`. Figure 18.7 traces the action of adding a value with priority 4, then adding a value with priority 7, then removing the value with the highest priority.

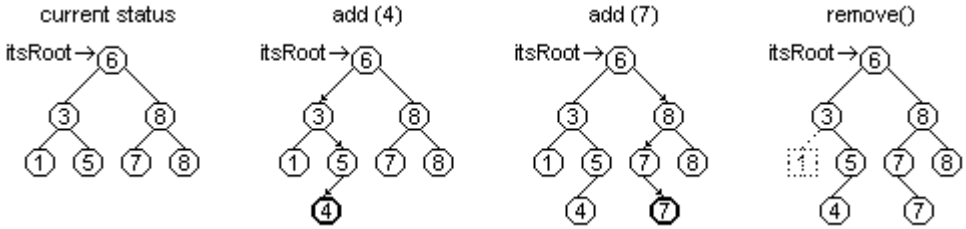


Figure 18.7 Effect on TreeNodes of some TreePriQue method calls

For the example in this figure, the `removeFirst` method would not change the parameter. But if the root node with the priority code 6 in it had no nodes to its left, a call of `removeFirst` would change the parameter: `root[0]` = the node with the 8.

Additions to the TreeNode class

The two new methods required for the TreeNode class are in Listing 18.9 (see next page). The `firstNode` method and the `getData` method are in Listing 17.2.

The `removeFirst` method first verifies that there is a data value to the left of the executor (line 1). If not, it must return the data value in the executor, and also "return" the TreeNode that will replace the executor in the TreePriQue structure (line 2; this "return" is done via the 1-element array parameter).

Listing 18.9 Additions to accomodate TreePriQue

```
// The following 2 methods are added to the TreeNode class

   /** Delete and return the first data value in a non-empty
    *  tree.  If that data value is in the executor, assign
    *  to root the TreeNode to the executor's right. */

   public Object removeFirst (TreeNode[] root)
   {  if (itsLeft == ET)                                        //1
      {  root[0] = this.itsRight;                               //2
         return this.itsData;                                   //3
      }                                                         //4
      TreeNode p = this;                                        //5
      while (p.itsLeft.itsLeft != ET)                           //6
         p = p.itsLeft; // p becomes parent of leftmost node    //7
      Object valueToReturn = p.itsLeft.itsData;                 //8
      p.itsLeft = p.itsLeft.itsRight; // it may be ET           //9
      return valueToReturn;                                     //10
   }  //======================


   /** Add ob to the tree, keeping the binary search property.
    *  Precondition:  this is not an empty tree.  */

   public void add (Object ob, java.util.Comparator test)
   {  if (test.compare (ob, itsData) < 0)                       //11
      {  if (itsLeft == ET)                                     //12
            itsLeft = new TreeNode (ob);                        //13
         else                                                   //14
            itsLeft.add (ob, test);                             //15
      }                                                         //16
      else                                                      //17
      {  if (itsRight == ET)                                    //18
            itsRight = new TreeNode (ob);                       //19
         else                                                   //20
            itsRight.add (ob, test);                            //21
      }                                                         //22
   }  //======================
```

If the executor of `removeFirst` has a TreeNode to its left (which therefore contains a higher-priority element than `this.itsData`), it can find the parent of the furthest-left Node and assign it to `p` (lines 5-7). It can then return the data value in `p.itsLeft`, since that data value has the highest priority of any. But first it must delete the Node `p.itsLeft` from the tree structure, replacing it by whatever was to the right of it (either null or another TreeNode). This coding is in the upper part of Listing 18.9.

The `add` method in the lower part of Listing 18.9 shows what a nonempty TreeNode object does when you ask it to `add` an element named `ob`: First it sees whether `ob` has higher priority than `itsData` (line 11). If so, `ob` goes to its left. If it does not have a TreeNode to its left (line 12), it can make a new one to hold `ob` (line 13), otherwise it can ask the TreeNode to its left to add `ob` (line 15). If `ob` goes to the right of the TreeNode executor, it works things out the same way but on the right instead of on the left. To accomplish all of this, the `add` method needs to have the Comparator object so it can make comparisons. So that Comparator object is passed as a parameter.

The TreeSort algorithm

The TreeNode implementation of a priority queue gives yet another way to sort an array of data: Add the values one at a time to a TreePriQue object until they are all in there. Then remove them one at a time; they will come out in order.

This **TreeSort** algorithm is essentially the QuickSort algorithm but with links rather than arrays. It executes about as fast as the QuickSort algorithm, but it takes up more space (one extra TreeNode of space for each data object means you need at least 3*N object references for N pieces of data). Both adding and removing are big-oh of log(N) operations on average for random sequences of data values. You can guarantee big-oh of log(N) as the worst-case behavior if you use a red-black or AVL tree, as described in Chapter Sixteen.

The coding in Listing 18.10 shows how the TreeSort algorithm could be written in the standard form used throughout Chapter Thirteen: You are to put the first `size` values of the array named `item` in ascending order using `compareTo`. It uses the traversal method from Listing 17.9. Obviously, it would be more efficient to have the last four lines transfer the data directly to the array instead of using a queue. This is left as an exercise.

Listing 18.10 Another sorting algorithm for the CompOp class

```
public static void treeSort (Comparable[] item, int size)
{  if (size <= 1)
      return;
   java.util.Comparator itsTest = new Ascendor();
   TreeNode root = new TreeNode (item[0]);
   for (int k = 1;  k < size;  k++)
      root.add (item[k], itsTest);  // coded in Listing 18.9
   NodeQueue queue = new NodeQueue();
   root.inorderTraverseLR (queue);  // coded in Listing 17.9
   for (int k = 0;  k < size;  k++)
      item[k] = (Comparable) queue.dequeue();
}  //======================
```

Historical Note Older programming languages that did not have the interface concept used "procedural parameters" instead. Specifically, you could write a method one of whose parameters was the `compare` method heading. The coding for your method called on `compare` as needed. A statement that called your method had to pass in a reference to the implementation of `compare` that it wanted your method to use. Java passes Comparator objects as parameters instead of passing `compare` methods.

Exercise 18.38 How would you code a `worstData` method for the TreeNode class to find the element with the lowest priority in a non-empty tree?
Exercise 18.39 (harder) It is often useful for debugging programs to have a method that prints all the values in a data structure. Write one for Listing 18.9 using `System.out.println`: Print all values in and below a given non-empty TreeNode in order of priority (left to right).
Exercise 18.40* Replace the last four lines of the `treeSort` method by coding that transfers the data values in the tree directly to the array. Have it call a recursive independent class method not in the TreeNode class. Write that method.
Exercise 18.41* Give an example of a sequence of data values that causes both `add` and `removeMin` to execute in big-oh of N time for a TreePriQue.
Exercise 18.42** Explain why TreePriQue is stable, and thus `treeSort` is a stable sorting algorithm, even though the `quickSort` it is based on is not.

18.7 Implementing Priority Queues Using Heaps; The HeapSort Algorithm

Adding one element to an ArrayOutPriQue or NodeOutPriQue object has a worst-case execution time that is big-oh of N, where N is the number of items already in the data structure. Removal is quite fast at big-oh of 1. For an ArrayInPriQue or NodeInPriQue object, adding has a worst-case execution time that is only big-oh of 1, but worst-case execution time for removal is big-oh of N.

This section shows you how to use a "heap" as the basis for a priority queue that executes very fast. Adding one value to the heap has a worst-case execution time that is big-oh of log(N), where N is the number of elements currently in the priority queue. And removal of one value from the heap also has a worst-case execution time that is big-oh of log(N). This is a great improvement over the InsertionSort and SelectionSort implementations discussed in the previous section, or even the TreeSort (which has average case big-oh of log(N) for both, but worst-case is big-oh of N for each).

Implementing a priority queue as a heap

The heap logic requires that you think of each component in an array as having two components as its "children". Specifically, the first component, `itItem[0]`, has the next two components at indexes 1 and 2 as its children. Those two have the next four, at indexes 3 through 6, as their children -- 3 and 4 are the children of 1, and 5 and 6 are the children of 2. Those four have the next eight components as their children -- 7 and 8 are the children of 3, 9 and 10 are the children of 4, 11 and 12 are the children of 5, etc.

Figure 18.8 shows this relationship as a sort of genealogy tree. You start with one "node" of the tree at the top, then draw two children below it, then two children below each of those, repeating for as many elements as you have to store. Then you number the top node 0, number the next level with the next two integers 1 and 2, number the third level with the next 4 integers (3 through 6), number the fourth level with the next 8 integers (7 through 14), etc.

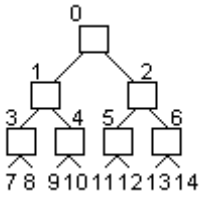


Figure 18.8 A tree

When you look at which numbers are "children" of which other numbers, you see that it can be expressed as a formula: The **children** of index n are indexes `2*n+1` and `2*n+2`. We also say that the elements at indexes `2*n+1` and `2*n+2` of the array are the children of the element at index `n`. Also, we say the element at index `n` is their parent; the formula for the **parent** of an index `k` is therefore `(k - 1) / 2`.

Our heap-based implementation **HeapPriQue** uses a partially-filled array (instance variables `Object[] itsItem` and `int itsSize`). You keep the array organized so that no child has higher priority than its parent. This is the **max-heap property**. That means that the data value with maximum priority is in `itsItem[0]`.

Adding a data value to the heap

Whenever you add another element to the priority queue whose data is already in `itsItem[0]` through `itsItem[itsSize-1]`, you start at `itsItem[itsSize]` and have the new data value "sift up" towards index 0. This means that you compare it with its parent (which would be at `itsItem[(itsSize-1)/2]`). If the parent does not have lower priority than the new value, put the new data value in `itsItem[itsSize]`. Otherwise move the parent down to that component, then compare the new data value with the parent of the parent. Repeat until you can insert the new data value in a way that maintains the max-heap property.

Figure 18.9 illustrates this process for the first nine values added. The values inside the components are integers, to simplify the description, though of course objects are normally used. Here we use larger integers to represent data with higher priorities. On each iteration (reading left to right on each level), one additional component is brought into compliance by swapping it up the tree until it is greater than all of its children. Data values are added in the order 13, 10, 16, 12, 11, 14, 17, 18, 15.

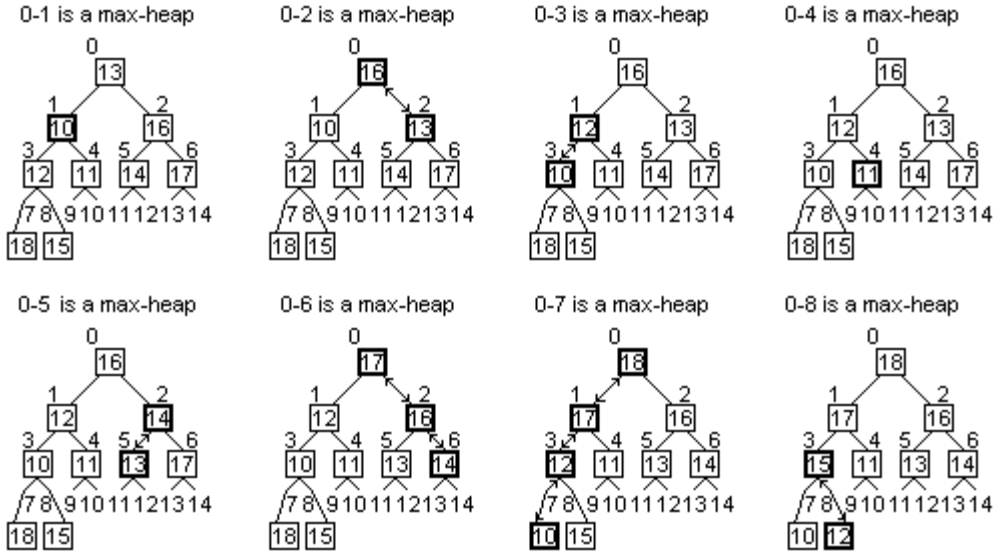


Figure 18.9 Adding values to the array, maintaining a max-heap

The coding in Listing 18.11 (see next page) embodies this logic for the `add` method. Compare it carefully with the coding for ArrayOutPriQue's `add` method in Listing 18.3. It is exactly the same except that `k - 1` has been replaced by `(k - 1) / 2` throughout. In other words, it looks like the `insertInOrder` logic except that each additional step is twice as close to index 0 as the step before instead of being just 1 component closer. For instance, when `add` is called for the element at index 127, it is inserted into the sequence of values at indexes 63, 31, 15, 7, 3, 1, and 0. That makes the `add` method a big-oh of log(N) process instead of big-oh of N. An exercise makes this coding more efficient though not as clear.

Removing a data value from the heap

Whenever you remove a data value from the priority queue, you remove it from `itsItem[0]`. Then you readjust the array to be a max-heap without the removed value. This coding is in the lower part of Listing 18.11. The readjusting of the array is left to a private method named siftDown.

To readjust the array, take `itsItem[itsSize-1]` from the rear part of the array and insert it somewhere in the rest of the array; we will call that value `toInsert`.

Removing the highest-priority data value from index 0 left an empty component. Compare its two children at indexes 1 and 2 to see which has higher priority. That child (call it `kid`) moves up to index 0, which leaves an empty spot where it was. You then compare the two children of that empty spot to see which has higher priority and move that one up to the empty spot, leaving another empty spot on the third level (at index 3, 4, 5, or 6). The moving-up part is coded as `itsItem[empty] = itsItem[kid]`.

Listing 18.11 The HeapPriQue class of objects, partially done

```
public class HeapPriQue implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private java.util.Comparator itsTest;


   // the two constructors and isEmpty are as for ArrayOutPriQue

   public Object peekMin()
   {  if (isEmpty())                                          //1
         throw new IllegalStateException ("priority Q is empty");
      return itsItem[0];                                      //3
   }  //======================

   public void add (Object ob)
   {  if (itsSize == itsItem.length)                          //4
      {  }  // left as an exercise in an earlier section     //5
      int k = itsSize;                                        //6
      while (k > 0 && itsTest.compare (ob,                    //7
                              itsItem[(k - 1) / 2]) < 0)      //8
      {  itsItem[k] = itsItem[(k - 1) / 2];                   //9
         k = (k - 1) / 2;                                     //10
      }                                                       //11
      itsItem[k] = ob;                                        //12
      itsSize++;                                              //13
   }  //======================

   public Object removeMin()
   {  if (isEmpty())                                          //14
         throw new IllegalStateException ("priority Q is empty");
      Object valueToReturn = itsItem[0];                      //16
      itsSize--;                                              //17
      if (itsSize >= 2)                                       //18
         siftDown (itsItem[itsSize]);                         //19
      else if (itsSize == 1)                                  //20
         itsItem[0] = itsItem[1];                             //21
      return valueToReturn;                                   //22
   }  //======================
}
```

Keep this up until either the empty spot does not have two children or else both of its children have lower priority than `toInsert` has. Then put `toInsert` in the empty spot and stop. This coding is in Listing 18.12 (see next page). For example, in the final tree on the bottom-right of Figure 18.9, removing 18 would require that 17 move to index 0 (since 17 is larger than its sibling 16), then 15 to index 1 (since 15 is larger than its sibling 11), then 12 to index 3 (since 12 is larger than its sibling 10). Figure 18.10 shows this application of `removeMin` and the next one as well (see next page).

Programming Style In Listing 18.12, some people would say that lines 8 and 9 should be replaced by a `break` statement. That would immediately terminate the loop and proceed with the statement at line 15. However, using a `break` statement to exit a loop generally makes coding harder to understand. In this case, the coding as shown actually executes faster without the `break` statement. You will not see the `break` statement used in this book. Any loop complex enough for a `break` statement is complex enough to put in a separate method where you use a `return` statement in place of a `break`.

Listing 18.12 The private siftDown method for HeapPriQue

```
/** Given that itsItem[0]..itsItem[itsSize] is a max-heap,
 *  in effect replace itsItem[0] by toInsert and then make
 *  the minimal changes to swap toInsert down so that
 *  itsItem[0]...itsItem[itsSize-1] is a max-heap again.  */

private void siftDown (Object toInsert)
{  int empty = 0;                                          //1
   int kid = 1;            // empty's child on the left   //2
   while (kid < itsSize)   // there are two children      //3
   {  if (itsTest.compare (itsItem[kid + 1],               //4
                           itsItem[kid]) < 0)              //5
         kid++;            // use the child on the right  //6
      if (itsTest.compare (toInsert, itsItem[kid]) < 0)    //7
      {  itsItem[empty] = toInsert;                        //8
         return;                                           //9
      }                                                    //10
      itsItem[empty] = itsItem[kid];                       //11
      empty = kid;                                         //12
      kid = 2 * empty + 1;  // empty's child on the left   //13
   }                                                       //14
   itsItem[empty] = toInsert;                              //15
}  //======================
```

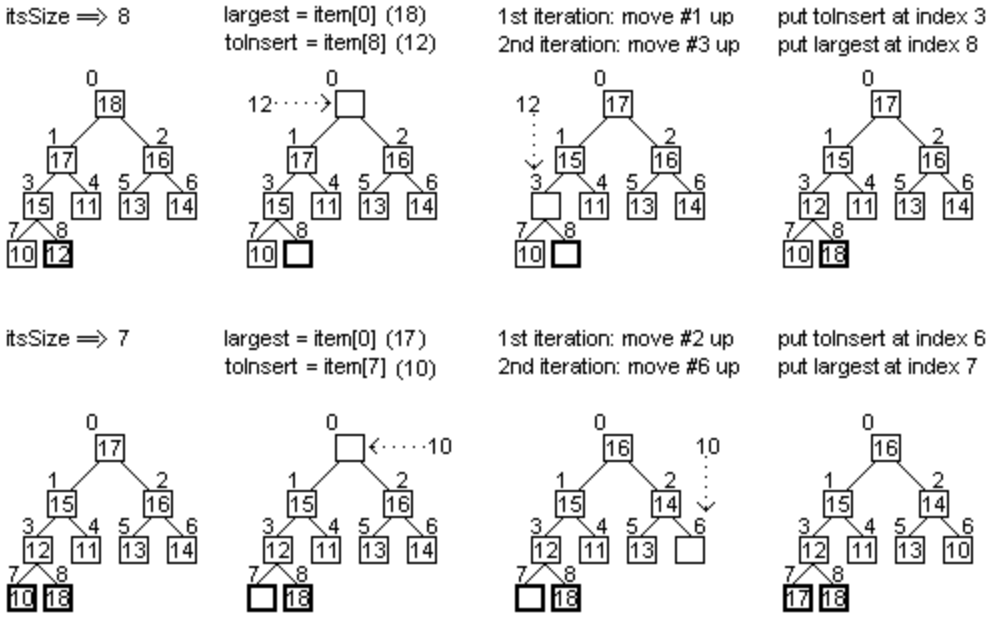


Figure 18.10 Two consecutive calls of the siftDown method

The HeapSort algorithm

A method to sort a given array of a given number of Comparable values using the **HeapSort** logic is as follows. It calls on a HeapPriQue constructor that assigns the given array to `itsItem`, then repeated applies the `add` method to create the initial heap. Finally, it repeatedly applies the `siftDown` method to get the data in ascending order. Writing the constructor itself is left as an exercise. Execution time is big-oh of N*log(N).

```
public static void heapSort (Comparable[] item, int size)
{  new HeapPriQue (item, size);
}  //======================
```

For the repeated application of `siftDown`, all of the values form a max-heap, from which it follows that `item[0]` must be the largest value of all. We want the largest value to end up in `item[size-1]`. So we swap `item[0]` with `item[size-1]`. But then what remains is not a max-heap any more, since `item[0]` is a relatively small value. So adjust the elements in `item[0]` through `item[size-2]` to be a max-heap. To do this, swap `item[0]` with whichever of its two children is larger, then swap it with the larger of that one's two children, etc., until it ends up where it should be.

At this point `item[0]` through `item[size-2]` is a max-heap, so we can repeat the whole process: Swap the second-largest element of all (now at `item[0]`) with the element at `item[size-2]`, so you have the two largest elements in the places they should be. Then adjust the rest of the array to be a max-heap again. Repeat as needed.

Note the similarity with the SelectionSort logic: We select the largest of the remaining values and swap it with the element in the component where that largest value goes. But the process of selecting the largest and readjusting the heap executes in big-oh of log(N) time rather than in big-oh of N time. The reason is that the adjusting to restore the heap condition jumps through the index values, doubling the size of the jump at each iteration. By contrast, the SelectionSort has to go through every single value.

Comparison with other sorting methods

As the preceding discussion shows, the HeapSort requires big-oh of N*log(N) time, even in the worst case. This is far better worst-case performance than the QuickSort logic, which can sometimes degenerate to big-oh of N^2 execution time. Of course, the MergeSort also has big-oh of N*log(N) worst-case execution time, but it requires an extra array for storage, which doubles the storage requirements. The HeapSort does not require any significant extra storage (two or three variables for `kid`, `empty`, etc.).

On the other hand, the HeapSort is the most complex of these three sorting methods, and it is the slowest in terms of average execution time. HeapSort is also difficult to understand, particularly the fact that, after you organize all the values into a heap, they are still not sorted. A sample run with 20 different random sets of 100,000 Double values gave average execution times of 2.69 seconds for HeapSort, 1.45 seconds for QuickSort (Listing 13.4), and 1.35 seconds for MergeSort (Listing 13.5). In fact, the MergeSort beat the QuickSort in 18 of the 20 runs.

Exercise 18.43 Rewrite the `add` method of Listing 18.11 so that `(k - 1) / 2` is only calculated once for each value of `k`. Use a local int variable named `parent`.
Exercise 18.44 Write out a complete trace of the execution of `heapSort` on the array of values {3, 7, 4, 6}.
Exercise 18.45 Explain why the coding would on average execute more slowly if the tests in lines 18 and 20 of the `removeMin` method were done in the opposite order.
Exercise 18.46 (harder) Count the maximum number of possible comparisons of elements that `heapSort` makes when `size` is 3, then count them when `size` is 7.
Exercise 18.47* Explain why the ArrayOutPriQue implementation of a priority queue keeps the highest-priority element at the largest index in the array, but the HeapPriQue implementation keeps it at index 0, the smallest index in the array.
Exercise 18.48* Show that, if you make a max-heap and then perform an InsertionSort on it, you still have big-oh of N^2 execution time for the worst case. Hint: What is the worst case for the order of the elements in the lower half of the array?

Exercise 18.49* A faster version of the `heapSort` has the first stage make `item[k]` through `item[size-1]` a max-heap for values of k decreasing from `size/2` to 0. Make the modifications in Listing 13.6 to do this. Explain why this process executes in big-oh of N time.
Exercise 18.50** Write the new HeapPriQue constructor called by the `heapSort` constructor. Hint: The Comparator object should be the reverse of Ascendor, so that the initial heap has the largest value in index 0. Then each value removed from the heap should go directly to the end of the array. The final result is then in ascending order.
Exercise 18.51** Determine whether HeapPriQue is stable and give sound reasons.

18.8 MergeSort For A Linked List; Recurrence Relations

This section provides you more practice with linked lists and recursion, as well as reviewing the merge sort logic. This review is preparatory to the next section, which discusses a good method for sorting data values in a very large file.

Most sorting algorithms for arrays can be coded for linked lists as well. Clearly, the logic in NodeOutPriQue can easily be shaped to provide a linked list InsertionSort and the logic in NodeInPriQue can easily be shaped to provide a linked list SelectionSort. We next develop the MergeSort for linked lists.

The easiest version of merge sort for a linked list is done with a header node. For instance, the HeaderList class at the end of Chapter Fourteen uses header nodes. That is, a HeaderList object has two instance variables, itsData and itsNext, where itsNext is a HeaderList object. But we do not store data in the first node on the list.

So a public instance method to sort such a list could be coded as follows. This coding passes the part of the list after the header node, as well as the number of data values in that list, to a recursive method named `sorted`. That method will use the header node `this` to store information during execution of the merging part of the algorithm:

```
public void sort()    // in HeaderList
{  itsNext = sorted (itsNext, size());
}  //======================
```

For the MergeSort logic, we are to divide the list into two lists of equal size (or differing by 1 if necessary), sort each one separately, and then merge them back together into one list sorted in ascending order. First, we return the linked list immediately if it has less than two elements, since it is already sorted. Otherwise, we run half-way down the list and set `end` to the Node at the half-way point (node number `size / 2`). The second half of the list starts in `end.itsNext`. We break the list into the two equal parts, call the `sorted` method for each part, and then call a private `merged` method to merge the two sorted lists into one long sorted list. This coding is in the upper part of Listing 18.13 (see next page). It could be part of the HeaderList class.

In this `merged` method, `one` and `two` denote the two sorted linked lists to be combined and placed on the completely sorted list. We add nodes from `one` and `two` to the rear of the list that begins with the header node `this`. When we exit the method, we return the first node after the header node `this`, which will be the first node in the completely sorted list.

Listing 18.13 The MergeSort logic for a linked list (specifically, in HeaderList)

```
public void sort()
{  itsNext = sorted (itsNext, size());                    //1
}  //======================

private HeaderList sorted (HeaderList item, int size)
{  if (size < 2)                                          //2
      return item;                                        //3
   int halfSize = size / 2;                               //4
   HeaderList end = item;                                 //5
   for (int k = 1;  k < halfSize;  k++)                   //6
      end = end.itsNext;                                  //7
   HeaderList secondHalf = end.itsNext;                   //8
   end.itsNext = null;                                    //9
   return merged (sorted (item, halfSize),                //10
                  sorted (secondHalf, size - halfSize));  //11
}  //======================

private HeaderList merged (HeaderList one, HeaderList two)
{  HeaderList rear = this;  // last node of sorted        //12
   while (one != null && two != null)                     //13
   {  if (((Comparable) one.itsData)                      //14
                     .compareTo (two.itsData) < 0)        //15
      {  rear.itsNext = one;                              //16
         one = one.itsNext;                               //17
      }                                                   //18
      else                                                //19
      {  rear.itsNext = two;                              //20
         two = two.itsNext;                               //21
      }                                                   //22
      rear = rear.itsNext;                                //23
   }                                                      //24
   rear.itsNext = (one == null)  ?  two  :  one;          //25
   return this.itsNext;                                   //26
}  //======================
```

So the `merged` logic repeatedly looks at the first node on each of `one` and `two`; whichever contains the smaller data is attached to the rear of the sorted list and detached from its own list (that is, detached from the `one` list or the `two` list). When one of the lists runs out of Nodes, the remainder of the other list is attached to the rear of the sorted list and the result is returned. This coding is in the lower part of Listing 18.13. Figure 18.11 shows how a list of eight data values would be sorted using this coding.

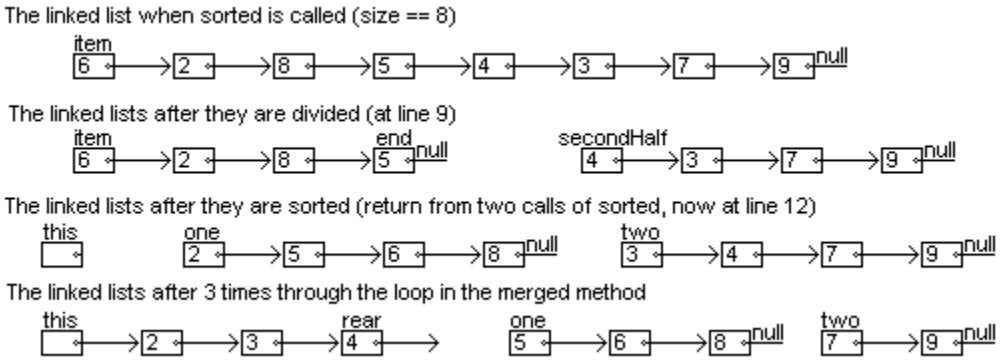


Figure 18.11 Stages in the execution of the merge sort

Recurrence relations

If you study the coding in Listing 18.13 carefully, you see that the maximum number of comparisons of data made by the MergeSort coding for n data values can be expressed by a recursive formula, written directly from the recursive coding. This is called a **recurrence relation**:

```
compsMS (n) = 0 if n <= 1, otherwise
compsMS (n) = compsMS (n/2) + compsMS (n - n/2) + (n - 1)
```

Any recurrence relation implies a corresponding recursive method for calculating the value. Call this method with any value of n to have the computer calculate it for you:

```
public static int compsMS (int n)  // for MergeSort
{  return n <= 1  ?  0
          : compsMS (n / 2) + compsMS (n - n / 2) + (n - 1);
}  //======================
```

Four more recurrence relations

The TreeSort coding towards the end of Section 18.6 adds data values one at a time to a binary search tree and then traverses the tree to obtain the values in order. When is the minimum number of comparisons for all the additions made? When half of the data values compared with the data in any given TreeNode go to the left and the rest go to the right. In that case, the number of comparisons made in building a tree of n nodes would be expressed by the following recurrence relation. Note that the first data value goes in the root and each data value thereafter is compared with the root, which makes n-1 root comparisons:

```
compsTS (n) = 0 if n <= 1, otherwise
compsTS (n) = compsTS (n/2) + compsTS (n - n/2) + (n - 1)
```

This is of course the same recurrence relation for compsMS, so the best case for TreeSort is the same as the worst case for MergeSort.

The InsertionSort repeatedly inserts a new value in an existing list of values where it goes in ascending order. So the maximum number of comparisons made can be expressed by the following recurrence relation (this is also the recurrence relation for SelectionSort):

```
compsIS (n) = 0 if n <= 1, otherwise
compsIS (n) = compsIS (n - 1) + (n - 1)
```

The HeapSort repeatedly inserts a new value in a heap of values to maintain the max-heap property. For instance, inserting the 64th value (at index 63) will require comparisons with at most the values at indexes 31, 15, 7, 3, 1, and 0. That is, the number of comparisons is at most log2(n). So an upper limit on the number of comparisons made during the insertion process that builds the heap can be expressed by the following recurrence relation:

```
compsHSBuild (n) = 0 if n <= 1, otherwise
compsHSBuild (n) = compsHSBuild (n - 1) + log2(n)
```

If you do the HeapSort directly on n data values without using a priority queue, you can build the initial heap faster. Define the "subheap at index k" to be the value at index k plus its two children plus their children, etc., going no further than n of course. Then you build the subheap at a given index k only after you have built the subheap for both of its children. A bit of thought and drawing pictures expresses the number of comparisons required by the following recurrence relation, at least for any n that is 1 less than a power of 2 (i.e., n being one of 1, 3, 7, 15, 31, 63, etc.):

```
compsHSBuild2 (n) = 0 if n <= 1, otherwise
compsHSBuild2 (n) = 2 * (compsHSBuild (n/2) + log2((n + 1)/2))
```

Closed forms of recurrence relations

We would like to have an upper bound on the value of compsXX(n) as a function of n, without the recursive call of the function. This is called a **closed form**. We can prove by induction that compsMS is bounded above by n*log2(n) (and thus so is compsTS, since they have the same recurrence relation). We use here the facts that log2(2) is 1, log2(1) is 0, and log2(n/x) = log2(n) - log2(x):

*To Prove compsMS(n) <= n * log2(n) for any n >= 1 that is a power of 2.*
Basic Step when n = 1: compsMS(n) is zero, which is less-equal 1 * log2(1).
Inductive Step To show that compsMS(n) <= n * log2(n) in situations where n > 1 and we know the relation is true for all smaller powers of 2, we reason as follows:

```
compsMS(n) = 2 * compsMS(n/2) + (n - 1)     // combines the first two terms
           <= 2 * (n/2 * log2(n/2)) + (n - 1)   // since it is true for smaller powers of 2
            = n * (log2(n/2))          + (n - 1)
            = n * (log2(n) - log2(2)) + (n - 1)
            = n * log2(n) - n * 1 + n - 1 = n * log2(n) - 1 < n * log2(n)
```

Conclusion The truth of the assertion for all positive powers of 2 follows by the Induction Principle from the Basis Step and the Inductive Step.

We can also prove by induction that compsHSBuild is bounded above by n*log2(n):

*To Prove compsHSBuild(n) <= n * log2(n) for any n >= 1.*
Basic Step when n = 1: compsHSBuild(n) is zero, which is less-equal 1 * log2(1).
Inductive Step To show that compsHSBuild(n) <= n * log2(n) in situations where n > 1 and we know the relation is true for all smaller values of n, we reason as follows:

```
compsHSBuild(n) = compsHSBuild(n-1) + log2(n)  // known recurrence relation
                <= (n-1) * log2(n-1) + log2(n)   // since it is true for smaller values of n
                <= (n-1) * log2(n) + log2(n)
                 = n * log2(n)
```

Conclusion The truth of the assertion for all positive n follows by the Induction Principle from the Basis Step and the Inductive Step.

The proofs that compsIS(n) <= $(n-1)^2$ and that compsHSBuild2(n) < 2*n are left as exercises. The latter fact shows that this way of building a heap executes in big-oh of N time.

Exercise 18.52 (harder) Rewrite the `merged` method to not use a header node or create any extra nodes.
Exercise 18.53** Revise the `merged` method on the precondition that `one`'s first Node contains the smallest data value. Run down the `one` list, inserting Nodes from `two`'s list wherever appropriate, then return the `one` list. Rewrite the `sorted` method to use this new `merged` method by calling it from two different places in the coding. Is this a faster implementation of the MergeSort logic?
Exercise 18.54** Prove that compsIS(n) <= $(n-1)^2$ for all positive values of n.
Exercise 18.55** Prove that compsHSBuild2(n) = 2*n - log(n+1)+1 for all positive numbers that are 1 less than a power of 2.
Exercise 18.56** Write a recurrence relation for the number of comparisons required for binary search in an array and prove inductively a good upper limit on them.

18.9 External Sorting: File Merge Using A Priority Queue

Sometimes we have so much data to put in sorted order that we cannot fit it all into RAM. Say it is stored in a hard-disk file and there are 2 million records to sort (a **record** is the set of current values of instance variables of an object). Our problem is to produce a new hard-disk file (we will call it SORTED.DAT) that contains all of those records in ascending order of IDs.

The piles-of-files algorithm

It is desirable to do as much of the sorting as possible in RAM, so a reasonable first step is to read as many values as we can into RAM, sort them, and write them out to a file. Then we read some more and write those to another file, etc. For instance, if we can handle 10,000 records at a time in RAM, we could end up with our original 2 million records in 200 different files. Then we could merge them together into one large file as shown in the accompanying design block (`numFiles` is 200 for this example).

STRUCTURED NATURAL LANGUAGE DESIGN for merging 200 files

1. Read the first record from each file into the corresponding component of an array of `numFiles` Comparable objects. Call the array `item`.
2. Find the smallest of those array components. Say it is at index `k`.
3. Write `item[k]` to the SORTED.DAT file.
4. If file number `k` is now empty, delete its entry from the array, otherwise get another value from the corresponding file `k` to go in `item[k]`.
5. Repeat steps 2 through 4 until done.

The total execution time for this algorithm is what is required for:
(a) the internal sorting, 200 groups of 10,000, plus
(b) reading and writing 2 million records 2 times each, and also
(c) making 199 comparisons 2 million times for a total of 398 million comparisons.
That last calculation generalizes to `N * (N / 10000)` for N records, so overall this is a big-oh of N-squared algorithm.

Object design

This design block is far too specific. We should not be making a commitment to unsorted arrays of values at this point in the design, or to a specific kind of file. To start with, we should just require some kind of file object that can provide the next object in a sequential file when asked or accept a new object to add to the end of the file. A reasonable object design is the following:

```
public class ObjectFile  // for generic files of objects
{   // Open the file of that name for output; open a temporary file if the name is "".
    public ObjectFile (String name)                 {  }
    // Add ob to the end of the file.
    public void writeObject (Object ob)             {  }
    // Change over to providing input, not output.
    public void openForInput()                      {  }
    // Retrieve the next available object in the file.
    public Object readObject()          { return null; }
    // Switch back to providing output, not input.
    public void openForOutput()                     {  }
    // Tell whether the input file has no more values.
    public boolean isEmpty()           { return false; }
}
```

We can then leave the details of how this is done in terms of the Sun standard library for the coding of this class's methods. You might want to look into Sun's ObjectInputStream class and serialization for a good way to implement this class.

To sort the data 10,000 units at a time (or whatever is appropriate), we basically need an object that provides two services: It can read 10,000 or so values from a given ObjectFile that has been opened for input; and it can write those 10,000 or so values in order to a given ObjectFile that has been opened for output. It could be as follows:

```
public class ObjectFileSorter  // for sorters of ObjectFiles
{   // Create the object capable of holding max values.
    public ObjectFileSorter (int max)                  {  }
    // Read max values from the given file, except stop reading at the end of the file.
    public void readManyFromFile (ObjectFile file)     {  }
    // Write all values you have to the given file in increasing order using compareTo.
    public void writeManyToFile (ObjectFile file)      {  }
}
```

The upper part of Listing 18.14 (see next page) shows the structure of the object that converts one very large unsorted file to a very large sorted file. For the constructor, you supply the names of the unsorted file and the sorted file. Both are initially open for output (according to the specifications for the ObjectFile class), so you have to open the unsorted file for input.

The `makeSortedFiles` method in the middle part of Listing 18.14 creates an ObjectFileSorter object from a numeric parameter that tells how many objects you want to have in the sorter at one time. Then it repeatedly reads 10,000 (or whatever) values from the unsorted file, writes them in sorted order to a temporary output file, and adds the temporary file along with its first value to a data structure named `itsData`.

For the process of merging 200 files (or however many we have), we need a kind of data structure that can store pairs consisting of a file plus the next available value from that file. When we get a value from this object, we want to receive the pair with the smallest value using the `compareTo` method. A priority queue class is acceptable for this purpose.

A priority queue can also do most of the task required of an ObjectFileSorter. That is, `readManyFromFile` can repeatedly read a data value and add it to a priority queue. And `writeManyToFile` can repeatedly call `removeMin` for that priority queue and write the result to the file.

Using only four files

A prime difficulty with this piles-of-files algorithm is that many systems do not allow you to keep more than a dozen or so files open at any one time. So we next present a method that only uses four files.

Stage 1 (for the `makeSortedFiles` method) Write the sorted groups of 10,000 (or whatever) alternately to just 2 files. That is, the first group goes into file `one`, the second into file `two`, the third into file `one` again, the fourth into file `two` again, the fifth into file `one`, the sixth into file `two`, etc. So for the example of 2 million data values, you end up with 100 groups of 10,000 in each of the 2 files.

Listing 18.14 File merge with piles of files

```
import ObjectFile;
import ObjectFileSorter;
import PriQue;

public class ManyFilesMerger
{
   private ObjectFile itsInFile;  // the original unsorted input
   private ObjectFile itsOutFile; // the final sorted output
   private HeapPriQue itsData = new HeapPriQue();


   public ManyFilesMerger (String inf, String outf)
   {  itsInFile = new ObjectFile (inf);   // for output      //1
      itsInFile.openForInput();        // but we need input  //2
      itsOutFile = new ObjectFile (outf); // for output      //3
   } //============================================

   /** Step 1:  Make many files, each sorted. */

   public void makeSortedFiles (int maxToSort)
   {  ObjectFileSorter sorter = new ObjectFileSorter (maxToSort);
      while ( ! itsInFile.isEmpty())                         //5
      {  ObjectFile tempFile = new ObjectFile ("");          //6
         sorter.readManyFromFile (itsInFile);                //7
         sorter.writeManyToFile (tempFile);                  //8
         tempFile.openForInput();                            //9
         itsData.add (new Pair (tempFile.readObject(),tempFile));
      }                                                      //11
   } //============================================

   /** Step 2:  Merge the many files into just one sorted file.*/

   public void mergeFiles()
   {  while ( ! itsData.isEmpty())                           //12
      {  Pair p = (Pair) itsData.removeMin();                //13
         itsOutFile.writeObject (p.itsData);                 //14
         if (! p.itsFile.isEmpty())                          //15
         {  p.itsData = p.itsFile.readObject();              //16
            itsData.add (p);                                 //17
         }                                                   //18
      }                                                      //19
   } //============================================

   private static class Pair implements Comparable
   {  public Object itsData;
      public final ObjectFile itsFile;

      public Pair (Object data, ObjectFile file)
      {  itsData = data;                                     //20
         itsFile = file;                                     //21
      }

      public int compareTo (Object ob)
      {  return ((Comparable) this.itsData).compareTo        //22
                     (((Pair) ob).itsData);                  //23
      }
   } //============================================
}
```

It will be difficult to make use of this data unless you can tell where one group of 10,000 ends and the next begins. So you need a special object value, different from any other, that you can use to mark the boundary between groups. Call this value `itsSentinel`. You will also find it convenient to have exactly the same number of groups in each of the two files; so if the last group goes in file `one`, just write `itsSentinel` again to `two`. That makes it an empty group of sorted values. The implementation for this part of the algorithm is in the upper and middle parts of Listing 18.15 (see next page).

Stage 2 (for the `mergeFiles` method) is the merging process:

1. Open files `one` and `two` for input and create two additional files `out1` and `out2` for output.
2. Combine the first 10,000 from file `one` with the first 10,000 from file `two` and write the resulting sorted group of 20,000 to file `out1`. This requires less than 20,000 comparisons using the standard merging logic.
3. Combine the second 10,000 from file `one` with the second 10,000 from file `two` and write the resulting sorted group of 20,000 to file `out2`. Again, you only need less than 20,000 comparisons.
4. Repeat steps 2 and 3 alternately until files `one` and `two` are empty. Now you have 50 groups of 20,000 in each of the two files `out1` and `out2`.
5. Open files `one` and `two` for output and files `out1` and `out2` for input.
6. Combine the first 20,000 from file `out1` with the first 20,000 from file `out2`; write those 40,000 to `one`.
7. Combine the second 20,000 from file `out1` with the second 20,000 from file `out2`; write them to file `two`.
8. Repeat steps 6 and 7 alternately until files `out1` and `out2` are empty. Now you have 25 groups of 40,000 in each of the files `one` and `two`.

Surely you can see where this is going. After the two passes through the data described above, you have six more passes to get it down to one completely sorted file of 2 million in just one file. You then write all of its values to the output file (except for the sentinel value at the end, of course). Note that you will occasionally get a group left over that has no group to be merged with (when you have an odd number of groups), in which case you just copy it into the appropriate file. The bottom part of Listing 18.15 contains this logic, except that the key merging logic is left as an exercise.

The total execution time for this algorithm is what is required for:
(a) the internal sorting, 200 groups of 10,000, plus
(b) reading and writing 2 million records 9 times each, and also
(c) making 2 million comparisons 8 times each for a total of 16 million comparisons.
The 9 and the 8 in this analysis are log2 (200) + 1 and log2(200). This generalizes to `N * log2(N / 10000)` for N records, so overall this is a big-oh of N * log(N) algorithm. But in real-life situations, it is slower than the piles-of-files method described first, since reading and writing disk records is excruciatingly slow.

Exercise 18.57 The ManyFilesMerger object in Listing 18.14 expects that a client class will call first the `makeSortedFiles` method and, directly after that, the `mergeFiles` method. What happens if a client class calls `mergeFiles` first?
Exercise 18.58 (harder) Still referring to the situation in the preceding exercise, what happens if a client class calls `makeSortedFiles` twice in a row without calling `mergeFiles`?
Exercise 18.59* Modify Listing 18.15 so that no client class can call `mergeFiles` before it calls `makeSortedFiles` and no client class can call either of those methods twice in a row. Hint: Add a boolean instance variable that tells whether `makeSortedFiles` has been called without an immediately following call of `mergeFiles`; use it appropriately.
Exercise 18.60*** Write the recursive coding for `mergeToOneFile` in Listing 18.15.

Listing 18.15 File merge with just four files

```
import ObjectFile;
import ObjectFileSorter;
import PriQue;

public class FourFilesMerger
{
   private ObjectFile itsInFile;  // the original unsorted input
   private ObjectFile itsOutFile; // the final sorted output
   private ObjectFile one, two;   // two scratch files
   private Object itsSentinel; // sentinel value to mark the end


   public FourFilesMerger (String inf, String outf, Object sent)
   {  itsInFile = new ObjectFile (inf);   // for output     //1
      itsInFile.openForInput();       // but we need input  //2
      itsOutFile = new ObjectFile (outf); // for output     //3
      itsSentinel = sent;                                   //4
   } //===============================================


   public void makeSortedFiles (int maxToSort)
   {  ObjectFileSorter sorter = new ObjectFileSorter (maxToSort);
      one = new ObjectFile ("");          // for output     //6
      two = new ObjectFile ("");          // for output     //7
      while ( ! itsInFile.isEmpty())                        //8
      {  sorter.readManyFromFile (itsInFile);               //9
         sorter.writeManyToFile (one);                      //10
         one.writeObject (itsSentinel);                     //11
         if ( ! itsInFile.isEmpty())                        //12
         {  sorter.readManyFromFile (itsInFile);            //13
            sorter.writeManyToFile (two);                   //14
         }                                                  //15
         two.writeObject (itsSentinel);                     //16
      }                                                     //17
   } //===============================================


   public void mergeFiles()
   {  if (one != null && ! one.isEmpty())                   //18
      {  one = mergeToOneFile (one, two,                    //19
               new ObjectFile (""), new ObjectFile (""));   //20
         Object data = one.readObject();                    //21
         while ( ! one.isEmpty())                           //22
         {  itsOutFile.writeObject (data);                  //23
            data = one.readObject();                        //24
         }                                                  //25
      }                                                     //26
   } //===============================================


   /** Return a file containing all the values in increasing
    *  order, plus a sentinel at the end. */

   private ObjectFile mergeToOneFile (ObjectFile in1,
           ObjectFile in2, ObjectFile out1, ObjectFile out2)
   {  return null;  // left as an exercise
   } //===============================================
}
```

18.10 Review Of Chapter Eighteen

- Stacks, queues, and priority queues are data structures that allow you to add an element, remove a particular element, see if they are empty, or see what you would get if you removed an element. The particular element you get depends on the structure: A **priority queue** gives you the element of highest priority (in case of a tie, the element that has been there longest). The **PriQue** interface has the methods `isEmpty()`, `put(ob)`, `removeMin()`, and `peekMin()`.
- A **java.util.Comparator** object has a method `compare(Object, Object)` that returns an int with the same meaning as for the standard `compareTo` method. This kind of functor object gives you full flexibility in choosing how things are prioritized.
- A priority queue implementation is **stable** if, in case of ties in priority, the element that has been in the data structure the longest is always removed first.
- This chapter presented several implementations of PriQue: ArrayOutPriQue and NodeOutPriQue (based on the InsertionSort), ArrayInPriQue and NodeInPriQue (based on the SelectionSort), NodeGroupPriQue (when there are very few different priority levels), TreePriQue (based on the QuickSort or its equivalent TreeSort), and HeapPriQue (based on the HeapSort logic).
- A **HeapSort** arranges the elements in an array so each element is greater than or equal to its two "children", then repeatedly moves the first element out and readjusts the heap structure. The children of index k are index 2*k+1 and 2*k+2.

Answers to Selected Exercises

18.1
```
public static void transfer (PriQue one, PriQue two)
{   while ( ! one.isEmpty())
        two.add (one.removeMin());
}
```
18.2 The third iteration combines the trees with 9 and 10 in their roots to obtain a tree with 19 in its root. This 19-node would be inserted between 16 and 22. Its left subtree would be a frequency of 9 with "gun" in its right subtree and the existing "peace"-4-"love" subtree as its left subtree.

18.3
```
public void combineHuffmanly (TreeNode par)  // in the TreeNode class
{   par.itsRight = this.itsLeft;
    this.itsLeft = par;
    this.itsData = new Integer (((Integer) this.itsData).intValue() + ((Integer) par.itsData).intValue());
}
```
18.7
```
Use the following Comparator class:
public class ByPosition implements java.util.Comparator
{   public int compare (Object one, Object two)
    {   RectangularShape a = (RectangularShape) one;
        RectangularShape b = (RectangularShape) two;
        return a.getY() != b.getY()  ?  a.getY() - b.getY()  :  b.getX() - a.getX();
    }
}
```
18.10
```
Object[] toDiscard = itsItem;
itsItem = new Object [itsItem.length * 2];
for (int k = 0;  k < itsSize;  k++)
    itsItem[k] = toDiscard[k];
```
18.11
```
public void add (Object ob)
if (itsSize == itsItem.length)
{ }  // same as for the preceding exercise
itsItem[itsSize] = ob;
itsSize++;
```
18.12 Replace the next-to-last statement in the removeMin method by the following:
```
for (int k = best;  k < itsSize;  k++)
    itsItem[k] = itsItem[k + 1];
```
18.17
```
public Object peekMin()
{   if (isEmpty())
        throw new IllegalStateException ("priority Q is empty");
    return itsFirst.itsData;
}
```
18.18 Replace lines 12-14 of NodeInPriQue's removeMin method by the following two:
```
best.itsData = best.itsNext.itsData;
best.itsNext = best.itsNext.itsData;
```

18.19 In the add method in Listing 18.5, replace the while-loop by the following four lines:

```
while (p.itsNext != null && itsTest.compare (ob, p.itsData) > 0)  // note change in the second condition
      p = p.itsNext;
if (p.itsNext != null && itsTest.compare (ob, p.itsData) == 0)
      return;
```

18.20
```
public int size()
{     int count = 0;
      for (Node p = itsFirst;  p.itsNext != null;  p = p.itsNext)
            count++;
      return count;
}
```

18.21
```
public String toString()
{     String valueToReturn = "";
      for (Node p = itsFirst;  p.itsNext != null;  p = p.itsNext)
            valueToReturn += '\t' + p.itsData.toString();
      return valueToReturn;
}
```

18.22
```
public void removeAbove (Object ob)   // in NodeOutPriQue
{     while (itsFirst.itsNext != null && itsTest.compare (itsFirst.itsData, ob) < 0)
            itsFirst = itsFirst.itsNext;
}
```

18.23
```
public void removeAbove (Object ob)   // in NodeInPriQue
{     while (itsFirst.itsNext != null && itsTest.compare (itsFirst.itsData, ob) < 0)  // check the first one
            itsFirst = itsFirst.itsNext;
      if (itsFirst.itsNext != null)
      {     Node p = itsFirst;
            while (p.itsNext.itsNext != null)   // so the next node has data to be compared
                  if (itsTest.compare (p.itsNext.itsData, ob) < 0)
                        p.itsNext = p.itsNext.itsNext;
                  else
                        p = p.itsNext;
      }
}
```

18.24
```
public void add (Object ob)  // revision for NodeInPriQue
{     if ( ! isEmpty() && itsTest.compare (ob, itsFirst.itsData) >= 0)
            itsFirst.itsNext = new Node (ob, itsFirst.itsNext);
      else
            itsFirst = new Node (ob, itsFirst);
}
```

18.35 In peekMin write: return ((QueueADT) itsFirst.itsData).peekFront();
In removeMin write: Object valueToReturn = ((QueueADT) itsFirst.itsData).dequeue();
Also in removeMin write: if (((QueueADT) itsFirst.itsData).isEmpty())
In add replace "p.itsData." by "((QueueADT) p.itsData)." in all three places.

18.38
```
public Object worstData()
{     return itsRight == EX  ?  itsData  :  itsRight.worstData();
}
```

18.39
```
public void printData()
{     if (isEmpty())
            return;
      itsLeft.printData();
      System.out.println (itsData.toString());
      itsRight.printData();
}
```

18.43 Just before the while in line 7, define: int parent = (k - 1) / 2;
Replace the phrase (k - 1) / 2 by parent in three places: lines 8, 9, and 10.
Just before the end of the loop body at line 9, update: parent = (k - 1) / 2.

18.44 The add method swaps 3 and 7, leaves 4 where it is, then swaps 6 and 3. Result: {7, 6, 4, 3}.
The siftDown method swaps 7 and 3, then swaps 3 with 6. Result: {6, 3, 4, 7}. The next call of siftDown swaps 6 and 4. The next call of siftDown swaps 4 and 3. Final result: {3, 4, 6, 7}.

18.45 Currently, itsSize >= 2 requires only 1 test, but itsSize < 2 requires 2 tests (at both lines 18 and 20).
The swap would require 2 tests except when itsSize==1, which happens less often than itsSize >= 2.

18.46 3 when size is 3: 2 for add, 1 for siftDown. 25 when size is 7: 2*1+4*2 for the two levels of add, then (4*4-2)+1 for siftDown.

18.52 Change the last statement to be return result. Then insert the following at the beginning:

```
HeaderList result = ((Comparable) one.itsData).compareTo (two.itsData) < 0  ?  one  :  two;
if (result == one)
      one = one.itsNext;
else
      two = two.itsNext;
```

18.57 Nothing happens if mergeFiles is called first, since itsData is empty until after makeSortedFiles executes.

18.58 In effect, nothing happens if makeSortedFiles is called twice. True, on the second time another new ObjectFileSorter object is created. But itsInfile is empty (caused by the preceding call of this method), so the method terminates immediately and the newly-created object is garbage collected.

19 Graphs

Overview

This chapter covers the concepts of graphs and networks. It uses object classes developed in earlier chapters: stacks, queues, priority queues, and collections.

- Section 19.1 describes the general abstraction of a graph and an application of graphs.
- Sections 19.2-19.4 present two different standard implementations of graphs, as well as a basic algorithm for a topological sort of the vertices of a graph.
- Section 19.5 discusses traversals of graphs, both depth-first and breadth-first. It solves the classic Traveling Salesman problem in graph theory.
- Sections 19.6-19.8 introduce the concept of a weighted graph (a network) and develop several standard algorithms for them: Kruskal's, Prim's, and Dijkstra's. Kruskal's algorithm requires UnionFind algorithms, so they are included here.
- Section 19.9 is just a bit of dynamic programming to finish the chapter up nicely.

19.1 The Hamiltonian Software

You have been hired to develop a game named Hamiltonian, a thinking game for children. It displays 20 to 25 small graphic images on the screen (we call these the **vertices**) and a large number of lines, each line connecting two of the vertices (we call these the **edges**). Each time the game is played, the number of vertices and the particular edges drawn change. The game is most interesting when each vertex has an average of 3 to 4 edges drawn to it.

One of the vertices is labeled END. The player clicks on a vertex connected (by an edge) to the END vertex, then clicks on a second vertex connected to the vertex chosen first, then clicks on a third vertex connected to the vertex chosen second, etc., until the player chooses to end play by clicking on END. Each vertex is green originally, but becomes red when clicked. A "bad click" is a vertex previously clicked or a vertex (other than END) not connected to the one previously clicked (or to END if it is the first vertex clicked). The bad clicks are ignored in the play. The score is 3 times the number of vertices clicked minus the number of elapsed seconds minus the number of bad clicks, plus a 20 point bonus if all of the vertices are clicked (forming what is called a **Hamiltonian path**), plus another 10 point bonus if the vertex clicked just before clicking END is connected to END(forming what is called a **Hamiltonian circuit**).

To plan this game software, we concentrate here on the graph-handling methods needed. A **graph** is a set of vertices and edges. An edge is a connection from one vertex to another. A **path** in the graph is a sequence of edges where the vertex at the end of each edge is the vertex at the beginning of the next edge. A graph is **connected** if you can get from any one vertex to any other vertex by following edges in the graph, i.e., there is a path from any one vertex to any other vertex.

The client wants the game software to first create a graph by choosing edges at random until you have enough for the game, eliminating choices that make the graph connected (so the game will not be too easy). Once you have enough edges, you then add one edge at a time until the graph becomes connected. So you need a way of testing the graph to see if it is connected yet, and you need to be able to tell whether two given vertices are connected to each other. The methods in the **HamGraph** class of Listing 19.1 (see next page) seem appropriate for a start. Figure 19.1 shows a possible game layout, with 20 vertices and 30 edges.

Listing 19.1 The HamGraph class of objects, stubbed partial documentation

```
public class HamGraph
{
   /** Create a graph on n vertices with at least 1.5 * n
    *  randomly chosen edges, but the graph cannot be connected
    *  (it must have at least two components). */

   public void createUnConnectedGraph (int n)
   {
   } //===========================================

   /** Add 1 randomly chosen edge at a time until this graph
    *  becomes connected.  */

   public void makeTheGraphConnected()
   {
   } //===========================================

   /** Tell whether this graph is connected. */

   public boolean isConnected()
   {  return false;
   } //===========================================
}
```

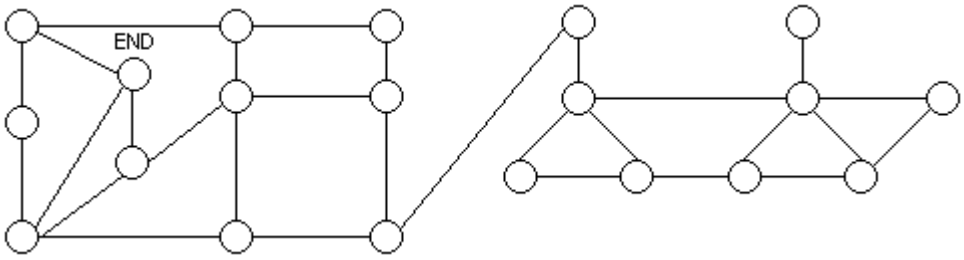


Figure 19.1 Possible game layout for the Hamiltonian software

Generic Graphs

The methods in the HamGraph class are specific to this game problem. We will be using graphs to solve various problems in this chapter, so we need to develop a more general definition of graphs, analogous to the Sun standard library Collection and Map types. Since Sun does not supply one, we develop it ourselves in this section. We will then use this Graph class in the HamGraph class.

A graph has a specific positive number of vertices, numbered consecutively with positive integer id numbers from 1 on up. We need to be able to create a graph, initially with no vertices, and ask a graph how many vertices it has, so the **Graph** class should have the following methods:

```
   public Graph()  // no edges, no vertices
   public final int getNumVertices() // tells number of vertices
```

We also need a class of Vertex objects to go with these Graph objects. The Graph should allow us to add a Vertex object with a given id number if no Vertex already has that number, to get access to the Vertex object with a given id number, and to create an Iterator object that goes through the collection of Vertex objects.

The Graph class with the methods described so far is in Listing 19.2. Note that three of these Graph methods are final methods, which means that subclasses cannot override them. So calls of these methods execute faster due to early binding. By contrast, some other Graph methods will be abstract, which means that subclasses must override them. Overriding `addVertex` is optional in subclasses of Graph.

Listing 19.2 The Graph class of objects with edge-related methods postponed

```
import java.util.ArrayList;
import java.util.Iterator;

public abstract class Graph
{
   private ArrayList itsVertices = new ArrayList();


   /** Create a graph with zero vertices and zero edges.  The
    *  vertices always have ids ranging 1...getNumVertices(). */

   public Graph()
   {  super();   // to remind you of the default constructor
   } //===========================================


   /** Return the current number of vertices. */

   public final int getNumVertices()
   {  return itsVertices.size();
   } //===========================================


   /** Return the Vertex object for the given id.  Throw a
    *  RuntimeException if the id is not 1...getNumVertices(). */

   public final Vertex getVertex (int id)
   {  return (Vertex) itsVertices.get (id - 1);
   } //===========================================


   /** Return an iteration over all Vertex objects. */

   public final Iterator vertices()
   {  return itsVertices.iterator();
   } //===========================================


   /** Add the given vertex to the graph if it has no vertex with
    *  the same id number; ignore non-positive ids.  Add vertices
    *  as needed for all positive ints below id, with null data.
    *  Throw a RuntimeException if the given vertex is null. */

   public void addVertex (Vertex v)
   {  if (v.ID > itsVertices.size())
      {  for (int k = itsVertices.size() + 1;  k < v.ID;  k++)
            itsVertices.add (new Vertex (k, null));
         itsVertices.add (v);  // added at the end of the list
      }
   } //===========================================
}
```

Reminder An **ArrayList**, discussed in some detail in Section 7.11, is a Sun standard library implementation of Collection, so it has the `size`, `add`, `contains`, `remove`, and `iterator` methods that any Collection has, along with the following additional array-like methods (this is all you need to know about ArrayLists to understand this chapter):

```
public set (int n, Object ob)  // replace at component n
public get (int n)   // return the object at component n
```

Edges in the Graph class

If a graph has only vertices and no edges, it is not of much use. We need to have some Edge objects as well. A graph should be able to add an Edge running from one Vertex to another Vertex, to tell whether there already exists an Edge running from one Vertex to another, to remove a given Edge, and to clear out all existing Edges. With these methods, you can easily implement HamGraph as an extension of some concrete implementation of the Graph class. For instance, add the following private method to HamGraph to make coding `createUnConnectedGraph()` easy. Assume that you will have an Edge constructor with two Vertex parameters:

```
/** Add a randomly chosen edge not already in the graph. */

private void addEdgeChosenRandomly()  // in HamGraph
{  int one = 1 + (int) (Math.random() * getNumVertices());
   int two = 1 + (int) (Math.random() * getNumVertices());
   Edge e = new Edge (getVertex (one), getVertex (two));
   if (this.contains (e))
      addEdgeChosenRandomly();
   else
   {  this.add (e);
      this.add (new Edge (getVertex (two), getVertex (one)));
      // in the opposite direction, for an undirected graph
   }
} //===============================================
```

Note that the `addEdgeChosenRandomly` method always adds a pair of Edges, one in each direction between `one` and `two`. That is because the game software calls for an **undirected graph**, i.e., a graph where there is an Edge from x to y if and only if there is an Edge from y to x. If a graph does not have this property, it is a **directed graph**. If a graph is undirected, then the **out-degree** of each vertex (defined to be the number of Edges exiting the vertex) always equals the **in-degree** of that vertex (defined to be the number of Edges coming in to that vertex).

The following method is a horribly inefficient algorithm for the HamGraph method, but it works. Let it go at that for now; once you have the beta version of the game ready, you can come back and improve this. Assume you have an appropriate Vertex constructor:

```
/** Create a graph on n vertices with at least 1.5 * n
 *  randomly chosen edges but not a connected graph. */

public void createUnConnectedGraph (int n)  // in HamGraph
{  this.clear();
   this.addVertex (new Vertex (n, null));
   for (int k = 1;  k < 1.5 * this.getNumVertices();  k++)
      this.addEdgeChosenRandomly();
   if (this.isConnected())
      this.createUnConnectedGraph (n);
} //===============================================
```

The Graph class also needs an Iterator that goes through the sequence of Edges that exit a given Vertex and a method to clear all edges from the graph. Listing 19.3 describes the additional Graph methods more precisely. All but one are abstract, to be overridden in any concrete implementation of Graph.

Listing 19.3 The rest of the Graph methods, six to override

```
// public abstract class Graph continued

   /** Remove all edges from the Graph.  Leave the vertices. */
   public abstract void clear();

   /** The following methods throw a RuntimeException if the
    *  parameter is null or its vertex ids are not in the range
    *  1...getNumVertices(), except they may simply ignore 0.
    *  Do not use a Graph iterator to remove anything. */

   /** Tell whether this graph contains the given Edge. */
   public abstract boolean contains (Edge given);

   /** Remove the given Edge if it is in there and return true.
    *  But if the Edge is not in there, simply return false. */
   public abstract boolean remove (Edge given);

   /** Add the given Edge if it is not in there and return true.
    *  But if the Edge is in there, simply return false. */
   public abstract void add (Edge given);

   /** Return an iteration over all edges that exit Vertex v. */
   public abstract Iterator edgesFrom (Vertex v);

   /** Return the number of Edges that end at that Vertex. */
   public abstract int inDegree (Vertex given);

   /** Return the number of Edges that start at that Vertex. */

   public int outDegree (Vertex given)
   {  Iterator it = edgesFrom (given);
      int count = 0;
      while (it.hasNext())
      {  it.next();
         count++;
      }
      return count;
   } //============================================
```

Exercise 19.1 Write the `makeTheGraphConnected()` HamGraph method.
Exercise 19.2 (harder) Write the Graph method that tells the in-degree of a given vertex.
Exercise 19.3* Write a Graph method that removes all the Edges exiting a given vertex.
Exercise 19.4* Explain how both `getVertex` and `addVertex` automatically throw the Exceptions that their comment descriptions say they do.
Exercise 19.5* Assume that MatrixGraph is a concrete implementation of Graph. Write an independent method that creates a connected MatrixGraph with a given number n of Vertices (the parameter) and 1 less Edge than it has Vertices.
Exercise 19.6* Write a Graph method that tells the total number of edges the Graph has.
Exercise 19.7* Write a Graph method that adds enough edges so that the vertex with ID 1 connects to every other vertex. Return the number of new edges that were added.
Exercise 19.8** Write a Graph method that tells whether the graph is directed. Assume that each Edge has public final instance variables TAIL and HEAD of Vertex type.

19.2 The Adjacency Matrix Implementation Of Graphs

A Vertex knows its id number and some associated data such as its name or color or graphical icon. In addition, we will find it useful to be able to assign a marker number to a Vertex from time to time. This marker number is always initialized to zero. A suitable definition is in the upper part of Listing 19.4. Note that we can make `ID` and `DATA` available for public use without violating the encapsulation principle, since they are unchangeable ("final") once the object has been constructed.

Listing 19.4 The Vertex and Edge classes of objects

```
public class Vertex extends GraphPart
{
   public final int ID;
   public final Object DATA;   // identifying info, e.g., a name


   public Vertex (int id, Object data)
   {  ID = id;
      DATA = data;
   } //===============================================
}
//#####################################################


public class GraphPart
{
   private int itsMark = 0;


   public final int getMark()
   {  return itsMark;
   } //===============================================

   public final void setMark (int given)
   {  itsMark = given;
   } //===============================================
}
//#####################################################


public class Edge extends GraphPart
{
   public final Vertex TAIL;  // vertex where it begins
   public final Vertex HEAD;  // vertex where it ends


   public Edge (Vertex from, Vertex to)
   {  TAIL = from;
      HEAD = to;
   } //===============================================

   public boolean equals (Object ob)
   {  return ob instanceof Edge && this.HEAD == ((Edge) ob).HEAD
                                && this.TAIL == ((Edge) ob).TAIL;
   } //===============================================
}
```

The marker part of the Vertex class is separated out so it can be used by the Edge class too. That is, we have both the **Vertex** class and the **Edge** class inherit from the **GraphPart** class, which provides services to set and inspect the current value of the int marker value on the Vertex or Edge, whichever the case may be. In practice, we mostly mark a Vertex or Edge with the number 1 to indicate something special about it and later change the mark back to 0.

Edge objects are also straightforward. An Edge runs from one Vertex, its **tail**, to another Vertex, its **head**. An `equals` method is occasionally needed; two Edges are equal if they have the same head and the same tail, even if the marking numbers are different. The lower part of Listing 19.4 defines the Edge class.

The adjacency-matrix design

One popular way of implementing a graph is to use a two-dimensional matrix: We make `edgeAt[x][y]` be the Edge going from `x` to `y` if the graph has that edge, otherwise `edgeAt[x][y]` is null.

One restriction is that it is quite time-consuming to add more vertices beyond the original amount, since that would require replacing the existing two-dimensional array by a larger one and copying all the Edge values into the new one. So this MatrixGraph implementation overrides `addVertex` to keep it from accepting vertex id numbers larger than the original number of vertices. As the upper part of Listing 19.5 shows (see next page), the constructor requires that you specify the maximum number of vertices.

For this book's particular design of the Graph concept, we never remove any vertices once we add them to the graph, so the `clear` method only clears out Edges. It does this by assigning null to each component of the matrix. Since an edge of a graph tells which two vertices are adjacent, and since we use a two-dimensional matrix to store the adjacency information, MatrixGraph is called the **adjacency-matrix implementation** of Graph. The next section discusses the use of a list of these adjacencies for each vertex; that implementation of Graph is called the **adjacency-list implementation** of Graph.

The real problem for MatrixGraph is to get an Iterator object that goes through the Edges exiting a given vertex id number. This method is left as an exercise; two alternatives are proposed. Whatever you do, iteration will execute in at least big-oh of N time for the MatrixGraph implementation. Consider that most graphs you run across in real life tend to have 3 to 6 Edges per vertex on average. Big-oh of N is a lot worse than big-oh of 6, which is what you get with the adjacency-list implementation discussed shortly.

The `contains`, `remove`, and `add` methods all have straightforward coding and execute in big-oh of 1 time. The `addVertex` method executes in big-oh of 1 time averaged over all vertices. Figure 19.2 illustrates this matrix structure. A picture of a graph is on the left and the corresponding matrix is on the right.

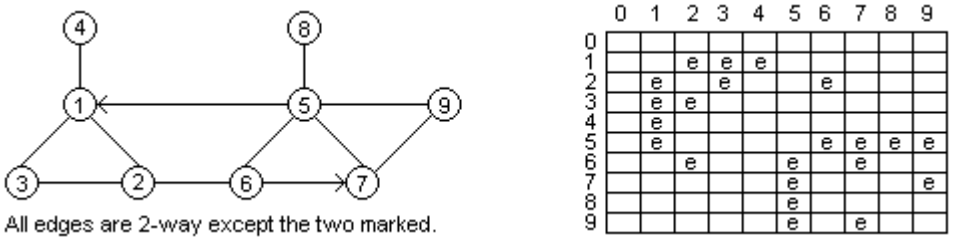


Figure 19.2 The adjacency-matrix representation of a Graph

Listing 19.5 The MatrixGraph class of objects

```
public class MatrixGraph extends Graph
{
   private Edge[][] edgeAt;


   /** Specify the maximum number of vertices the graph can have.
    *  This number cannot be changed by later method calls. */

   public MatrixGraph (int maxVerts)
   {  super();
      edgeAt = new Edge[maxVerts + 1][maxVerts + 1]; // all null
   } //===========================================

   public final void addVertex (Vertex v)
   {  if (v.ID >= edgeAt.length)
         throw new IllegalArgumentException ("not with matrix");
      super.addVertex (v);
   } //===========================================

   public final void clear()
   {  edgeAt = new Edge[maxVerts + 1][maxVerts + 1]; // all null
   } //===========================================

   public final boolean contains (Edge given)
   {  return edgeAt[given.TAIL.ID][given.HEAD.ID] != null;
   } //===========================================

   public final boolean remove (Edge given)
   {  Edge e = edgeAt[given.TAIL.ID][given.HEAD.ID];
      edgeAt[given.TAIL.ID][given.HEAD.ID] = null;
      return e != null;
   } //===========================================

   public final void add (Edge given)
   {  edgeAt[given.TAIL.ID][given.HEAD.ID] = given;
   } //===========================================

   public final java.util.Iterator edgesFrom (Vertex v)
   {  return null; // left as an exercise
   } //===========================================
}
```

As an example of additional methods that could be in the MatrixGraph class, you could have a method that tells whether a given vertex has a connection both to and from any other vertex, using the standard some-A-are-B logic:

```
   public boolean hasTwoWayEdge (Vertex given) // in MatrixGraph
   {  for (int k = 1;  k <= getNumVertices();  k++)
      {  if (edgeAt[given.ID][k] != null
                && edgeAt[k][given.ID] != null)
            return true;
      }
      return false;
   }  //=======================================
```

Exercise 19.9 Write a MatrixGraph method that removes all the Edges exiting a given vertex, by directly accessing the two-dimensional array.
Exercise 19.10 Write a MatrixGraph method that tells the out-degree of a given vertex id, by directly accessing the two-dimensional array.
Exercise 19.11 Write a MatrixGraph method that tells the in-degree of a given vertex id, by directly accessing the two-dimensional array.
Exercise 19.12 (harder) Implement the `edgesFrom` method by creating an ArrayList object, adding to it all the non-null Edge values in the given row, then returning its Iterator object.
Exercise 19.13 (harder) Add an instance variable to the MatrixGraph class of objects and coding so that one can find the in-degree of a vertex simply by calling the following:
`public int inDegree (Vertex given) { return itsIns[given.ID]; }`
Exercise 19.14* Write a Graph method that resets the mark on every Vertex to zero.
Exercise 19.15* Write a Graph method that resets the mark on every Edge to zero.
Exercise 19.16* Write a Graph method that tells how many edges have a tail whose ID is a smaller number than its head's ID.
Exercise 19.17* Write a MatrixGraph method `public Graph copy()`: The executor creates and returns an exact duplicate of itself.
Exercise 19.18* Write a MatrixGraph method `public Graph transpose()`: The executor returns a new MatrixGraph object that has the same vertices but the reversed edges, i.e., an edge from v to w is in one if and only if an edge from w to v is in the other.
Exercise 19.19** Implement the `edgesFrom` method by using a private Iterator class nested in the MatrixGraph class. This executes faster than the way described in the earlier exercise, if each such Iterator has an instance variable `itsPos` that keeps track of the index of the next available Edge object.

19.3 The Adjacency List Implementation Of Graphs

Another popular way of implementing a graph is to keep, for each vertex id number, a list of all the Edges exiting the Vertex of that id number. The list for one vertex is only accessed sequentially, not random-access like an array. However, the list of all these Edge lists is random-access, i.e., it can be directly accessed by the index number of a Vertex. This makes some graph operations execute faster, but other graph operations execute slower, than in the adjacency matrix implementation.

We use an ArrayList named `itsList` to random-access the Edge lists, one list of Edges for each Vertex id number. We do not use the zeroth component of this main list. Any Collection type would be fine for each individual Edge list, but for convenience we also use ArrayLists for them. Listing 19.6 (see next page) gives this concrete ListGraph subclass of Graph.

The ListGraph constructor puts a dummy value in for the zeroth component of `itsList`, so that a Vertex with id `k` will have its edge list at index `k`. We need to override the `addVertex` method from the basic Graph class to add coding that allows for the list of edges. First we execute the basic `addVertex` method from the Graph class that we override (the statement `super.addVertex(v)`). Then we go on to add one new empty list of Edges for each of the new vertices.

Listing 19.6 The ListGraph class of objects

```
import java.util.*;

public class ListGraph extends Graph
{
   private ArrayList itsList = new ArrayList();

   public ListGraph()
   {  super();
      itsList.add (null);  // we do not use the value at index 0
   } //=============================================

   public final void addVertex (Vertex v)
   {  super.addVertex (v);
      if (v.ID >= itsList.size())
      {  for (int k = itsList.size();  k <= v.ID;  k++)
            itsList.add (new ArrayList());
      }
   } //=============================================

   /** Remove all edges from the graph, but not the vertices. */

   public final void clear()
   {  for (int k = 1;  k <= getNumVertices();  k++)
         itsList.set (k, new ArrayList());
   } //=============================================

   public final boolean contains (Edge given)
   {  return ((Collection) itsList.get (given.TAIL.ID))
               .contains (given);
   } //=============================================

   public final boolean remove (Edge given)
   {  return ((Collection) itsList.get (given.TAIL.ID))
               .remove (given);
   } //=============================================

   public final void add (Edge given)
   {  if (given.HEAD.ID < 1 || given.HEAD.ID > getNumVertices())
         throw new IllegalArgumentException ("bad vertex");
      Collection z = (Collection) itsList.get (given.TAIL.ID);
      if (z.contains (given))
         z.remove (given);
      z.add (given);
   } //=============================================

   public final Iterator edgesFrom (Vertex v)
   {  return ((Collection) itsList.get (v.ID)).iterator();
   } //=============================================
}
```

Remember that, if e.g. we add a Vertex with id number 10 at a time when `getNumVertices()` is 6, we have to add four more vertices (numbered 7, 8, 9, 10), not just one. That is, we always have one vertex for every id number from 1 up to `getNumVertices()` inclusive.

For a given Vertex v, `(Collection)itsList.get(v.ID)` is the Collection object (specifically an ArrayList object) that contains all the edges exiting v. So you can use its iterator to go through the list of all edges that exit from that vertex. The `edgesFrom` method simply returns that Iterator object. Figure 19.3 shows a graph and the corresponding ArrayLists that the ListGraph implementation would have.

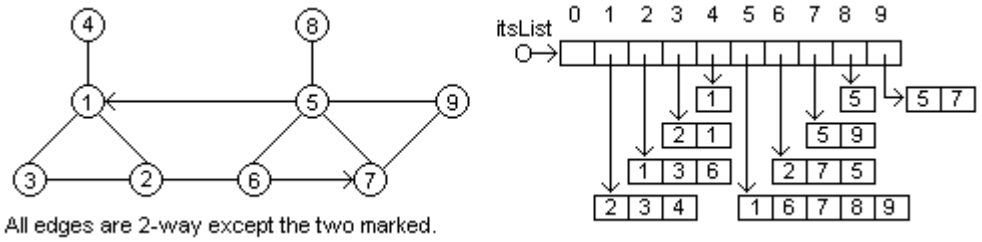


Figure 19.3 The adjacency-list representation of a Graph

A given Edge exits from the vertex `given.TAIL`, so `itsList.get(given.TAIL.ID)` is the Collection object that contains all the edges exiting `given.TAIL`. So ListGraph's `contains(Edge)` method simply checks whether that Collection object contains an Edge that equals the given Edge, and its `remove(Edge)` method simply has that Collection object remove the given Edge. Both should throw an Exception if their vertices have ID numbers outside the range `1...numVertices()`; this is left as an exercise.The `add(Edge)` method is trickier because an ArrayList always adds the value it is given regardless of whether it is already in there. So you have to first check whether the given Edge is already there before you add it. Also, you have to throw a RuntimeException if the vertex ids in the given Edge are not from 1 to `getNumVertices()`. That happens "automatically" if the list of Edges for the given Edge's tail does not exist, but you have to check separately that the given Edge's head is also in the allowable range.

Note that the coding in Listing 19.6 follows the general coding principle that you create a variable to store a value only if that value will be used later more than once. In the `add` method, the value of `z` is used several times in the statement after it is assigned. In the `contains` and `remove` method, the value returned by `itsList.get` is only used once, as the executor of a Collection method, so we do not assign the value to a variable.

Exercise 19.20 Revise the contains and remove methods in Listing 19.6 to throw an Exception when the vertex indexes are outside the range `1...numVertices()`.
Exercise 19.21 Write a ListGraph method that tells the out-degree of a given vertex, by directly accessing the ArrayList corresponding to the given vertex id.
Exercise 19.22 (harder) Write a ListGraph method that removes all the Edges exiting a given vertex v whose head has a larger ID than v, by directly accessing the ArrayLists.
Exercise 19.23* Rewrite the `clear` method in Listing 19.6 to clear out all the values in the ArrayList objects that are already in the list, rather than replacing them by new empty ArrayList objects.
Exercise 19.24* Rewrite the `add` method in Listing 19.6 to execute much faster, using indexes in the ArrayList.
Exercise 19.25** Write a ListGraph method `public Graph copy()`: The executor creates and returns an exact duplicate of itself.
Exercise 19.26** Write a ListGraph method `public Graph transpose()`: The executor returns a new ListGraph object with the same vertices but with the reversed edges, i.e., an edge from v to w is in one if and only if an edge from w to v is in the other.

19.4 Topological Sorting; Big-Oh For Graph Algorithms

There are many applications where a directed graph tells which activities must be performed before which other activities (an edge from v to w indicates that v must be performed before w). **Critical-path analysis** tries to find optimal scheduling of tasks in such situations. Scheduling college courses to satisfy prerequisites is an example.

Listing the vertices of a graph in a sequence so that, for each edge from some vertex v to some other vertex w, v comes before w in the sequence, is a **topological sort** of the vertices. Then the activities to be scheduled can be performed in the order that their vertices appear on that list.

First we obtain a list of those vertices that have no edges coming in to them. If none of these "starter vertices" exist, then the vertices cannot be sorted. The method in the upper part of Listing 19.7 has a queue as the parameter, initially empty; the method fills the queue with all of the "starter vertices" available. It uses an `inDegree` method from the exercises. It also marks each non-starter vertex with the number of edges that come into it. Reminder: The three basic QueueADT methods for queues are as follows:

- `q.enqueue(v)` to add a value v;
- `q.dequeue()` to remove a value and return it (the return type is Object); and
- `q.isEmpty()` to see whether any values are left.

Listing 19.7 Methods used in topological sorting

```
private void getStarters (QueueADT toCheck)
{  Iterator verts = vertices();
   while (verts.hasNext())
   {  Vertex v = (Vertex) verts.next();
      int numPredecessors = inDegree (v);
      if (numPredecessors == 0)
         toCheck.enqueue (v);
      else
         v.setMark (numPredecessors);
   }
}  //=======================================


private void topoSort (QueueADT toCheck, QueueADT sorted)
{  getStarters (toCheck);
   while ( ! toCheck.isEmpty())   // once per Vertex
   {  Vertex ok = (Vertex) toCheck.dequeue();
      sorted.enqueue (ok);
      Iterator edges = edgesFrom (ok);
      while (edges.hasNext())
      {  Vertex v = ((Edge) edges.next()).HEAD;
         v.setMark (v.getMark() - 1);
         if (v.getMark() == 0)
            toCheck.enqueue (v);
      }
   }
}  //=======================================
```

To illustrate, the `getStarters` method applied to the directed graph in Figure 19.4 (see next page) would put the vertices numbered 4 and 8 on the `toCheck` queue and set the mark on each other vertex to its in-degree. So the number at the left of each vertex is the mark for that vertex.

The directed graph at right has vertices numbered 1 to 9. To the left of each vertex is its in-degree -- the number of edges that have that vertex as the head.

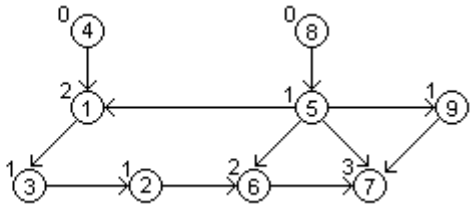


Figure 19.4 A directed graph with in-degrees specified

The topoSort method

Once we have the starter vertices put on a queue, we can go through the elements of that queue and do two things with each vertex we take from the queue: shift that vertex to the rear of a different queue of sorted vertices, and reduce the mark on every other vertex to what its in-degree would be if the shifted vertex were not in the graph.

For the graph in Figure 19.4, we would move vertex 4 from the `toCheck` queue to the sorted queue and reduce the mark on vertex 1 from 2 to 1. We would then move vertex 8 from the `toCheck` queue to the sorted queue and reduce the mark on vertex 5 from 1 to 0, which entitles vertex 5 to go on the `toCheck` queue.

In other words, we have the mark on any remaining vertex v be the number of edges coming into v from vertices that are not on the sorted queue. So when the amount marked on v is reduced to zero, that means that every vertex that comes before v is already on the sorted queue, so v is also a candidate for the sorted queue. This logic is coded in the lower part of Listing 19.7.

Continuing with the example of Figure 19.4, the only vertex now on the `toCheck` queue is vertex 5. We move it from the `toCheck` queue to the sorted queue and go through the edges exiting vertex 5, which reduces the marks on vertices 1 and 9 from 1 to 0, and reduces the marks on vertices 6 and 7 to 1 and 2, respectively. That puts vertices 1 and 9 on the `toCheck` queue. The rest of the process is left as an exercise.

These two methods are private because a client of the class would call some public method that would create the two queues, call the `topoSort` method, then print out the information stored in the sorted queue. The sorted queue will contain all the vertices if the Graph can be topologically sorted, otherwise it will be missing some vertices.

Big-oh for basic Graph algorithms

For this section and later in the chapter, let **NV** stand for the number of vertices in the graph under discussion and let **NE** stand for the number of edges in that same graph. The execution time for graph operations is generally a function of one or both of these numbers.

The `clear` method for ListGraph is a big-oh of NV operation, since it executes one operation for each vertex in `itsList`. The `clear` method for MatrixGraph is a big-oh of 1 operation, since it executes a single assignment. Check the coding in Listings 19.4 and 19.5 to verify this.

The `contains(Edge)` method for MatrixGraph is a big-oh of 1 operation, since it only executes three actions independently of NE or NV (okay, you could call it big-oh of 3, but we do not do that -- review the definition of big-oh to see why). The `contains(Edge)` method for ListGraph is a big-oh of NE/NV operation, since it directly accesses the ArrayList of the given edge's tail vertex but then searches sequentially through that ArrayList to find the specified edge. Thus the execution time is proportional to the number of edges that particular vertex has. On average, that will be NE/NV, because if you were to add up the sizes of each of the ArrayList adjacency lists, one per vertex, you would have a total of NE. The NE/NV value is called an **amortized average**.

The `remove(Edge)` and `add(Edge)` methods are also big-oh of 1 for MatrixGraph and big-oh of NE/NV for ListGraph, and for the same reasons, but we restrict our discussion in the rest of this section to graphs with a fixed number of edges and vertices.

Big-oh for the topological sort

The `getStarters` method in Listing 19.7 clearly takes big-oh of NV time plus whatever time calculation of the in-degrees takes. If these in-degree values are stored as the graph is built, it takes time proportional to NE (one or two extra operations each time a new edge is added). So big-oh for `getStarters` is NV+NE.

The `topoSort` method has a while-loop that executes once for each vertex. That while-loop contains an inner while-loop that executes once for each edge out of the current vertex. So the statements in the body of the inner while-loop are executed once for each edge in the graph. All of the other methods called in that coding can be done in big-oh of 1 time. So the overall execution time for the sorting process is NV+NE for `getStarters` followed by NV+NE for the actual sorting. The big-oh execution time is therefore big-oh of NV+NE.

Oops! That is not quite right. One pass through an `edgesFrom(v)` iterator requires NE/NV operations for ListGraph (i.e., the average number of edges per vertex), but it requires NV operations for MatrixGraph (the number of components in one row of the matrix). So for the adjacency-matrix implementation, a topological sort requires NV+NE followed by NV*NV, which is big-oh of NV-squared.

To see the difference, consider that many situations have a low average number of edges per vertex regardless of the number of vertices. For instance, it is common that NE ranges from 3 to 6 times the number of vertices, whether you have tens or hundreds or thousands of vertices. So a topological sort for ListGraph is big-oh of 6*NV, which is equivalent to big-oh of NV, whereas a topological sort for MatrixGraph is big-oh of NV^2.

Exercise 19.27* Complete the description of the topological sort process for Figure 19.4. You already have vertices 4, 8, and 5 on the sorted queue and vertices 1 and 9 on the `toCheck` queue.

Exercise 19.28* Write the public sort method that calls the `topoSort` method in Listing 19.7 and prints the information in the queue.

19.5 Graph Traversals

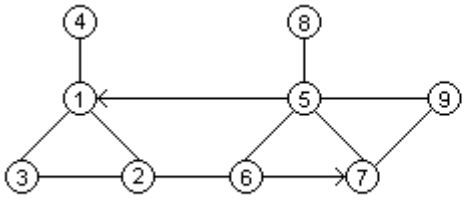
A **subgraph** of a graph G is a graph all of whose edges and vertices are in G, though not necessarily vice versa. A **subtree** of a graph G is a directed subgraph of G for which you can get from a specific vertex (the subtree's **root**) to any other vertex in the subtree by one and only one path in the subtree. The subtree is a **spanning tree** of G if it contains all the vertices of G.

A **depth-first traversal** of a graph is a processing of the vertices of some subtree of the graph starting with the root of the subtree, such that (a) no vertex is processed before its parent is processed, and (b) once a vertex v is processed, then all the vertices you can reach from v by following a path in the subtree are processed before any other vertices are processed. If this definition seems too abstract, a simple definition is that depth-first traversal is what you get from an appropriate recursive method such as the one to be described next.

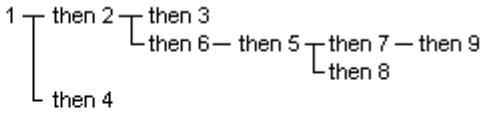
An example of a depth-first traversal is the `markReachables` method in the upper part of Listing 19.8 (see next page). It marks with 1 every vertex that can be reached from a given vertex. The 1 indicates that vertex has been "visited" by the algorithm. We leave the mark on the vertex until after the traversal is finished, to avoid visiting it twice.

All of the vertices marked 1 are in the traversal subtree, assuming that the initial call of the method was when all vertices of the graph were marked zero. The root is the first vertex marked 1. Also, any edge obtained in line 4 of that method whose head v satisfies `v.getMark() == 0` is to be considered part of the traversal subtree. So if you reach all the vertices of the graph with this method, you also get a spanning tree of the graph by recording those particular edges as you go.

An obvious problem with this recursive logic is that it leaves marks on some of the vertices, which should be removed before the method returns its answer. So that method should only be called by something like the public `isConnected` method in the middle part of Listing 19.8, which is the method needed by the Hamiltonian software of Section 19.1: It tells whether the graph is connected. Figure 19.5 gives the order in which vertices are visited for one particular graph using `markReachables`.



All edges are 2-way except the two marked.



So 1, 2, 3, 6, 5, 7, 9, 8, 4 in that order.
This order of depth-first traversal assumes that edgesFrom produces vertices in increasing order of id number.

Figure 19.5 Depth-first traversal of a graph starting from vertex #1

This basic depth-first traversal logic can be modified to solve several different problems. If you want a list of all edges in a spanning tree, assuming that the graph is connected, you could insert at line 6 (subordinate to the if-condition) a statement that puts that edge on a list of edges. Later you return that list as the answer. If you want to find out whether the graph has a **cycle** (a path of two or more vertices that starts and ends at the same vertex) involving the starting vertex, it will be when line 4 detects the starting vertex.

Listing 19.8 Traversal methods for the Graph class of objects

```
   /** Mark 1 on each vertex reachable from the given Vertex. */

   private void markReachables (Vertex given)
   {  given.setMark (1);                                    //1
      Iterator it = edgesFrom (given);                      //2
      while (it.hasNext())                                  //3
      {  Vertex v = ((Edge) it.next()).HEAD;                //4
         if (v.getMark() == 0)                              //5
            markReachables (v);                             //6
      }                                                     //7
   } //===========================================

   /** Tell whether you can reach each vertex from vertex #1. */

   public boolean isConnected()
   {  markReachables (getVertex (1));                       //8
      Iterator it = vertices();                             //9
      int count = 0;                                        //10
      while (it.hasNext())                                  //11
      {  Vertex v = (Vertex) it.next();                     //12
         if (v.getMark() == 1)                              //13
         {  count++;                                        //14
            v.setMark (0);                                  //15
         }                                                  //16
      }                                                     //17
      return count == getNumVertices();                     //18
   } //===========================================

   /** Find a path from given through all unmarked vertices and
    *  mark each vertex with its sequence number in that path. */

   private boolean solveSalesman (Vertex given, int seqNum)
   {  given.setMark (seqNum);                               //19
      if (seqNum == getNumVertices())                       //20
         return true;                                       //21
      Iterator it = edgesFrom (given);                      //22
      while (it.hasNext())                                  //23
      {  Vertex v = ((Edge) it.next()).HEAD;                //24
         if (v.getMark() == 0)                              //25
         {  if (solveSalesman (v, seqNum + 1))              //26
               return true;                                 //27
         }                                                  //28
      }                                                     //29
      given.setMark (0);                                    //30
      return false;                                         //31
   }  //=======================================
```

You could also use depth-first traversal to do a topological sort when you are certain the graph does not contain cycles: Recursively set the mark of each vertex to 1 plus the sum of all marks of vertices that it connects to. Then for any edge connecting v to w, v will have a larger mark than w and thus comes earlier in the "sorted" sequence.

Note that the following problems can be solved nicely with depth-first traversals:

1. You have a two-dimensional array of pixels and you have to start from a given white pixel and color it and all adjacent white pixels a given different color.
2. You have a two-dimensional representation of a maze and you have to find your way out of it from a given point.

The Traveling Salesman Problem

A classic problem in graph theory is to find a path from a given vertex that goes through every other vertex in the graph exactly one time. That is, you cannot pass through a vertex you have previously passed through. An application is to find the sequence of cities a traveling salesman should visit so that he or she does not go through the same city twice. So this is called the **Traveling Salesman Problem**.

The basic depth-first traversal logic can be adapted to this problem, but we have to actually say which vertices are visited in which order. The `solveSalesman` method in the lower part of Listing 19.8 is given a vertex that we can visit next and the number of that vertex in the path we are constructing. For instance, the second parameter is 5 if the given vertex is the fifth one in the sequence from the starting vertex. We would call it initially as `solveSalesman(startingVertex,1)`.

As an illustration, the `solveSalesman` method applied with the starting vertex numbered 4 in the earlier Figure 19.5 would set the mark on vertex 4 to 1, then set the mark on vertex 1 to 2, then set the mark on vertex 2 to 3 but fail to complete the path. So it would change the mark on vertex 2 back to 0 and set the mark on vertex 3 to 3. Next it sets the mark on vertex 2 to 4, then the mark on vertex 6 to 5. The rest of the process is left as an exercise.

Compare this logic step for step with the logic of `markReachables`. We of course assign consecutive numbers to the vertices as they are visited instead of 1 to each of them. This is why we pass `seqNum + 1` on the recursive call in line 24. Other than that, there are only a few differences:

- Lines 26-27 return `true` if a path is found in later calls, without re-setting the markers.
- Lines 20-21 return `true` if a path is found in this call, without re-setting the marker.
- Lines 30 and 31 are executed when we fail to complete the path from the given vertex. They set the mark back to zero and return `false` (since we cannot put the given vertex on the current path and manage to complete it).

Using a stack to store the sequence of visits

If a call of `solveSalesman` with an initial count of 1 returns `true`, we can easily find the sequence number for each vertex, but we do not have the vertices in the order they occur in that sequence. Perhaps a better solution would be to also pass a stack object as a parameter instead of `seqNum`. We would then push a vertex on the stack when we mark it, and pop it from the stack when we set its mark back to zero.

Then the public method that calls the `solveSalesman` method tests whether the stack is empty when the method returns and, if not, prints the values in the order in which they come off the stack.

Breadth-first traversal

Suppose you would like to know the length of the shortest path from a given vertex v to each of the vertices you can reach from v. You would mark all the vertices that you can reach directly from v with a 1. Then you would mark with a 2 all the vertices you can reach in one more step, i.e., directly from a vertex marked 1. This keeps up until you have marked all the vertices possible.

This is easiest to do with a queue. The following method would be passed an initially empty QueueADT object. Compare it with the `markReachables` method in Listing 19.8 to see how it differs:

```
private void markDistance (Vertex given, QueueADT queue)
{  given.setMark (1);
   queue.enqueue (given);
   while (! queue.isEmpty())
   {  Vertex current = (Vertex) queue.dequeue();
      Iterator it = this.edgesFrom (current);
      while (it.hasNext())
      {  Vertex v = ((Edge) it.next()).HEAD;
         if (v.getMark() == 0)
         {  v.setMark (current.getMark() + 1);
            queue.enqueue (v);
         }
      }
   }
}  //=======================================
```

For the graph shown in the earlier Figure 19.5, the order in which this method visits the vertices is as follows, assuming the initial call is with vertex 1: visit 1, then put 2, 3, and 4 on the queue. So visit 2 next, which puts 6 on the queue after 3 and 4. Visit 3 next, which adds nothing to the queue. Visit 4 next, which also adds nothing to the queue, so the queue now just has 6 on it. Visit 6 next, which adds 5 and 7 to the queue. Visit 5 next, which adds 8 and 9 to the queue after 7. Finally, visit 7, then 8, then 9. So the overall sequence is 1,2,3,4,6,5,7,8,9.

All of these traversal algorithms mark the vertices they visit and, if they see a previously visited vertex, ignore it. This causes the algorithm to traverse only a subtree of the graph, not the whole graph. The advantage is that you avoid going around in circles, perhaps never terminating the algorithm.

Big-oh for depth-first or breadth-first traversal of a graph

The `markReachables` method in the upper part of Listing 19.8 is a prototypical depth-first traversal. Since a call of the method marks its parameter with a 1, and the method is never called when the parameter is already marked 1, it follows that the method is called at most once for each vertex. So the execution time is NV multiplied by the average execution time for one pass through the `edgesFrom` iterator.

In general, traversal of a graph goes through each vertex one time and, at each vertex, goes through each edge out of that vertex one time. That is again big-oh of NV+NE for the adjacency-list implementation and big-oh of NV^2 for the adjacency-matrix implementation, just as for the topological sort.

Since hardly anyone deals with graphs that have much fewer edges than vertices, NE will generally be larger than NV. In such cases, the topological sort and depth-first traversal for adjacency lists are simply big-oh of NE, though they are NV^2 for an adjacency matrix.

Big-oh for the traveling salesman

The above analysis applies for true traversal of the vertices, where each vertex is visited but once, whether depth-first or breadth-first. But when you call the `solveSalesman` method in Listing 19.8 recursively, you mark the vertex, amble around for a while, then unmark it. That means that it can be visited again, perhaps many times. That can make the execution time exponential in the number of vertices.

For a specific example: Consider a graph of 8 vertices in which vertex #2 is the only vertex that connects to vertex #8, but every vertex numbered 1 through 7 connects to every other vertex numbered 1 through 7. Figure 19.6 (see next page) shows the adjacency matrix with a mark where there is an edge (the empty boxes are nulls; the components with indexes of zero are left out, since they are ignored).

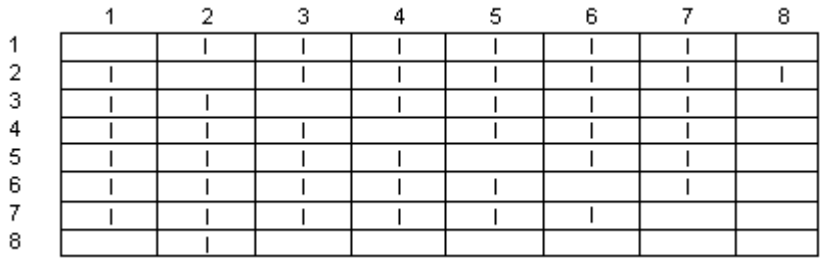


Figure 19.6 Edges between vertices (only #2 connects to #8)

Then `solveSalesman (getVertex(1), 1)` goes through more than 5! (that is 5-factorial, i.e., 120) different paths that start with the edge from 1 to 2 before it finds the path 1 -> 3 -> 4 -> 5 -> 6 -> 7 -> 2 -> 8. And if we add one more vertex, so that #2 is the only one that connects to #9 but all other possible connections are there, it will look at more than 6! = 720 different paths. If there are 20 vertices for which #2 is the only one that connects to #20 but all other possible connections are there, the method will look at more than 17! > trillions of different paths before it finds the right one. In general, the execution time is greater than (NV-3)!, which is greater than 2^{NV} when NV is at least 9. That is, this method has at least an exponential big-oh execution time.

Inductive proof that (NV-3)! > 2^{NV} when NV is at least 9: When NV is 9, (9-3)! is 720 and 2^9 is only 512, so the inequality clearly holds when NV is 9. Once we have proven the inequality true when NV is any k >= 9, then the inequality for k+1 has k-2 times as much on the left (thus at least 7 times as much), and the inequality has only twice as much on the right, so the left-hand side remains larger than the right-hand side.

Exercise 19.29 What is the depth-first traversal sequence for the graph in Figure 19.5 starting from vertex #6 instead of from vertex #1? Assume that `edgesFrom` returns vertices in increasing order of id numbers.
Exercise 19.30 Same question as the previous exercise, except for breadth-first traversal rather than depth-first traversal.
Exercise 19.31 Complete the given description of the traveling salesman process for Figure 19.5. You already have visited vertices 4, 1, 3, 2, 6 in that order. Include all vertices mentioned, even those that led to a failure. Can you find a traversal starting from any other vertex besides vertex 4?
Exercise 19.32 Revise the coding in Listing 19.8 to obtain a method `public boolean CanGo(Vertex one, Vertex two)` to tell whether there is a path from `one` to `two`.
Exercise 19.33 Revise the coding of the `markDistance` method to have an extra int parameter `n` and to return the distance to the vertex with ID `n` as soon as it is found.
Exercise 19.34* For each value of N from 2 to 5, draw a graph with N vertices and no cycles that has the maximum possible number of edges. Through trial and error, find the formula for the number of edges in terms of N.
Exercise 19.35* Write a method to change all vertices' marks back to zero.
Exercise 19.36* Write a method that calls `solveSalesman` in Listing 19.8 and returns a String value with pairs such as (3,5) indicating that vertex 3 is the fifth vertex in the traveling salesman path. It should return the empty String if there is no solution.
Exercise 19.37* Revise `solveSalesman` as described for a stack parameter and write the method that contains the method call `solveSalesman (getVertex(1), 1, stack)`. It should print the vertices in the full traveling salesman path in the order they are to be visited.
Exercise 19.38* Write a method that prints the edges in a spanning tree of a graph if it is a connected graph. Revise `markReachables` in Listing 19.8 slightly and use it to store the edges on a queue.
Exercise 19.39* Rewrite `isConnected` to do the same thing but without using a counter. Hint: Use a boolean local variable instead.

Part B Enrichment And Reinforcement

19.6 Union-Find Structures

The algorithms for weighted graphs in the next section require the use of a **UnionFind** structure. This, in its simplest form, is a data structure that stores information about the non-negative integers up to but not including some fixed number `N`. Specifically, it treats those integers as divided into a number of disjoint groups. This partitioning of the integers is initially done (when the UnionFind object is constructed) with each integer in its own group of that one number. The groups can then be combined.

The only change you can make in a UnionFind object is to tell it to combine two separate groups into one larger group; the `unite` method does this. The only query you can make of a UnionFind object is to ask it whether two integers you specify are in the same group; the `equates` method does this. Example: The following coding divides the int values 0 through 99 into two groups, the odd numbers and the even numbers. Then the expression `group.equates(x, 1)` tells whether x is odd:

```
UnionFind group = new UnionFind (100);
for (int k = 0;  k < 98;  k++)
   group.unite (k, k+2);
```

The coding in Listing 19.9 (see next page) gives one of the simplest implementations of this data structure: We use an array of `N` different int values to name the groups. So initially 0 is in group #0, 1 is in group #1, 2 is in group #2, etc. When we unite the groups for two values `one` and `two`, we simply go through the array of group names and change the group name of each element of `two`'s group to be `one`'s group name. Naturally, we first make sure that the two given ints are in the range `0...N-1`.

The execution time for `equates` is big-oh of 1, but the execution time for `unite` is big-oh of `N` time. This is simply not good enough for our algorithms. We need something that executes faster.

Using circular linked lists of nodes

An interesting idea is to store in each array component a reference to one node in a circular linked list of nodes, where each node contains an int value that is in the group. Circular means that the last node links back to the first. For instance, if 2, 7, and 4 are all in the same group, then we could have a linked list of three nodes with 2 in the first node, which links to a node containing 7 as its data, which links to a node containing 4 as its data, which links back to the node containing 2 as its data.

Initially, each component `itsItem[k]` references a node `p` that contains `k` as its data and links back to itself (`p.itsData` is `k` and `p.itsNext` is `p`). The really interesting part is that we never need to change the values stored in the array; we only need to change the links. For instance, to unite the group containing 2 (and possibly others) with the group containing 5 (and possibly others), execute this sequence of statements:

```
Node saved = itsItem[2].itsNext;
itsItem[2].itsNext = itsItem[5].itsNext;
itsItem[5].itsNext = saved;
```

So execution time for `unite` is now reduced to big-oh of 1. However, answering the true-false question `equates(3,7)` requires starting from the node in `itsItem[3]` (which will of course contain 3) and checking each node on that linked list to see if it contains 7. So `equates` now executes in big-oh of `N` time. That is still not acceptable.

Listing 19.9 The UnionFind class of objects

```
public class UnionFind    // simple array implementation
{
   private final int itsLength;
   private int[] itsItem;


   /** Create a partition of the int values 0...numValues-1,
    *  initially with each int in its own 1-element group.  Use
    *  1 in place of numValues if numValues is not positive. */

   public UnionFind (int numValues)
   {  itsLength = numValues > 0  ?  numValues  :  1;
      itsItem = new int[itsLength];
      for (int k = 0;  k < itsLength;  k++)
         itsItem[k] = k;
   } //=============================================

   /** Tell whether one and two are in the same group. */

   public boolean equates (int one, int two)
   {  return one >= 0 && one < itsLength && two >= 0
             && two < itsLength && itsItem[one] == itsItem[two];
   } //=============================================

   /** Assure that one and two are in the same group.
    *  No effect if one or two is not in 0...numValues-1. */

   public void unite (int one, int two)
   {  if (one >= 0 && one < itsLength && two >= 0
             && two < itsLength && itsItem[one] != itsItem[two])
      {  for (int k = 0;  k < itsLength;  k++)
         {  if (itsItem[k] == itsItem[two])
               itsItem[k] = itsItem[one];
         }
      }
   }  //=========================================
}
```

Using queues of nodes

We try a third implementation: The component for each element of a given group G stores a non-empty queue object implemented as a linked list of nodes, each node containing one of the elements of G (rather like the NodeQueue class in Section 14.4). Now `equates` is back to being big-oh of 1: simply check that `itsItem[one]` and `itsItem[two]` are the same queue. The `unite` method for two groups of size x and y changes one of the two queues to contain a linked list of x+y nodes; this can be done in big-oh of 1 time. Then it changes the component for each element of the other group to be that revised queue value.

As long as we can be sure to always change the components for the smaller group, this will take at most `(N/2)*log(N)` operations by the time we have combined all `N` elements into one group. Since that requires that `unite` is called `N-1` times, average execution time is about `log(N)/2` per `unite` operation. This `unite` method is in Listing 19.10. The rest of this improved UnionFind class is left as an exercise. An alternate implementation in which `unite` is big-oh of 1 and `equates` is big-oh of log(N) is left as a major programming project.

Listing 19.10 The UnionFind class of objects, improved

```
public class UnionFind    // Queue of Nodes implementation
{
   private final int itsLength;
   private Queue[] itsItem;     // Queue is a nested class
   // equates is unchanged.  The constructor is an exercise.


   /** Assure that one and two are in the same group.
    *  No effect if one or two is not in 0...numValues-1. */

   public void unite (int one, int two)
   {  if (one >= 0 && one < itsLength && two >= 0
            && two < itsLength && itsItem[one] != itsItem[two])
      {  Queue first = itsItem[one];
         Queue second = itsItem[two];
         if (first.itsSize < second.itsSize)  // swap the two
         {  Queue saved = first;
            first = second;
            second = saved;
         }
         first.itsRear.itsNext = second.itsFront;
         first.itsRear = second.itsRear;
         first.itsSize += second.itsSize;
         for (Node p = second.itsFront; p != null; p = p.itsNext)
            itsItem[p.itsData] = first;
      }
   }  //=========================================


   private static class Queue
   {
      public Node itsFront;
      public Node itsRear;
      public int itsSize;

      public Queue (int given)
      {  itsFront = new Node (given, null);
         itsRear = itsFront;
         itsSize = 1;
      }
   }  //=========================================

   private static class Node { /* left as an exercise */ }
}
```

It is hard to prove the total number of operations is at most `(N/2)*log(N)`, but you can easily prove a limit of `N*log(N)`: Each time the value of a component of `itsItem` changes, the new queue is at least twice the size of the old queue, so it does not change more than `log(N)` times. Example for a power of 2: Let `N` be 16, and make 8 groups of 2 (8 operations), then 4 groups of 4 (8 more operations), then 2 groups of 8 (8 more), then 1 group of 16 (8 more), for a total of 32 operations, which is `(16/2) * log(16)`.

Exercise 19.40 Write the Node class for Listing 19.10.
Exercise 19.41 Write the constructor for Listing 19.10.
Exercise 19.42* Write the complete UnionFind implementation using a circular linked list, as described in this section.

19.7 Weighted Graphs; Algorithms For Minimum Spanning Trees

In some applications, each Edge of a graph has a positive number assigned to it called its **weight**. The graph is then called a **weighted graph** or a **network**. For example, if each Vertex represents a city and each Edge represents a section of train track that could be built between two cities, then the problem of finding the cheapest way to connect all cities depends on the cost of each segment of track. In that case, the cost of the track is the weight of the Edge. The problem of finding a way to connect cities that minimizes travel time depends on the mileage of each segment of track. In that case, the mileage of the track is the weight of the Edge.

It is easy to modify the declarations in the Edge class to allow for Edges having weights. Simply add a constructor that has the weight as a third parameter and provide a way of retrieving the weight of the Edge later. Also amend the existing 2-parameter Edge constructor to assign WEIGHT = 1. We only allow positive weights:

```
// modifications to the Edge class
public final double WEIGHT;

public Edge (Vertex from, Vertex to, double weight)
{  TAIL = from;
   HEAD = to;
   WEIGHT = weight > 0.0 ? weight : 1.0;
} //================================================
```

A method to find the average weight of the edges leaving a given vertex is as follows, assuming that the vertex does in fact have any edges leaving it:

```
public double averageWeight (Vertex given)
{  Iterator it = edgesFrom (given);
   double total = ((Edge) it.next()).WEIGHT;
   int count = 1;
   while (it.hasNext())
   {  total += ((Edge) it.next()).WEIGHT;
      count++;
   }
   return total / count;
} //================================================
```

For the rest of this section, we will assume that our graphs are undirected, which means that for every edge from vertex v to vertex w there is an edge from w to v of the same weight. From an undirected point of view, we consider these to be the same edge. When we work with a subtree of the undirected graph, we have a particular root vertex in mind, and paths in the tree structure go away from that root vertex. But we also talk about a path in the tree from one vertex to another, in which case we allow going in whatever direction is necessary to get from the first vertex to the second.

Minimum spanning trees

A **minimum spanning tree (MST)** for a weighted graph is a spanning tree for which the total of the weights of its edges is no more than the total for any other spanning tree. That is, out of all spanning trees that the graph has, none is "lighter" in total weight.

Reminder A spanning tree is a subgraph which includes all NV vertices but has no cycles. The number of edges in a tree with NV vertices is NV-1 (since each vertex except the root is at the head of exactly one edge in the tree).

You could find a minimum spanning tree for a weighted graph by considering all possible collections of NV-1 edges, checking each collection to see if it forms a tree and, if so, calculating its total weight. Then take one with the smallest total (there may be several). The execution time for this algorithm is exponential in the number of edges.

But a simple **greedy algorithm** can find the answer much faster. What we do is make a sequence of NV-1 choices, adding one edge at each choice to a subgraph that is initially empty but gradually grows to be a MST. A greedy algorithm chooses the edge each time which saves as much weight as possible. Computer scientists named Kruskal and Prim have found two moderately different ways to do this. The underlined parts of the following two descriptions are the only significant differences between the two algorithms.

Kruskal's algorithm: For the first choice, choose the lightest edge of all and add it (plus its two end points) to the subgraph. For each additional choice, choose the lightest edge that connects two vertices for which no path yet exists between them in the subgraph. If at any point along the way this gives you several different edges of the same weight, it makes no difference which one you pick.

Prim's algorithm: For the first choice, choose the lightest edge exiting any chosen vertex and add it (plus its two end points) to the subgraph. For each additional choice, choose the lightest edge that connects two vertices for which you already have exactly one in the subgraph so far. If at any point along the way this gives you several different edges of the same weight, it makes no difference which one you pick.

In Figure 19.7, the same graph is shown twice, with the weight of each edge marked above or to the side of the edge. On the left, Kruskal's algorithm would first choose the edge between 1 and 4, since it has the smallest weight of all. Second, Kruskal's would choose the edge between 6 and 7, since it is the second lightest. On the right, Prim's algorithm starting from vertex 3 would choose the edge between 3 and 2, since it has the smallest weight of all edges out of vertex 3. Second, Prim's would choose the edge between 2 and 6, since it has the smallest weight of all edges out of either 2 or 3.

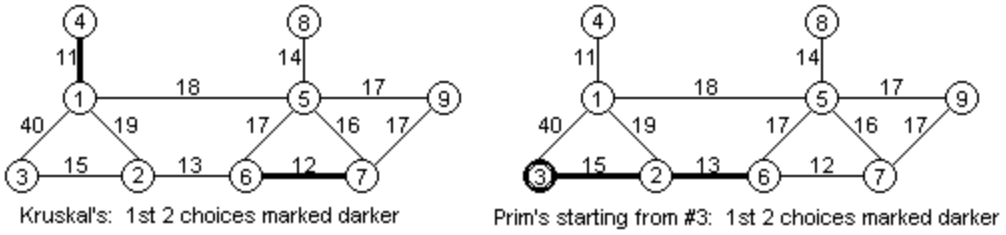


Figure 19.7 The first two edges chosen in the two algorithms

Kruskal's next choice is the edge between 2 and 6, then the edge between 5 and 8, because those are the two lightest still left. Prim's next choice is the edge between 6 and 7, then the edge between 7 and 5, because those are the lightest that connect to vertices already in the subgraph. Neither algorithm will ever choose the edge between 5 and 6.

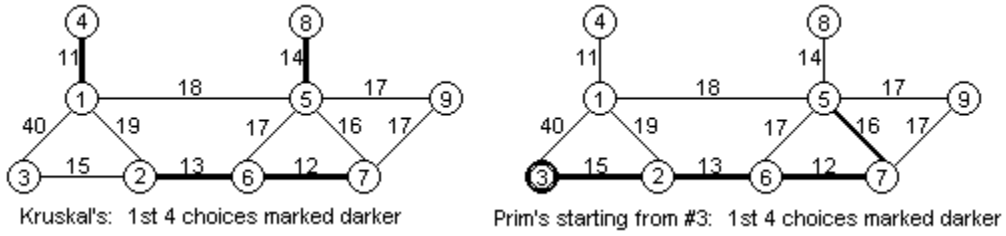


Figure 19.8 The first four edges chosen in the two algorithms

The correctness of the two algorithms

Any time you have an algorithm proposed as a solution to a problem, you have to find out at least three things about it: (a) Does it always give the right answer? (b) How do you code it? (c) How fast does it execute? We begin with the first question.

Both algorithms can be expressed with the following structured design:

1. Let X be an empty set of edges.
2. Do the following NV-1 times...
   2a. Add to X the lightest edge that satisfies the required property.

We define X[k] to be the set X after k iterations of this loop. So X[0] is the empty set of edges, X[1] contains just one edge, X[2] contains two edges, etc. X[NV-1] is the final result.

Proof of the correctness of these two algorithms

Theorem: Prim's algorithm and Kruskal's algorithm each produces an MST.

Proof part 1, that X[NV-1] is an MST: Any MST contains NV-1 edges. X[NV-1] contains NV-1 edges (obviously) and X[NV-1] is contained in some MST (this requires further proof; see part 2). Therefore, X[NV-1] must be the whole MST.

Proof part 2, that X[NV-1] is contained in some MST: X[0] is contained in some MST (obviously, since it is empty), and X[k+1] is contained in some MST whenever X[k] is (this requires further proof; see part 3). So by the induction principle, X[NV-1] is contained in some MST.

Proof part 3, that X[k+1] is contained in some MST if X[k] is contained in some MST:

1. Define <v,w> to be the edge added to X[k] to obtain X[k+1]. Define `ralph` to be the MST known to contain X[k]. If <v,w> is in `ralph`, we are done. Otherwise continue through steps 2-10 below.
2. There is some path from v to w in `ralph` (because `ralph` spans the graph).
3. For Prim's, v is in an edge of X[k] but w is not. So there is a first edge <x,y> along the path of step 2 for which x is in an edge of X[k] but y is not.
4. For Kruskal's, we cannot get from v to w using only edges in X[k]. So there is a first edge <x,y> along the path of step 2 for which you cannot get from x to y using only edges in X[k].
5. The edge <v,w> is at least as light as the edge <x,y> (since otherwise the algorithms would have chosen <x,y> instead of <v,w> to be added to the subgraph).
6. Define M to be the subgraph built by having <v,w> replace <x,y> in `ralph`.
7. M contains X[k+1] (since `ralph` contains X[k], M contains all of the edges of X[k] plus the edge <v,w> that was added to X[k] to get X[k+1]).
8. M is a spanning tree (since you can still get from x to y within M, by following the path of step 2 from x back to v, then taking the edge <v,w>, then following the path of step 2 from w back to y).
9. M has no more total weight than `ralph` (by step 5, since M is `ralph` except that <v,w> replaces <x,y>).
10. So M is a minimal spanning tree containing X[k+1] (from steps 7, 8, and 9).

It may help you understand Kruskal's algorithm better if you contrast it with a different algorithm for finding a minimum spanning tree, to wit, Jones's algorithm:

1. Let X be the set of all the NE edges.
2. Do the following until X has only NV-1 edges...
   2a. Delete the heaviest edge that connects two vertices for which there is another path in X between them.

This algorithm can be proven logically to give the correct answer. However, it has the longest execution time of the three, which is why it is not written up in more books. For one thing, the basic loop of step 2 executes NE-NV times, which is substantially larger than the basic loop for Kruskal's algorithm except for very "sparse" graphs (less than two edges per vertex on average). For another, the time required inside the basic loop to see if there is another path between two vertices is around big-oh of NE time.

Coding and the big-oh

Listing 19.11 (see next page) gives the coding for Kruskal's algorithm. Lines 2-3 put all the edges on a priority queue in increasing order of weight; this along with the taking off the priority queue can be done in NE*log(NE) time using a heap (which was described in Section 18.7, but you do not need to have read it for this section). We make sure that a two-way edge only goes on the list once, when its tail is smaller than its head.

Reminder The three basic PriQue methods for priority queues are as follows:

- `q.add(x)` to add a value x;
- `q.removeMin()` to remove a value and return it; and
- `q.isEmpty()` to see whether any values are left.

The HeapPriQue constructor has a parameter of type Comparator, which is an object that supplies a function for deciding the priority of elements on the queue. This function is the `compare` method in the ByWeight class at the bottom of Listing 19.11.

Line 3 calls a method to construct the priority queue of edges in the MST, which executes in big-oh of NE*log(NE) time. Line 4 calls a method that creates a UnionFind object that partitions the vertices so that each is in a separate group by itself. Then it uses a while-loop that executes NE times and contains a call of `unite` whose average execution time is big-oh of log(NV)/2. So the overall execution time of `constructMST` is big-oh of NE*log(NV)/2. Since the big-oh of the `Kruskals` method is the largest of these values, it has an execution time that is big-oh of NE*log(NE).

By contrast, Prim's algorithm maintains a priority queue only of the edges that run from a vertex in the subgraph so far to a vertex not in the subgraph so far, in increasing order of weight. Instead of keeping the vertices in many groups according to which vertices are connected to each other, it only has two groups: those vertices that are on an edge in the subgraph so far, and those that are not. If the priority queue of edges is implemented using a heap, then the average time to put an edge on the queue and take it off again is big-oh of log(NE) time. So Prim's algorithm also has an overall execution time of big-oh of NE*log(NE).

Both of these algorithms actually execute in big-oh of NE*log(NV) time, because that is the same as big-oh of NE*log(NE) time. The reason is that the number of edges must be less than the square of the number of vertices, so $\log(NE) < \log(NV^2) = 2*\log(NV)$.

Exercise 19.43 Complete the discussion of the process for Prim's algorithm, starting from the situation on the right side of Figure 19.8.
Exercise 19.44 Tell which edges are removed in what order, and why, in the application of Jones's algorithm to the graph in Figure 19.8.
Exercise 19.45* Complete the discussion of the process for Kruskal's algorithm, starting from the situation on the left side of Figure 19.8.
Exercise 19.46** Essay: Explain why, for a graph where all edges have different weights, every MST contains the lightest edge exiting every vertex. Note that this is a stronger assertion than what Kruskal's algorithm indicates, namely, that the lightest edge exiting any particular vertex is in some MST.

Listing 19.11 Kruskal's algorithm in the Graph class for finding an MST

```
/** Return a stack containing the edges on an MST for an
 *  undirected graph.  If the graph is not connected, the
 *  edges should form one MST for each connected part. */

public StackADT Kruskals()
{  StackADT spanner = new NodeStack();                    //1
   PriQue queue = new HeapPriQue (new ByWeight());        //2
   putEdgesOnPriorityQueueByWeight (queue);               //3
   constructMST (spanner, queue, getNumVertices());       //4
   return spanner;                                        //5
} //============================================


/** Put all Edges of the Graph on the priority queue with
 *  lighter Edges having higher priority.
 *  Precondition:  the queue is not null.  */

private void putEdgesOnPriorityQueueByWeight (PriQue queue)
{  Iterator it = vertices();                              //6
   while (it.hasNext())                                   //7
   {  Iterator edges = edgesFrom ((Vertex) it.next());    //8
      while (edges.hasNext())                             //9
      {  Edge e = (Edge) edges.next();                    //10
         if (e.TAIL.ID < e.HEAD.ID)                       //11
            queue.add (e);                                //12
      }                                                   //13
   }                                                      //14
}   //============================================


/** Check each Edge in order from lightest to heaviest.  Put
 *  the Edge in spanner whenever its two endpoints do not
 *  have a path between them using Edges already in spanner.
 *  Precondition:  spanner and queue are not null.  */

private static void constructMST (StackADT spanner,
                PriQue queue, int numLeft)
{  UnionFind group = new UnionFind (numLeft + 1);         //15
   while (! queue.isEmpty() && numLeft > 1)               //16
   {  Edge e = (Edge) queue.removeMin();                  //17
      if (! group.equates (e.TAIL.ID, e.HEAD.ID))         //18
      {  spanner.push (e);                                //19
         group.unite (e.TAIL.ID, e.HEAD.ID);              //20
         numLeft--;                                       //21
      }                                                   //22
   }                                                      //23
} //============================================


private static class ByWeight implements java.util.Comparator
{
   public in compare (Object one, Object two)
   {  double diff = ((Edge) one).WEIGHT - ((Edge) two).WEIGHT;
      return diff > 0 ? 1 : diff < 0 ? -1 : 0;            //25
   }
} //============================================
```

19.8 The Shortest Path Problem: Dijkstra's Algorithm

In a weighted graph, it is often very useful to know the cheapest path from a given vertex to every other vertex. For instance, you may live in a town situated at the source vertex and want to know how little you can get by with to travel to any other town. You would have to know what each individual part of the trip costs, i.e., the weight of the edge from each vertex to each other vertex. Or if a company were to send messages over a network to various other points in the network, it would want to know the shortest distance from its position to each other vertex in the weighted graph.

Dijkstra's algorithm

Dijkstra's algorithm is a greedy algorithm. It fills in and returns an array of double values, one per vertex, such that `dist[k]` is the shortest distance from the given starting vertex `start` to the vertex with id `k`. It leaves `dist[k]` zero if you cannot reach vertex `k` from `start`.

You start by putting all the vertices you can reach in one step from `start` on a list of vertices conventionally called the "fringe". You also set `dist[k]` equal to the length of the edge from `start` to the vertex with ID `k`, for each of those vertices on the fringe list, since you know you can reach them in that distance (or possibly less). It turns out later to be convenient to set `dist[start.ID]` to -1.

Next you find the vertex `best` on the fringe list with the shortest `dist[k]` value. That will be the shortest distance to that vertex. Remove `best` from the list. Then for each edge `e` whose tail is `best`, put `e.HEAD` on the fringe list, setting `dist[e.HEAD.ID]` to `newDist`, which is the distance from `start` to `best` plus the weight of the edge from `best` to the vertex added.

Exception: If `e.HEAD` already has a non-zero value for its distance, that means you have previously found a path to it from `start` and it is already on the fringe list. So do not add it (again) to the fringe list, and replace its `dist[e.HEAD.ID]` by `newDist` only if that is less than `dist[e.HEAD.ID]`. In other words, you make a change only when you find a shorter path than any path you had previously found to `e.HEAD`.

Repeat the process of finding the best (closest) vertex on the fringe list, removing it, and putting on the fringe list all vertices you can reach from best that you could not reach before, as well as updating the `dist` array. When the fringe list is empty, `dist` tells the shortest distance from `start` to each other vertex that is reachable from `start`. Example: For Figure 19.8 with `start` being vertex 3, the first distance found is the 15 from 3 to 2, then the 28 from 3 to 6 by way of 2, then the 34 from 3 to 1 by way of 2.

The coding for this algorithm is the `Dijkstras` method in the upper part of Listing 19.12. It uses an ArrayList for the fringe list, since it is reasonably efficient to do so. The `updateFringe` method (line 3) iterates through all the edges `e` exiting `start` and adds `e.HEAD` to the fringe list. It also assigns to each vertex the shortest distance known so far from `start` to that vertex. Then the method loops as long as the fringe list is not empty, finding the vertex `best` on the fringe list with the shortest distance (line 6) and readjusting the fringe list and `dist` array accordingly.

Execution time for Dijkstra's algorithm is as follows: The while-statement in line 5 executes NV times, each time calling `findClosest`, which iterates NV times, and then calling `updateFringe`, which iterates less than NV times. So the overall execution time is big-oh of NV^2. However, use of a priority queue can improve this (see the exercises).

Listing 19.12 Dijkstra's algorithm in Graph class for finding shortest paths from a vertex

```
/** Return an array of doubles in which the kth component is
 *  the minimum distance from the given vertex to vertex #k.
 *  A component value of 0.0 means it is not reachable. */

public double[] Dijkstras (Vertex start)
{  ArrayList fringe = new ArrayList();                    //1
   double[] dist = new double [getNumVertices() + 1];     //2
   updateFringe (start, fringe, dist);                    //3
   dist[start.ID] = -1.0;                                 //4
   while (! fringe.isEmpty())                             //5
   {  Vertex best = findClosest (fringe, dist);           //6
      updateFringe (best, fringe, dist);                  //7
   }                                                      //8
   return dist;                                           //9
}  //=============================================


/** Add to the fringe each vertex reachable from vert but
 *  not previously processed. Update the shortest distance
 *  array for any vertex that is reachable from vert. */

private void updateFringe (Vertex vert, ArrayList fringe,
                           double[] dist)
{  Iterator it = edgesFrom (vert);                        //10
   while (it.hasNext())                                   //11
   {  Edge e = (Edge) it.next();                          //12
      double newDist = dist[vert.ID] + e.WEIGHT;          //13
      if (dist[e.HEAD.ID] == 0.00) //not on fringe list   //14
      {  dist[e.HEAD.ID] = newDist;                       //15
         fringe.add (e.HEAD);     // add to list          //16
      }                                                   //17
      else if (newDist < dist[e.HEAD.ID])                 //18
         dist[e.HEAD.ID] = newDist;                       //19
   }                                                      //20
} //============================================


/** Find and return the vertex in the fringe that has the
 *  smallest dist value, after removing it from the fringe. */

private Vertex findClosest (ArrayList fringe, double[] dist)
{  Vertex best = (Vertex) fringe.get (0);                 //21
   int indexBest = 0;                                     //22
   for (int k = 1;  k < fringe.size();  k++)              //23
   {  Vertex v = (Vertex) fringe.get (k);                 //24
      if (dist[v.ID] < dist[best.ID])                     //25
      {  best = v;                                        //26
         indexBest = k;                                   //27
      }                                                   //28
   }                                                      //29
   fringe.remove (indexBest);                             //30
   return best;                                           //31
} //============================================
```

Correctness of Dijkstra's algorithm

How do we know that Dijkstra's algorithm always gives the shortest path from `start` to each other vertex? At any point when the loop tests `! fringe.isEmpty()` in line 5, let us say the "cloud" is `start` plus all of the vertices found to be best up to that point (and thus removed from the fringe). For purposes of this discussion, treat `start` as being at a distance of -1 from itself. We claim that each vertex in the cloud does in fact have recorded in the `dist` array the shortest distance possible from `start`.

This is trivially true the first time line 5 is evaluated (when `start` is the only vertex in the cloud). Suppose it is true the Nth time line 5 is evaluated (N is 1 or more). Then line 6 finds a `best` value. Let w be the last cloud vertex on the shortest path from `start` to `best`. Then w must come immediately before `best` on that path (otherwise the vertex after w must be in the fringe and have a smaller `dist` value than `best`'s, so it would have been chosen instead of `best`). Since w was in the cloud the Nth time line 5 was evaluated, `dist[w]` is the shortest distance possible from `start` to w, so `dist[best]` is the length of that shortest path from `start` through w. Thus we have an inductive proof of the correctness of Dijkstra's algorithm.

Directed graphs

Dijkstra's algorithm finds the shortest paths from a given vertex in a directed graph as well as in an undirected graph, in as little as big-oh of NE*log(NV). In a **directed acyclic graph** (DAG for short; acyclic means there are no cycles), topological sorting can find either the shortest path or the longest path in big-oh of NE time, even with some negative weights on the edges.

The **transitive closure** of a graph is the graph in which there is an edge from vertex v to vertex w when (and only when) there is a path from v to w in the original graph. You can add edges to an undirected graph that turn it into its transitive closure in big-oh of NE*log(NV) time if you make good use of the stack of edges produced by Kruskal's algorithm. This is left as an exercise. But computing the transitive closure of a directed graph is trickier; this is left as a major programming project.

Exercise 19.47 For Figure 19.8, after Dijkstra's algorithm finds the shortest paths from vertex 3 to vertices 2, 6, and 1, what are the next two shortest paths it finds?
Exercise 19.48 Explain why it helps to set `dist[start.ID]` to -1 instead of leaving it 0. Why is the assignment made in line 4 instead of immediately, after line 2?
Exercise 19.49 Add a QueueADT parameter to the `Dijkstras` method and put on it all the vertices you can reach from `start` in increasing order of the shortest distances from `start`, with the closest one at the front of the queue.
Exercise 19.50 Improve execution of the `findClosest` method by replacing the value at index `closest` by the last value in the ArrayList and then eliminating the last value.
Exercise 19.51 (harder) Essay: Explain why using a priority queue of vertices is problematic in Dijkstra's algorithm, even though finding the best would execute faster.
Exercise 19.52* Essay: Explain how best to modify Listing 19.12 to use a priority queue of <Vertex,double> objects. Use marks to correct for multiple appearances of a Vertex.
Exercise 19.53* Add an array parameter named `lastEdge` to `Dijkstras` method and assign `lastEdge[k]` to be the last edge in the shortest path from `start` to the vertex with ID `k`. This is used in the following exercise.
Exercise 19.54* Write a method that accepts two vertices as parameters, calls `Dijkstras` method with the `lastEdge` array parameter described in the preceding exercise, and then prints out the shortest path from the first vertex to the second vertex.
Exercise 19.55* Essay: Explain clearly how you would use the stack produced by Kruskal's algorithm to compute the transitive closure of an undirected graph.

*19.9 Dynamic Programming (*Enrichment)*

The Fibonacci sequence of numbers is 1, 1, 2, 3, 5, 8, 13, 21,..., where each number is the sum of the two numbers before it: `fib(n) = fib(n-2) + fib(n-1)` except the first two numbers `fib(0)` and `fib(1)` are 1. This sequence appears in several different situations.

For example, if a pair of rabbits requires 1 month to mature and then gives birth to a pair of rabbits each month thereafter, and you start with one new-born pair of rabbits, you have 1 pair at the start (`fib(0)`) and at the end of 1 month (`fib(1)`). A birth gives you 2 pairs at the end of 2 months (`fib(2)`) and another birth gives you 3 pairs at the end of 3 months (`fib(3)`). Then the pair born at the end of month 2 starts producing, which gives you 5 pairs at the end of 4 months (`fib(4)`), 3 of which are mature. So at the end of 5 months, you have your 5 pairs plus 3 more pairs produced by the mature ones, which gives you 8 pairs altogether, of which 5 pairs are now mature, etc.

An obvious but horribly inefficient method of calculating Fibonacci numbers is the following:

```
public static int fib (int n)
{  return n <= 1  ?  1  :  fib (n - 2) + fib (n - 1);
} //===============================================
```

The problem with this coding is that you calculate the answer to the same problem many times over. For instance, `fib(10)` calculates `fib(8)` twice, once directly and once as part of the calculation of `fib(9)`. This leads to the calculation of `fib(7)` 3 times, the calculation of `fib(6)` 5 times, the calculation of `fib(5)` 8 times, etc. Those numbers of times are themselves Fibonacci numbers.

The basic kind of **dynamic programming** stores the solutions to subproblems in an array of values so that, the next time it is to calculate the same result, it looks it up in the array instead of recalculating. For the Fibonacci sequence, you could have an array named `answer` for which (1) `answer[k]` initially stores 0; (2) when `fib(k)` is first calculated, `answer[k]` stores that value in place of 0; (3) any later time `fib(k)` is needed, the value is taken from `answer[k]`. This leads to the coding in Listing 19.13. We make the restriction that the method is not called for any value greater than 10000.

Listing 19.13 The Fibonacci function using dynamic programming

```
public class Fibonacci
{
   public static final int MAX = 10000;
   private static int[] answer = new int[MAX];


   /** Return the nth Fibonacci number, where fib(0) = 1 and
    *  fib(1) = 1 and otherwise fib(n) = fib(n-2) + fib(n-1).
    *  But return 1 if n >= MAX. */

   public static int fib (int n)
   {  if (n <= 1 || n >= MAX)
         return 1;
      if (answer[n] == 0)  // the first request for this value
         answer[n] = fib (n - 2) + fib (n - 1);
      return answer[n];
   }  //===============================================
}
```

Look what happens when this function is first called for n == 5: It calculates `fib(n-2)` which is `fib(3)`. That fills in `answer[3]` with 3 and `answer[2]` with 2. Then it calculates `fib(n-1)`, which is `fib(4)`. That calls `fib(3)` and `fib(2)`, which simply return 3 and 2, storing the 5 in `answer[4]`. Then 3+5 is stored in `answer[5]` and returned for the original call. Any future call of the method will only go as deep as `fib(5)` and `fib(4)` before terminating recursion and returning the answer previously calculated. So execution time is reduced from exponential big-oh time to big-oh of n.

Counting comparisons for the merge sort

The merge sort logic divides a set of values to be sorted into two equal halves, except one half is 1 larger than the other half if you are sorting an odd number of values. Then it sorts each half using the merge sort logic. Finally, it merges the two halves together into one long sorted list, in a process that requires at most `n-1` comparisons if the original list has n values. The question is, what is `compsMS(n)`, the number of comparisons required to sort n values in the worst case?

An obvious but very inefficient method of calculating `compsMS` is the following:

```
   public static int compsMS (int n)
   {  return n <= 1  ?  0
             :  compsMS (n / 2) + compsMS (n - n / 2) + n - 1;
   } //=============================================
```

The problem with this coding is that you calculate the answer to the same problem many times over. For instance, `compsMS(16)` calculates `compsMS(8)` twice, and each of those calculates `compsMS(4)` twice for a total of 4 times.

A dynamic programming solution reduces the execution time from big-oh of n to big-oh of log(n). The straightforward solution is in Listing 19.14. Applying it several times gives the following table of values:

```
n =            2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
compsMS(n) =   1  3  5  8 11 14 17 21  25 29 33 37 41 45 49 54
```

Listing 19.14 The compsMS function using dynamic programming

```
public class MergeCount
{
   public static final int MAX = 10000;
   private static int[] answer = new int[MAX];


   /** Return the nth compsMS number, which is the maximum
    *  number of comparisons required to sort n values using
    *  the merge sort logic.  But return 0 if n >= MAX. */

   public static int compsMS (int n)
   {  if (n <= 1 || n >= MAX)
         return 0;
      if (answer[n] == 0)  // the first request for this value
         answer[n] = compsMS (n / 2) + compsMS (n - n / 2) + n-1;
      return answer[n];
   }  //=============================================
}
```

You might have noticed a pattern in this table: Add 2 each time up to n==4, then add 3 each time up to n==8, then add 4 each time up to n==16, then start adding 5 each time. For instance, `compsMS(32)` is `2 * compsMS(16) + 31` which is 129, which is 80 more than `compsMS(16)`, which means you add 5 every time as you go from 16 up to 32.

The number of binary trees

How would you count the number of different shapes of binary search trees with 6 nodes? You could categorize them according to which data value the root node holds. The root could contain the first data value, thus it would have a 5-node tree as its right subtree. Or it could contain the second data value, thus it would have a 4-node tree on its right and a 1-node tree on its left. If `trees(n)` is the number of different binary search trees with n nodes, then there are `trees(5)` cases with the root node the first data value and `trees(4)` cases with the root node the second data value.

If the root contains the third data value, it has a 3-node tree on its right and a 2-node tree on its left. Since there are 2 different 2-node trees and 5 different 3-node trees (draw the pictures and see), you multiply to see there are `2 * 5` possible combinations. That is, there are `trees(2) * trees(3)` cases with the root node the third data value.

Since the other alternatives are symmetric to those three, you have to double the total, so you have the following formula:

```
trees(6) = 2*trees(5) + 2*trees(4) + 2*trees(3)*trees(2)
```

This process can be generalized to calculate `trees(n)` for any positive value n. You must make a small adjustment for odd values of n, however; the number of trees with the root in the exact center is not doubled in its calculation.

There is only 1 tree of size 1 and there are only 2 trees of size 2, so a not-so-obvious yet horribly inefficient method of calculating `trees` is as follows:

```
public static int trees (int n)
{  if (n <= 2)
      return n;
   int total = trees (n - 1);  // root at far left
   for (int k = 2;  k <= n / 2;  k++)
      total += trees (k - 1) * trees (n - k);
   return (n % 2 == 0)  ?  2 * total
          :  2 * total + trees (n / 2) * trees (n / 2);
} //================================================
```

A dynamic programming solution reduces the execution time greatly, from exponential big-oh time to only $n^2/2$ additions and multiplications performed. The coding is in Listing 19.15 (see next page). There are 14 of the 4-node trees, 42 of the 5-node trees, and 132 of the 6-node trees.

The basic approach in all three of these situations is the same: After you write an algorithm recursively, check whether the same subproblems are being solved many times over. If so, store the result of the first calculation in an array, generally indexed by the parameter(s), and retrieve that result when the problem is to be re-solved.

Listing 19.15 The trees function using dynamic programming

```
public class TreeCount
{
   public static final int MAX = 10000;
   private static int[] answer = new int[MAX];

   /** Return the nth trees number, which is the number of
    *  different binary trees.  But return n if n >= MAX. */

   public static int trees (int n)
   {  if (n <= 2 || n >= MAX)
         return n;
      if (answer[n] == 0)  // the first request for this value
      {  int total = trees (n - 1);
         for (int k = 2;  k <= n / 2;  k++)
            total += trees (k - 1) * trees (n - k);
         answer[n] = (n % 2 == 0)  ?  2 * total
                   : 2 * total + trees (n / 2) * trees (n / 2);
      }
      return answer[n];
   }  //=============================================
}
```

The longest common subsequence problem

To test whether two individuals have a genetic relationship, scientists look at their DNA strands and see if they have the same long subsequence. Dynamic programming can help find it. A **subsequence** `sub` of a string s is a sequence of characters that appear in s in the same order they appear in `sub`, though not necessarily right next to each other in s. For instance, "CAT" has seven subsequences of positive length: C, A, T, CAT, CA, AT, and CT. Notice that the last one is not a substring of "CAT".

The general problem is, given two strings s and t, to find the longest subsequence of characters that s and t have in common. We could list all possible subsequence of s and check each one against t, but there are $2^{s.length()}$ subsequences of s, which makes this an exponential solution.

Let LCS stand for the phrase "longest common subsequence". Let `longest(s,t)` denote the length of the longest subsequence of the strings s and t. For convenience, let `s'` denote the result of removing the last character of s. We begin by noticing that, if the last characters of s and t are the same, then their longest substring includes that last character, so the following is true:

```
longest(s, t) = longest(s', t') + 1
```

On the other hand, if the last characters of s and t are different, then their LCS must be either the LCS of s and t' or else the LCS of s' and t, whichever is longer:

```
longest(s, t) = Math.max (longest(s', t), longest(s, t'))
```

The following method is a straightforward but quite inefficient method of calculating the longest value, assuming it can directly access the Strings s and t and is passed as parameters the number of initial characters of each String that it is to find the LCS of:

```
public static int longest (int len1, int len2)
{  if (len1 == 0 || len2 == 0)
      return 0;
   else if (s.charAt (len1 - 1) == t.charAt (len2 - 1))
      return longest (len1 - 1, len2 - 1) + 1;
   else
      return Math.max (longest (len1 - 1, len2),
                        longest (len1, len2 - 1));
} //============================================
```

A dynamic programming solution reduces the execution time greatly. The coding is in Listing 19.16 (see next page). There are some obvious differences from the preceding three listings: The public method is originally passed the two strings rather than two int values. So it checks the two strings to make sure they are non-null and not too long to handle. Then it sets up a two-dimensional matrix `answer[i][j]` that tells the length of the LCS that can be formed with the first i characters of the first string and the first j characters of the second string. This matrix cannot be filled with zeros as placeholders, since zero is a legitimate answer to the length of a common subsequence, so the method fills the array with -1. For speed, it also fills in the length 0 of the LCS for the cases where one of the substrings has no characters. Then it calls a private method to calculate the answers.

After the answer is calculated, the client of this class will find it useful to see what values were calculated for various initial substrings of the given strings. A public method named `lcs` is provided for that purpose. The client can easily find out the exact subsequence of characters that is longest by using the public `lcs` method. This is left as an exercise. Execution time is big-oh of `s.length() * t.length()` for this dynamic programming method to fill in the two-dimensional array. The execution time for finding the actual longest common subsequence of characters is `s.length() + t.length()`.

Once you have this straightforward solution, further analysis shows that you can increase the efficiency somewhat by replacing the body of the private `longest(int,int)` method down to the return statement by the following non-recursive equivalent:

```
for (int row = 1;  row <= len1;  row++)
   for (int col = 1;  col <= len2;  col++)
   {  if (one.charAt (row - 1) == two.charAt (col - 1))
         answer[row][col] = answer[row - 1][col - 1] + 1;
      else if (answer[row - 1][col] > answer[row][col - 1])
         answer[row][col] = answer[row - 1][col];
      else
         answer[row][col] = answer[row][col - 1];
   }
```

Listing 19.16 The LCS function using dynamic programming

```
public class LongestCommonSubsequence
{
   public static final int MAX = 1000;
   private static int[][] answer = new int[MAX][MAX];
   private static String one;
   private static String two;

   /** Return the length of the longest common subsequence of
    *  the two given Strings. Return 0 if either is null or
    *  has MAX or more characters.  */

   public static int longest (String first, String second)
   {  if (first == null || second == null
               || first.length() >= MAX || second.length() >= MAX)
         return 0;
      for (int row = 0;  row <= first.length();  row++)
      {  for (int col = 0;  col <= second.length();  col++)
            answer[row][col] = row * col == 0  ?  0  :  -1;
      }
      one = first;
      two = second;
      return longest (one.length(), two.length());
   }  //=============================================

   private static int longest (int len1, int len2)
   {  if (answer[len1][len2] >= 0)
         return answer[len1][len2];
      if (one.charAt (len1 - 1) == two.charAt (len2 - 1))
         answer[len1][len2] = longest (len1 - 1, len2 - 1) + 1;
      else
         answer[len1][len2] = Math.max (longest (len1 - 1, len2),
                                        longest (len1, len2 - 1));
      return answer[len1][len2];
   }  //=============================================

   /** Return the length of the LCS of the first len1 characters
    *  of the first String and the first len2 characters of the
    *  second String.  Throw an Exception for bad indexes. */

   public static int lcs (int len1, int len2)
   {  return answer[len1][len2];
   }  //=============================================
}
```

Exercise 19.56 List the next six Fibonacci numbers after 13 and 21.
Exercise 19.57 The double for-loop in the public `longest` method of Listing 19.16 will execute faster if you replace it by coding that fills in zeros separately from the -1s. Do so.
Exercise 19.58* Write a method that prints the longest common subsequence of s and t after calling `LongestCommonSubsequence.longest(s, t)`.

19.10 Review Of Chapter Nineteen

- A graph consists of a number of **Vertex** objects and **Edge** objects. Each Edge object runs from one Vertex object, its **tail**, to another Vertex object, its **head**. The graph is **undirected** if every edge from vertex v to vertex w is matched by an edge from vertex w to vertex v. In that case we generally treat both edges as one object.
- The **out-degree** of a vertex is the number of edges exiting the vertex, and the **in-degree** of a vertex is the number of edges entering the vertex.
- A **path** in a graph is a sequence of edges where the head of each edge is the tail of the next. A graph is **connected** if you can get from any one vertex to any other vertex by following edges in the graph, i.e., if there is a path from any one vertex to any other vertex.
- The two standard ways to implement a graph are the **adjacency list** (keeping a list of the edges exiting each vertex) and the **adjacency matrix** (keeping a matrix whose <v,w> component is the edge from vertex v to vertex w). The former is faster for depth-first traversals, but the latter is faster for seeing whether an edge exists.
- A **topological sort** of a graph is an ordering of the vertices so that, for each edge from some vertex v to some other vertex w, v comes before w in the sequence.
- A **subgraph** of a graph G is a subset of the edges of G plus the heads and tails of those edges. A **spanning tree** of a graph is a subgraph containing all the vertices of the graph and having exactly one path from the root vertex to any other vertex. A **minimum spanning tree** for a **weighted graph** (where edges have a positive weight value) is a spanning tree for which the total weight of the edges is the least possible.
- **Kruskal's algorithm** and **Prim's algorithm** find a minimum spanning tree for a connected graph. **Dijkstra's algorithm** finds the shortest paths from a given vertex to all other vertices.

Answers to Selected Exercises

19.1
```
public void makeTheGraphConnected()   // in HamGraph
{     while ( ! isConnected())
            addEdgeChosenRandomly();
}
```
19.2
```
public int inDegree (Vertex given)   // could be in abstract Graph
{     int count = 0;
      for (int k = 1;  k <= getNumVertices();  k++)
      {     if (contains (new Edge (getVertex (k), given)))
                  count++;
      }
      return count;
}
```
19.9
```
public void removeAllEdgesFrom (Vertex v)  // for MatrixGraph
{     for (int k = 1;  k <= getNumVertices();  k++)
            edgeAt[v.ID][k] = null;
}
```
19.10
```
public int outDegree (Vertex given) // for MatrixGraph
{     int count = 0;
      for (int k = 1;  k <= getNumVertices();  k++)
      {     if (edgeAt[given.ID][k] != null)
                  count++;
      }
      return count;
}
```
19.11 Same as for the preceding exercise except reverse the order of indexes: edgeAt[k][given.ID].

19.12
```
public Iterator edgesFrom (Vertex v)  // for MatrixGraph
{     ArrayList list = new ArrayList();
      for (int k = 1;  k <= getNumVertices();  k++)
      {     if (edgeAt[v.ID][k] != null)
                  list.add (edgeAt[v.ID][k]);
      }
      return list.iterator();
}
```

19.13 Declare a new instance variable: private int[] itsIns;
In the constructor, add: itsIns = new int [maxVerts + 1]; // initially all components are zero.
In remove(Edge), add: if (e != null) itsIns[given.HEAD.ID]--;
In add(Edge), add: if (edgeAt[given.TAIL.ID][given.HEAD.ID] == null) itsIns[given.HEAD.ID]++;

19.20 Copy the first two lines of the add method (the if statement) into both contains and remove.

19.21
```
public int outDegree (Vertex given)  // for ListGraph
{     return ((Collection) itsList.get (given.ID)).size();
}
```

19.22
```
public void removeSomeEdges (Vertex given)  // for ListGraph
{     ArrayList z = (ArrayList) itsList.get (given.ID);
      for (int k = z.size() - 1;  k >= 0;  k--)  // question: why doesn't it work in the other direction?
      {     if (((Edge) z.get (k)).HEAD.ID > given.ID)
                  z.remove (k);
      }
}
```

19.29 From 6 you visit 2, from 2 you go to 1, from 1 you go to 3. Backing up, from 1 you go to 4. Backing up, from 6 you go to 5, from 5 you go to 7, from 7 you go to 8. Finally, from 5 you go to 9.

19.30 From 6 you visit 2, 5, 7 in that order. Then from 2 you visit 1 and 3. From 5 you visit 8 and 9. From 8 and 9 there are no more visits. From 1 you visit 4, and you are done.

19.31 Now that you have visited 4, 1, 3, 2, 6 in that order, the next vertex to visit is 5 (since we assume that the edgesFrom iterator produces vertices in numeric order). From 5 you go to 7 and then 9 but you get stuck. You try again: from 5 you go to 8 but get stuck. Again: from 5 you go to 9 and then 7 but you get stuck. So you back up to 6 and go to 7, marking it 6. From 7 you go to 5 and then 8 but you get stuck. You try again: from 7 you go to 5 and then 9 but you get stuck. Again: from 7 you go to 9 (marking it 7), and then to 5 (marking it 8), and finally 8 (marking it 9). That finishes the traversal. No other traveling salesman traversal is possible with that graph.

19.32 Use the body of isConnected with these changes: Replace line 8 by: markReachables (one);
Insert after line 8: boolean valueToReturn = two.getMark() > 0;
Replace the last line of the method by: return valueToReturn;

19.33 Change the heading to: private int markDistance (Vertex given, QueueADT queue, int n).
Just before the if statement, insert: if (v.ID == n) return current.getMark() + 1;
At the end of the method, put return -1; (to signal failure).

19.40
```
private static class Node
{     public int itsData;
      public Node itsNext;
      public Node (int data, Node next)
      {     itsData = data;
            itsNext = next;
      }
}
```

19.41
```
public UnionFind (int numValues)
{     itsLength = numValues > 0  ?  numValues  :  1;
      itsItem = new Queue[itsLength];
      for (int k = 0;  k < itsLength;  k++)
            itsItem[k] = new Queue (k);
}
```

19.43 Prim's fifth choice is the edge of weight 14 between 5 and 8. Prim's sixth choice has to be an edge of weight 17. It cannot be the one between 5 and 6, because 5 is already reachable from 6. You may choose either of the two edges of weight 17 leaving vertex 9. Prim's seventh choice is the edge of weight 18 between 5 and 1, and finally the edge of weight 11 between 1 and 4.

19.44 Removing edges starting from the heaviest, you take the edge (1,3) of weight 40, then the edge (1,2) of weight 19, but skip the edge (1,5) of weight 18 (since that disconnects the graph). Then take one of the edges of weight 17 from 9 as well as the edge (5,6) of weight 17. The 8 edges left are an MST.

19.47 The path of length 40 from 3 to 7, then the path of length 45 from 3 to either 4 or 5.

19.48 Otherwise the test in line 14 of Listing 19.12 would be dist[e.HEAD.ID] == 0.0 && e.HEAD != start, which would execute slower and be harder to understand. If the assignment were before line 3, the updateFringe method would calculate the newDist as 1 less than it should be.

19.49 The method heading should be: public double[] Dijkstras (Vertex start, QueueADT queue)
Insert this statement after line 6: queue.enqueue (best);

19.50 Replace line 30 of Listing 19.12 by the following two statements:
fringe.set (closest, fringe.get (fringe.size() - 1)); fringe.remove (fringe.size() - 1);

19.51 The problem is in line 19. You have a vertex e.HEAD that has previously been added to the fringe with a certain dist value, which is what the Comparator of the priority queue would use. Line 19 reduces the dist value, so that vertex would have to be shifted in most implementations of PriQue, such as HeapPriQue or TreePriQue. Basic priority queues do not allow for such changes in priority.

19.56 13,21,34,55,89,144,233,377.

19.57
```
for (int row = 0;  row <= first.length();  row++)
      answer[row][0] = 0;
for (int col = 1;  col <= second.length();  col++)
      answer[0][col] = 0;
```
Then have the same double loop but start each index from 1 and simply assign -1 to each.

Appendix A
Guidelines for Program Style

1. Use spacing and indenting in the conventional way

1.1 Indentations from the left margin should be in units of three or more blanks. It is easiest to set your tab stops at three to five characters each (0.25 to 0.4 inches apart) and use tabs to indent. At the first level of indentation should be imports, the class heading, and the left and right braces that set off the body of the class. At the second level of indentation should be the words `private` and `public` for variable and method declarations, along with the left and right braces that set off the body of the methods.

```
import javax.swing.*;
public class Whatever
{
     private String itsFirstName = "Darryl";

     public void getFirstName()
     {    return itsFirstName;
     }
}
```

1.2 All statements should be indented at least two levels further to the right than the class heading. Each of the keywords `if`, `else`, `do`, `for`, and `while` (for statements that contain statements) should have the subordinate statements indented one level further to the right than the keyword itself. If that subordinate material is enclosed in braces, the left brace should be directly below the first letter of the keyword and the right brace should be somewhat lower down and aligned with the left brace. There are three exceptions: (a) The multiway selection structure with the `else if` combination on one line together is usually more natural, as if it were one word `elsif`; (b) the `while` that goes with a `do` should be indented somewhat further than the `do` and have a left brace just before it, so it does not look like the beginning of a new statement; (c) the `catch` that goes with a `try` should be formatted the same way for the same reason.

1.3 No other indenting to the right should be used, except for at least a double indenting for a continuation of one statement onto a second or even third line. These continuations should come just before an operator (usually +) or just before a left parenthesis or just after a comma. Do not allow a line to run over 80 characters (thereby wrapping around when it is printed); press the ENTER key at a good point and indent the continuation.

1.4 A right brace should always be lined up with the corresponding left brace and one or more lines lower than it. These braces should be the first characters on their lines. Beginning programmers will be far less likely to have unmatched or mismatched braces if they follow this Allman style.

1.5 Operators (e.g., + <= ! ? != * &&) should have at least one space on either side of them. After all, they are equivalent to an entire word or phrase in English, and you put spaces around words, don't you?

1.6 If you want to use spacing to group a sequence of statements together, put a blank line between logical subunits of a method. Put a comment heading on such subgroups if you wish. But keep the statements indented the same. And do not put blank lines between every pair of statements; that just wastes paper and does no good. In any case, the division between two methods should be stronger (more blank lines or a line of marks such as =======) than the divisions (blank lines) within methods.

2. Choose appropriate names of variables, methods, and classes

2.1 Names of final class variables or final instance variables should be entirely in capital letters. Use an underscore, as in NUMBER_OF_ITEMS, to set off separate English words in these identifiers. You may also use this form for local final variables if you wish.

2.2 All other names of variables and methods should begin with a lowercase letter. All names of classes should begin with a capital letter. All the other letters in a variable name, method name, or class name should be lowercase except for the first letter of a separate English word, which should be capitalized. Examples are variable `itsFirstName`, method `getFirstName()`, and class `ListOfFirstNames`.

2.3 Do not use a single letter for a variable name unless it is a variable that is local to a single method, and even then do so sparingly. Do not use a single letter for any method or class name. The only justification for a single letter is that it is only used in a short range of statements after its declaration.

2.4 Use names that clearly indicate the meaning of the variable, method, or class. It is generally not difficult to think of such a name except for some local variables which indicate a generic sort of value with no particular relation to other values. If you do not wish to type a longer name, even though you know it more clearly communicates the meaning, use temporarily a unique short name such as xzx and, after typing most of the uses of the name, do a search-and-replace to make it right.

2.5 Do not name a local variable the same as an instance variable, class variable, or parameter. Only name two local variables the same if the range of statements over which one can be used does not overlap with the range for the other. An acceptable use is separate cases of `for(int k =...)`.

3. Choose good comments and declarations

3.1 Put a comment on almost every method, describing what the statements accomplish and what has to be true before the method is called (the precondition). An exception can be made for those for which the name makes the meaning perfectly obvious, particularly those that are standard names such as `equals`, `compareTo`, `toString`, `getFirstName`, `setLastName`, and most constructors.

3.2 Do not put a comment on much of anything else. Anyone who would profit from a comment such as "add 1 to k" on the statement `k++` should not be reading the coding anyway. And most of the time that you use a comment to explain what a variable stands for, the variable name should be changed to incorporate that comment.

3.3 Use a named constant (final variable) for any value that is used in two or more methods of one class, except generally 0, 1, and "". The name serves as a comment describing what the value is for.

3.4 Declare the loop control variable of a for-statement inside its heading when feasible.

3.5 Declare a variable locally if possible. If not, declare it private instead of public, if feasible.

3.6 Assign a value to each class variable in its declaration. Assign a value to each instance variable in all constructors unless it is done in the declaration itself or you have a solid reason for not doing so. Initialize each local variable in its declaration or in the statement right after its declaration (the latter is sometimes necessary if you need an if-else statement to determine the initial value). Do not initialize a variable to a value that will never be accessed by any statement under any circumstances.

4. Shorter is better, all other things being equal

4.1 If you can express the logic more compactly without losing clarity, do so. In particular, if you have any segment of coding that could be eliminated without any difference in effect, eliminate it. For instance, the following two phrases should almost always be rewritten:

```
else ;     // simply omit it
if (condition) ; else ...  // rephrase as if (! condition) ...
```

4.2 If you assign a value to a variable just so you can execute only one statement using it shortly thereafter, you should usually just use the expression assigned to the variable in the statement itself. For instance, `x = y + z;... callMethod(x)` should be written as `callMethod(y + z)` if (a) those are the only uses of the variable `x`, and (b) that method call is not inside a loop that the assignment of `x` is outside of, and (c) `y` and `z` do not change their values between the two statements.

4.3 However, if an application of the above principle would produce a fairly lengthy statement (say 1.5 lines of coding) when you would otherwise have two moderately short one-liners, your coding will probably be clearer with the one-time use of a variable.

4.4 If you have an if-else statement with the same statement as the last action in both subordinate parts, factor it out by putting it after the if-else statement. And when you have the same statement as the first action in both subordinate parts, you can often factor it out by putting it before the if-else statement.

4.5 If you have a calculation inside a loop that is known to produce the same result each time, factor it out: Make the calculation outside the loop, assign it to a local variable, and use that local variable instead.

4.6 Make a method out of a group of three or more statements that would otherwise have to be written three or more times at various places in one or more classes. It is often a good idea to make a method out of an even smaller group or out of a group that is used only in two different places.

4.7 Every method method should fit on a single screen (maximum of thirty lines), unless perhaps it is a quite lengthy switch statement or it has no loops and not even much in the way of if statements (e.g., lots of output statements or lots of adding components to a container). Even better, try to keep it under fifteen statements.

4.8 If you need to process the values in a list of items one at a time, and the first one has to be processed differently from the rest, process that first one before starting the loop. And process the last value after the loop when the last value must be processed differently from the rest. For instance, avoid the following:

```
for (int k = 0;  k < someString.length();  k++)
   if (k == 0)...
```

APP-4

5. More advanced points

5.1 **Generalize**: Write a method to perform a more general task if you can do so without significantly slowing down execution of the particular task you need done. For instance, do not declare a parameter as String if Comparable will do just as well, and do not declare it as Comparable if Object will do just as well. Also, do not use a specific numeric value in a method if it works as well to pass in the numeric value as a parameter.

5.2 **Explain**: Give a prompting message when asking for input from the user at the keyboard.

5.3 **Be robust**: If an unexpected situation could throw a RuntimeException, guard the dangerous statement with an appropriate test. If it is too complex to guard against it, be prepared to catch the Exception and recover enough to go on. If you cannot usefully continue, at least print an appropriate message explaining the problem and save what you can of the work so far.

5.4 **Do one thing well**: If a method returns a value, then it is highly preferable that it not cause a change in any object (i.e., it should be what this book calls a "query method"). Two reasonable exceptions are (a) a method to get input from the user may change the input object (e.g., a file pointer), and (b) a method that takes an action may return a value that describes what action it took, such as whether its attempt to delete was successful or how many objects it modified in the process of doing what it does.

5.5 **Additional style principles used in this book**:

- Do not use `continue`. Do not use `break` except within a `switch` statement.
- Do not use `protected` or default visibility.
- Do not declare two variables on the same line unless they are very intimately related, such as `x` and `y` for coordinates of a point.
- Do not put other methods in a class with a `main` method.
- List field variables in the order (a) public class variables, (b) private class variables, (c) instance variables.
- Use "the" as a prefix of non-final class variables and "its" as a prefix of instance variables.

Appendix B Glossary of Terms

action method: a method that changes the state of a variable but does not return a value. By contrast, a query method returns a value but does not change the state of any object. Try to avoid methods that do both.

actual parameter: an expression supplied in the parentheses of a method call. By contrast, a formal parameter is a variable declared in the parentheses of a method heading. When the method executes, each formal parameter in the heading is initialized to the value of the corresponding actual parameter.

algorithm: a step-by-step procedure for solving a problem in a finite number of clearly-stated steps.

alias: a variable that refers to the same object that another variable refers to. A formal object parameter is an alias for the corresponding actual parameter.

analysis: The process of finding out the details of what a client wants a particular piece of software to do, so there is no question about what it should or should not do.

application program: a class that contains a method with the heading `public static void main (String[] args)`. An application program named `SomeClass` can then have that main method executed by the runtime system with a command that begins `java SomeClass`. The rest of the words on the command line are passed to the `args` parameter.

array: a reference to a sequence of variables that are all of the same type, e.g., to 100 int variables or to 5000 String variables. If `x` is an array variable, then the number of variables referred to is `x.length`, and you may refer to each such variable by `x[k]` where `k` is an int expression with a value in the range `0..x.length-1`.

ascending order: Each value is greater than or equal to the value just before it in the sequence.

binding: The compiler binds a method call to a particular method definition when it can, namely, for a class method or for a final instance method. This is early binding, which reduces execution time compared with late binding (done at runtime).

block: a sequence of one or more statements with a matched pair of braces `{` and `}` marking the beginning and end of the sequence.

body of a method declaration: the part of a method declaration that starts with the first left brace `{` and ends with the right brace `}` that corresponds to it, including the contents of those matched braces, is the body of the method declaration.

boolean operator: a symbol that combines one or two true-false expressions to obtain a new boolean expression. `&&` and `||` apply to two boolean expressions; `!` applies to just one.

braces: the wiggly grouping symbols `{` and `}`. Don't call parentheses `()` braces.

brackets: the straight-sided grouping symbols `[` and `]`. Don't call parentheses `()` brackets.

buffering: An output file is buffered if it saves up in RAM the small chunks of data you give it until it has a big chunk of data that it can write to the hard disk or screen all at once. An input file is buffered if it gets one big chunk of data at a time from the hard disk and saves it in RAM until you ask for it, typically in small chunks. If you have a reason to move the saved-up output to its ultimate destination before the buffering algorithm feels like doing so, you flush the buffer.

bug: a failure of the software to have an effect its specs say it should have, or any effect that the software has that its specs do not say it should have.

busywait: execution of statements that accomplish nothing except to make time pass. Used in animation by programmers who do not know how to put their thread to sleep.

byte: one byte of storage, requiring 8 bits (on/off values), so there are 256 different possible values for one byte.

cast: the placement of a type description in parentheses in front of an expression. `(int) x` tells compiler to use the value in `x` in modified form, without a decimal point. `(Sub) x` tells the compiler to treat `x` as though it refers to a member of class Sub. The cast does not change the value stored in `x`.

chaining: If a method call returns an object, it can be used as the executor of another method call. This is called chaining. An example, is `sam.getCD().getName()`, wherein the first method call returns a CD object, and the second method call asks that CD object for its name.

checked exception: a non-RuntimeException. Any method that contains a statement that can throw a checked Exception must handle the potential Exception in one of two ways: Put the statement within a try/catch statement that can catch it, or put a throws clause, `throws XXException` at the end of the method heading.

class diagram: one or more rectangles connected by arrows using UML notation. The rectangles are divided into three parts. The top part has the class name and the bottom part lists any of its method calls that you wish to mention. A dependency of the form `X uses Y` is indicated in UML by an arrow with a dotted line. A generalization of the form `X is a kind of Y` is indicated in UML by an arrow with a solid line and a big triangular head.

class method: a method that does not have an executor for the method call. It is signaled by the word `static` in the method heading. You may call a class method with the name of its class in place of an executor.

class variable: a variable declared outside of every method and with the word `static` in its declaration. It exists independently of any object of that class.

command-line argument: a String value on the command line after `java ClassName`. The command-line arguments are passed in to the `String[] args` parameter of the main method. `args.length` tells how many values are passed in.

comment: material in a program intended to explain the operation of the program to a human reader. The compiler ignores comments, which are (a) any material after // in a program, down to the end of its line, or (b) anything between /* and the next */.

compiler: a program that translates an entire file from one form to another form. A class SomeClass is stored in a plain text file typically named `SomeClass.java`, readable by humans. The compiler translates this file into a form the runtime system can use, stored in a file named `SomeClass.class`.

compile-time constant: A variable initializer is an assignment of a value to a variable at the time it is defined. A variable initializer of a class variable that can be computed by the compiler without calling any methods is done by the compiler. That class variable is called a compile-time constant.

composition: A class of objects is formed by composition if its only or its primary instance variable is an object of another class.

concatenation: the combination of two strings of characters, one attached to the end of the other.

condition: another word for a boolean expression, which is an expression that is either true or false.

conditional operator: An expression of the form `testing() ? getOneThing() : getAnother()` uses the conditional operator to obtain a value. Both the question mark and the colon are required. If `testing()` is true then `getOneThing()` gives the value; otherwise `getAnother()` gives the value. Those two parts have to be of the same type (int or boolean or Worker or whatever).

constructor: a method that instantiates a new object of the class when it is called. Within the constructor you may use `this` to refer to the object created, although a constructor is not an instance method.

continuation condition: A `while` statement states a continuation condition followed by the subordinate statements. The continuation condition must be true in order for the subordinate statements to be executed. A `for` statement and a `do-while` statement also have continuation conditions.

conversion: Assigning a char value to an int variable or an int value to a double variable causes a widening conversion, a.k.a. promotion; the compiler does this automatically. The opposite is a narrowing conversion; it requires a cast.

crash-guard: a condition that must evaluate as true in order for another expression to be evaluated, when that second expression would throw an Exception if the crash-guard condition were false, as in `(x != 0 && y > 2 / x)`. Or else a condition that must evaluate as false in order for the other expression to be evaluated, when the other expression would throw an Exception if the crash-guard were true, as in `(x == 0 || y <= 2 / x)`.

dangling else problem: a statement of the form `if (c1) if (c2) st1 else st2` where the `else` is meant to go with the first `if`, though the compiler always matches it with the second `if`.

declaration: `SomeClass sam` creates a variable of type SomeClass and declares `sam` as the name of that variable. The phrase `SomeClass sam` is a variable declaration. For instance, `String sue` declares a variable for storing a String object. The heading of a method declares what its name is, what must be supplied in the parentheses of the call, and what is returned by the call.

default: a language element supplied by the compiler because it is optional in a particular position in the coding and you do not supply it. Examples: The default constructor when you do not supply one is `public SomeClass(){super();}`. The default executor of a call of an instance method is `this`, which refers to the executor of the method the call is inside of. The default extension of a class is `extends Object`.

definition: The body of a method is the definition of what happens when you call it. By contrast, a declaration tells how to call it and how to use what it returns to you.

dependency: When a class uses methods defined in another class, that is a dependency relationship.

descending order: Each value is less than or equal to the value just before it in the sequence.

driver method: a main method whose only purpose is to test one or more method definitions.

efficiency: a measure of the time, storage space, or programmer effort required to execute an algorithm.

empty string: the String value whose length is zero. It can be written as `""`.

encapsulation: preventing outside classes from changing field variables directly (i.e., without calling a method in the class). The standard way to encapsulate is to declare field variables as `private`.

event-driven programming: attaching a listener object to a component and arranging to have a change in the component send an action message to the listener object.

exception: an object that a method throws to notify the runtime system of a problem that crashes the program (unless you have special statements that are discussed in Chapter Eleven). Examples are a NumberFormatException (thrown by an attempt to parse a string of characters as a numeral when it is not), an ArithmeticException (caused by trying to divide by zero), an IndexOutOfBoundsException (thrown by e.g. `si.charAt(k)` if it is false that `0 <= k < si.length()`), and a NullPointerException (thrown by e.g. `si.charAt(k)` if `si` has the value `null`).

executor: When a method call has a variable before the dot (period), as in `sam.moveOn()`, that variable refers to the executor of that call. If the statements within some method M include a method call whose executor is not stated, then the executor of the method called is M's executor.

field variable: a variable that is declared in a class outside of every method. It may be an instance variable or a class variable. If the declaration of a field variable assigns it a value, that takes effect for an object's instance variable when the constructor is called, for a class variable when the program begins. The runtime system supplies a default value if you do not: zero or `null` or `false`, as appropriate.

final: Putting `final` on a variable means it cannot be changed at a later point. Putting `final` on a method means it cannot be overridden. Putting `final` on a class means it cannot be subclassed.

formal parameter: a variable declared in the parentheses of a method heading. By contrast, an actual parameter is an expression supplied in the parentheses of a method call. When the method executes, each formal parameter in the heading is initialized to the value of the corresponding actual parameter.

garbage collection: recycling of an object that your program has previously created but currently has no variable that refers to it. That is, reusing space in RAM that is no longer needed.

generalization: When one class inherits from (extends) another class, it is a generalization of it.

hardware: the physical components of the computer (chip, RAM, disk, monitor, etc.). By contrast, the programs that you run and all the classes they use are software.

heading of a method declaration: the part of a method declaration up to but not including the first left brace `{`. In particular, it includes the parentheses after the method name.

identifier: all names of variables, methods, and classes.

immutable: what we call a class for which the values of instance variables cannot changed after an object has been constructed.

implementation: translation of a design or plan into actual Java coding (or another language).

independent method: a class method that can be placed in any other class because it does not refer to any instance variables or class variables. All of the information it uses to do its job is in its parameters.

inherit: to have available in a class all of the public methods defined in a superclass of the class. The phrase `extends SomeClass` in the heading of X makes SomeClass the superclass of X. Each instance of the subclass X can use those methods as if they were defined in the subclass, unless the subclass redefines them with the same name and parameter types.

initializer list: a list that follows the assignment symbol `=` in an expression of the form `Type[ ] x = {...}`. The comma-separated values listed inside those braces create the array `x` and initialize it to have the values listed.

inner class: a class that is defined inside of another class. The inner definition cannot have the word `static` in it (otherwise it is simply a nested class). Instantiation of an object of the inner class requires an object of the outer class that is to be associated with the inner object.

instance: an object created by execution of a phrase of the form `sam = new SomeClass()`. The phrase puts a reference to that object in `sam`. The newly-created object belongs to SomeClass.

instance method: a method that requires an executor for the method call. It is signaled by the absence of the word `static` in the method heading.

instance variable: a variable declared outside of every method and without the use of the word `static`. Each object of that class has its own value for that variable.

instantiating a class: Using a class's constructor to obtain a new object, i.e., creating an object of that class.

interface: a compilable unit; the heading `public interface X` means `X` cannot contain anything but non-final instance method headings and final class variables. An instantiable class with the heading `class Y implements X` must define all methods in that interface.

interpreter: a program that translates one line of source code (written by the programmer) to object code (executable by the chip) at a time. It executes each line before going on to translate the next line.

iteration: one execution of the subordinate statements of a `while` or `for` or `do-while` statement.

keyword: a word (i.e. starting with a letter) whose meaning is specified by the language. All the words in a program except names of variables, methods, and classes are keywords. Keywords are always entirely in lowercase letters, even `instanceof`. You cannot declare a keyword to have another meaning.

local variable: a variable declared within the body of a method, i.e., within the braces of the body. These local variables have no connection with variables outside of that method, and they have no initial value. You can only use a local variable after the point where it is declared and inside whatever braces containing the declaration.

loop-control variable: the only variable that appears in the continuation condition for a looping statement and is modified during execution of the loop.

main method: a method with the heading `public static void main (String[] args)`. It can be executed by the runtime system with a command of the form `java SomeClass` where `SomeClass` is the class containing the main method. Such a class is called an application program. This book rarely puts any other method in the same class with a main method, though it is legal to do so.

modular programming: the creation of software for one purpose as a number of modules (methods and classes), designed so that, when you want to use the software for a different purpose, you just pull out a few of the inappropriate modules and slot in others.

multiway selection format: formatting that puts an `if` directly after an `else` instead of indented on the next physical line. This makes it clear that the choice is from among three or more alternatives.

newline character: the character that causes the output to appear further down on the screen. The pair `\n` indicates a newline character within a string literal.

nested class: a class that is declared inside of another class X. If it is declared as `static`, the only difference from declaring it outside X is its visibility: (a) The nested class can access the private variables and methods of X; (b) No class outside of X can access the name of the nested class if that nested class is declared as private, even if its variables and methods are public. If the nested class is not private, outside classes refer to it as `X.ThatNestedClassName`. If the nested class is declared without the use of the word `static`, it can be called an inner class.

non-instantiable class: a class that does not have any instance methods or instance variables. It is therefore pointless to instantiate it (i.e., use `new`). Some people prefer the word to mean that the class has no public constructors. However, the Node class in Chapter Five has no public constructors and yet you may use `getNext()` to obtain an instance of that class to invoke instance methods of that class.

null: the value assigned to a variable to indicate that it does not reference any object at all. If you execute a command with that variable as the executor, you cause a NullPointerException.

object class: a class that has either instance methods or instance variables.

object-oriented design: determining the kinds of objects that a piece of software needs to get its job done, especially the capabilities (methods) that those objects must have. The key technique is to find objects that have the capabilities the software needs; if you know of none, then look for objects to which needed capabilities can be added (typically by subclassing existing classes); if that is still not enough, design suitable objects from scratch.

operator: a symbol that, when applied to one or more expressions according to the rules of Java, produces a value of a particular type. For instance, + is an operator that applies to two int values to produce an int value, and `!` is an operator that applies to one boolean value to produce a boolean value.

overloading a method name: having two methods of the same name in the same class. The two must have different numbers and/or types of parameters.

overriding a method definition: having an instance method in a class X with the same signature as an instance method in a superclass of X. Suppose the signature is `doStuff()` and `sam` refers to an object of class X. Then `sam.doStuff()` calls the subclass method and `sam.super.doStuff()` calls the superclass method.

package: a way of organizing classes. Each class in Java is categorized by the package it is in. If you put `package X;` at the top of a file, that puts the classes in that file into package X. If your file does not have this package directive, its classes are in the default package associated with that folder.

parameter: When you have a method call, the values in the parentheses after the method name are the parameters of the call. They supply additional information so the method can do its job.

partially-filled array: an array that has useable values only in part of the array. In this book, we normally put the useable value at components indexed `0...itsSize-1`, where `itsSize` is no more than the length of the array.

polymorphism: what occurs when a particular method call in a program could invoke any of two or more different method definitions at different execution times, depending on the class of the executor. The two or more method definitions must necessarily have the same signature.

postcondition: a statement of what will change as a result of calling a method, assuming the precondition for that method is met.

precondition: what has to be true when a method begins execution, in order that the method produce the expected result.

primitive type: a value that does not refer to an object. The four primitive integer types are long (8 bytes), int (4 bytes), short (2 bytes), and byte (1 byte). One byte of storage is 8 bits, so there are 256 different possible values for one byte. The two primitive decimal number types are double (8 bytes, 15 digits) and float (4 bytes, 7 digits). The other two primitive types are boolean and char.

private: A method declared as `private` can only be called from within the class where it is defined. By contrast, a method declared as `public` can be called from any class.

promotion: what occurs when you assign an int value to a double variable or combine an int value with a double value using an operator, e.g. +. The runtime system will promote the int value to a number with a decimal point. By contrast, assigning a double value to an int variable would be a "demotion", which requires that you put an `(int)` cast in front of it to convert it to an int value.

prompt: a string of characters printed to tell the user what kind of input is expected.

public: A method declared as `public` can be called from any class. By contrast, a method declared as `private` can only be called from within the class where it is defined.

query method: a method that returns a value but does not change the state of any object. By contrast, an action method changes the state of a variable but does not return a value. Try to avoid methods that do both.

queue: a list of values for which the primary operations are adding a new value and removing an old value; the latter operation always removes the value that has been on the list for the longest period of time. So it is often called a FIFO (first-in-first-out) data structure.

recursive call: a method call that occurs within the body of the method being called. If the call is made only when a particular parameter or attribute of the executor is a smaller positive integer than it was on the current call, then you cannot have an infinite sequence of recursive calls.

return type: tells what type of value (e.g., boolean, String, int) is returned by a call of a method. It goes just before the method name in the method heading. A method that returns no value must have void just before the method name in the heading.

robust program: a program that handles most unexpected or unusual situations in a reasonable manner without crashing.

scientific notation: a numeric form having a number from 1.0 up to but not including 10.0, followed by 'E' and then an int value indicating the power of 10 to multiply by. For instance, -2.75E6 represents -2750000 and 3.0E-6 represents 0.000003.

sentinel-controlled loop: a loop that repeats for each value in a sequence of values until you reach a special value, different from normal values, signaling the end of the sequence.

short-circuit evaluation: If the first operand `x` is true in `x || y`, or is false in `x && y`, then the second operand `y` is not evaluated, and the result is the value of `x`. This is short-circuiting.

signature of a method: the name of the method followed by parentheses containing the types of its parameters. Different methods can have the same name if they have a different parameter pattern (signature) or are in different classes. This is overloading of method names if they are in the same class.

software: the programs that you run and all the classes they use. By contrast, the physical components of the computer (chip, RAM, disk, monitor, etc.) are hardware.

source code: A file containing a human-readable compilable Java class, named `SomeClass.java`.

stack: a list of values for which the primary operations are adding a new value and removing an old value; the latter operation always removes the value that has been on the list for the shortest period of time. So it is often called a LIFO (last-in-first-out) data structure.

string literal: a sequence of characters enclosed in quotes, e.g., `"5 \t x"`.

structured natural language design: a design of an algorithm entirely in English or another natural language, except (a) steps that are to be done only if a certain condition holds, or for as long as a certain condition holds, are indented beyond the description of the condition, and (b) values computed in one step and used in another may be given a variable name.

stubbed documentation: a list of the method headings of a class with comments describing their functions, and not much else. It can be compilable if you have minimal bodies, e.g., `{return 0;}`.

subclass: a class defined with the phrase `extends SomeClass` in the heading; it is a subclass of the class named SomeClass. The subclass inherits each public method defined in SomeClass, i.e., each instance of the subclass can use those methods as if they were defined in the subclass.

subordinate: When one statement is part of another, the first is said to be subordinate to the second. A `while` statement contains a subordinate statement. An `if-else` statement contains two subordinate statements, one to be executed if the `if` condition is true and the other to be executed if the `if` condition is false. If you want several statements to be subordinate to a `while` or `if`, you must enclose them in braces.

superclass: A class from which another class X inherits public methods. `SomeClass` is a direct superclass of X if X has the phrase `extends SomeClass` in its heading.

terminal window: a window on the monitor, generally created by selecting MS-DOS Prompt, where you can enter commands to be executed by the operating system, including `java SomeClass`.

test plan: a large enough number of test sets that you can be confident that almost all of the bugs are out of the program.

thread of execution: An event-driven program has a thread of execution that responds to a user action when an event such as a button click or mouse movement occurs. If the user performs some other action (perhaps another button click) before that method finishes executing, the program will not respond to the second event -- the thread of execution that responds to events is busy. But you can have that event-handling thread spin off another thread of execution to respond to the first action. That lets the main event-handler go back to listening for events and therefore be able to respond to the second event.

trace of a program: a listing of the values that the most important variables have at various points in the execution of the program. It is used for studying and debugging the program.

Unicode: A specification that assigns an integer in the range 0 to 65535 to each character. It assigns consecutive integers to `'A'` through `'Z'`, also to `'a'` through `'z'`, and also to `'0'` through `'9'`. Characters with Unicode values below that of a blank are called whitespace. You may assign a char value to an int variable, in which case its Unicode value is used for the promotion.

utilities class: a class that does not have any instance methods or instance variables or main method. Therefore, it is pointless to create any object of that class.

visibility modifier: any of `protected`, `private`, and `public`. A `protected` variable or method is accessible only by classes in the same package (e.g., disk folder) and by subclasses. A variable or method with no visibility modifier has default visibility, which means that it is accessible only by classes in the same package.

wrapper class: a class that does not add significant functionality; it only changes the name or the form of the values stored.

whitespace: any one or more characters whose Unicode values are less than or equal to that of the blank character. Whitespace includes tabs, ends-of-lines, and ends-of-pages. But the official Java definition is only the characters with Unicode values of 9 through 13 and 28 through 32.

Appendix C
Common Compile-Time Errors

The following are most of the compiler error messages you will get with your programs and what the most likely cause of the error is. Also be alert for the following points with the terminal window:

(a) You can't use arrow keys or highlighting;
(b) You have to click the closer X on an applet or frame or use CTRL/C before you can get the C:> prompt and give more instructions in the DOS window;
(c) You have to use CTRL/C to kill a program that has fallen into an infinite loop;
(d) If you have several hundred error messages, you probably saved your program as a document (with all the formatting codes), not a plain text file. Use an appropriate "Save as" to remove all formatting.

You will save yourself some trouble if you verify the following facts about your program before you compile it. For the purposes of this description, a brace-pair is a left brace at the beginning of a line plus a right brace further down and aligned with it; all material between them should be indented further than the two braces:

(a) The unindented part of your program is 0 or more import directives ending in semicolons, followed by a line beginning "public class Whatever" with no semicolon at the end, followed by a brace-pair directly below the "p" in "public". All other material in the file is between those braces and indented further at least one level from the left margin.
(b) All lines that are indented one level begin with either "public" or "private". Each ends in a semicolon (field variables) or in parentheses (methods). Those that end in parentheses have a brace-pair directly below the "p" in "public" or "private". All other material in the program is indented at least two levels and is between some pair of braces for a method.
(c) No lines that are indented at least two levels begin with "public" or "private".

Can't convert boolean to java.lang.Boolean.
You misspelled boolean with a capital 'B'; it needs a small 'b'.

Can't make a static reference to nonstatic variable data in class X.
You declared the `data` variable outside of the `main` method (or outside of some other class method) instead of inside it.

Class Container not found.
You did not put `import java.awt.*` at the top of your file, or you misspelled the name of the class.

Class Turtle not found.
You declared a variable of type Turtle but you did not compile the Turtle class first. Or you compiled it but in a different folder from the one you are now in. Or your particular computer has other software on it that interferes with the compiler finding the Turtle class.

Class or interface declaration expected. [suppresses other errors]
You misspelled the word `import`, or you have an `import` after the class instead of before it.

'class' or 'interface' keyword expected. [suppresses other errors]
You have declared a method or variable outside of the class definition. Perhaps the preceding line ends in a right brace which, because you earlier accidentally left out a left brace, is taken as the end of the entire class when you meant it to be the end of the previous method.

Cannot resolve symbol. Symbol: data
The compiler cannot find a declaration of data. Perhaps you forgot to declare data in this class. Or you misspelled it. Or you are using it as an instance variable for one kind of object and it belongs to another kind of object.

Duplicate method declaration: void whatEver(int)
Maybe you meant to declare two methods with the same name and different parameter structures, but they do not differ in the parameters after all. Or you misspelled the method name, or you thought that just having a different return type would be okay (it isn't).

ExceptionInInitializerError in X.<init>
Your initialization of a class variable caused problems. An example is `private static int x = y / z` when z is a class variable whose value is zero.

Exception in thread "main" java.lang.NoClassDefFoundError X/java
You typed `java X.java`. You should type `javac X.java` to compile it. Or your program uses another class named X which you have not written with the name spelled that way.

Exception in thread "main" java.lang.NoSuchMethodError: main
You typed `java X` for an object class or a utilities class. You cannot "run" such classes. You should only try to run an application class, that is, a class that contains a `main` method.

Identifier expected. [suppresses other errors]
The method heading is `whatEver(x)` where you forgot to say what type `x` is. Or the preceding line ends in a right brace which, because you earlier left out a left brace, is taken as the end of the method instead of the end of a compound statement. Or you made an earlier error that confused the compiler so much that this is one of many error messages about things that are not really errors. Fix the earliest error first, then see if this one and others go away.

Incompatible type for =. Explicit cast required to convert double to int.
You tried to assign a decimal number to a whole-number variable. Use `(int)` to lose the fractional part.

Incompatible type for declaration. Explicit cast required to convert double to int.
You declared an int variable and initialized it to a double value, which is the wrong type.

Incompatible type for if. Can't convert int to boolean.
You had `if (x = 5)` but you need a double-equals == instead of an assignment symbol =.

Incompatible type for method. Can't convert anyOldThing to int.
The heading of the method you are calling has int for the parameter type, so you cannot put just anyOldThing in the parentheses of the method call and expect it to work; make sure it is an int.

Incompatible type for return. Can't convert java.lang.String to int.
The heading of the method you are in promises that it returns an int, but you have a String value following the word `return`. Either change the return type or change the value being returned.

Incompatible type for return. Explicit cast needed to convert X to Comparable.
You left out the phrase `implements Comparable` in the class heading. The heading of the method you are in promises it returns a Comparable value but you return an X value.

Invalid declaration.
The semicolon is missing at the end of the previous statement. Or you declared a variable directly after `else` or after `if(...)` (with no intervening left brace) or as the only statement in the body of a looping statement; this is illegal. You probably need to declare it before the compound statement, perhaps without assigning it a value.

Invalid expression statement. [suppresses other errors]
You called a method but forgot that every method call requires parentheses.

Invalid left hand side of assignment.
The part to the left of an assignment operator such as = or += must be a variable. Perhaps you wrote `if(x + 2 = y)` when you mean to test for equality instead.

Invalid type expression.
The semicolon is missing at the end of this statement.

'}' expected.
The semicolon is missing at the end of this statement. Most of the time, an "expected" message means you left something out and the compiler is only wildly guessing what that something was.

'(' expected.
Perhaps you forgot that you must have a left parenthesis immediately after every `while` or `if`, and of course a matching right parenthesis at the end of the `while` or `if` condition.

')' expected. [suppresses other errors]
Perhaps you declared a variable inside the parentheses for an `if` or `while` condition; a declaration (e.g., `int n`) should only be a stand-alone statement or inside the parentheses of a for-statement. Or maybe you left out a plus sign between Strings or left out some other operator.

';' expected. [suppresses other errors]
You used a reserved word (such as const or break or native) as the name of a variable or method. Or maybe you just forgot to put the semicolon at the end of a statement.

Method takeCd() not found in class X.
You forgot to put "extends Vic" in the class heading so it could inherit takeCD(). Or you misspelled the name of the method you are calling (check the capitals). Or you have the wrong parameter pattern (maybe takeCd has a parameter). Or the executor's class is not the class that contains the takeCd() method.

No constructor matching X() found in class X.
Okay, so you have a constructor in X, but not one with EMPTY parentheses.

No method matching whatEver() found in class X.
You forgot to declare the method named `whatEver`, or you misspelled the name (check the capitals), or you declared it in another class besides X, or the method heading calls for one or more parameters and you supplied none in the parentheses.

No method matching whatEver(int, int) found in class X.
You have the wrong number of parameters for the method call, compared with the method heading, or they are of the wrong type. The call has two int parameters, the heading does not.

public class X must be defined in a file called "X.java".
The class heading is `public class X` but the name of the file is not precisely `X.java`. Perhaps a letter is wrongly capitalized or you misspelled it, either in the class name or the file name or even the javac command. Or you might have saved it without using Text Document, so WordPad added ".txt" to the name. Type `dir x*` to see all files that begin with "X".

Return required at end of int whatEver(double).
The method heading for `whatEver` promises it will return an int value, so the last statement in the method must be either an unindented return statement or an if-else statement with a return statement for all alternatives.

'return' with value from void whatEver(double).
The method heading for the `whatEver` method you are in promises it will not return any value (that is what void means), but you went ahead and put a value after the word `return` anyway.

Statement expected.
You declared a variable as private or public inside a method. You only use private or public outside of a method. It also gives rise to later spurious errors such as "Identifier expected".

Statement not reached.
You have a statement after a return statement. But return means to exit the method immediately.

Undefined variable or class name: recieve.
You misspelled the variable name, or you forgot to declare it before you used it. Or it is the name of a library class and you forgot the import statement. Or you are using Vic but it is not in your current folder. Or it is the loop control variable declared within a for-statement and you are trying to use it after the end of the for-statement.

Unclosed String literal.
If you start some quoted material on a line, you have to finish it on the same line. So the number of quotes on each line must be even. You probably need to put a quote at the end of the line pointed to, then a plus sign and a new quote at the beginning of the next line.

Variable 'sue' is already defined in this method.
You have declared `sue` twice in the same method. If you have previously declared `sue` and now you want to assign it 7, write `sue=7` instead of `int sue=7`.

Wrong number of arguments in method.
You spelled the name of the method right but you have the wrong number or type of parameters.

```
public class LotsOfErrors    // does not compile correctly
{
   private int data = 12;
   public void first()
   {  int sue = 5;
      private int sam = 2;    /// Statement expected.
      sue = 12.3;              // Incompatible type for =.
      sue - 3 = 12;           /// Invalid left hand side of
assignment.
      methodSecond (false);    // Incompatible type for method.
      if (sue = 5)             // Incompatible type for if.
         int steve = 4;        // Invalid declaration.
      int sue = 7;             // Variable 'sue' is already
defined...
      sue = sam + 1           /// ';' expected.
      return 3;                // 'return' with value from
void...
      int don = 3;             // Statement not reached.
   }
   public static int methodSecond (int par)
   {  data = 20;              /// Can't make a static
reference...
      if (sue == 12)           // Undefined variable: sue.
         return true;          // Incompatible type for return.
      first;                  /// Invalid expression statement.
      Boolean x = false;       // Incompatible type for
declaration.
      String x = "go" "ne";   /// ')' expected
      if (int dru == 12)      /// ')' expected
         first(2);             // Wrong number of arguments...
      furst(3);                // No method matching furst()...
   }
   //public void main (args){} // Identifier expected.
}
```

APP-18

Appendix D Java Keywords

Java has forty-eight keywords. You cannot declare the name of a variable, method, or class to be one of these keywords. If you try to do so, the compiler may give a rather puzzling error message.

The keywords come in the several categories, listed below. Where the keywords are listed on two lines, the second line consists of keywords defined in the text in one place and never used anywhere else in the text.

8 primitive types of values:

```
boolean char int double long
byte short float
```

3 constant values (technically these are two boolean literals and one null literal instead of keywords):

```
true false null
```

14 statement indicators (the first word in the statement except for `else` and `catch`:

```
if else while do for return try catch throw
switch case default continue break
```

11 modifiers (can appear before the return type of a method):

```
public private static final abstract
protected synchronized native volatile transient strictfp
```

6 global (can be used outside of a class body):

```
class interface extends implements import
package
```

7 miscellaneous:

```
void new this super throws instanceof
finally
```

2 with no meaning, reserved so that people who accidentally put these C++ words in their programs will receive useful error messages:

```
const goto
```

Appendix E Sun Library Classes

The following are all of the Sun standard library packages, classes, and interfaces described in this book. We list them by package, and within each package we list one class per line, with indentations to show which are subclasses of which others in that package. Classes on the same level are listed alphabetically.

Interfaces have the word "interface" after them. After each class or interface we list the signatures and return types of all of their methods that are mentioned in this book other than those listed for the indicated superclass or interface. Class methods are indicated by "static" before the method name. The numbers in brackets are page references.

java.lang

Object [4-22,7-4,7-5,7-35] boolean equals(Object), String toString()
- Boolean [11-13] new(boolean), new(String), boolean booleanValue(), static Boolean TRUE, static Boolean FALSE
- Character [11-13,11-36] new(char), char charValue(), static boolean isDigit(char), static boolean isWhiteSpace(char), static boolean isLetter(char), static char toLowerCase(char), static boolean isLowerCase(char), static boolean isUpperCase(char)
- Comparable interface [6-10,6-11,6-12,11-5,11-12,13-1] int compareTo(Object)
- Math [6-21,6-40,11-35,11-36] static double abs(double), static double min(double,double), static double max(double,double), static int min(int,int), static int max(int,int), static long min(long,long), static long max(long,long), static double pow (double,double), static double cos(double), static double sin(double), static double E, static double log(double), static double exp(double), static double PI, static double random(), static double sqrt(double), static double rint(double), static double ceil(x)
- Number [11-12] static double doubleValue(), int intValue(), long longValue(), float floatValue(), byte byteValue(), short shortValue()
  - Byte [11-12] new(byte), new(String)
  - Double [6-2,6-8,6-18,6-40,11-11,11-12] new(double), new(String), static double parseDouble(String)
  - Float [11-12] new(float), new(String)
  - Integer [4-17,4-18,11-11,11-12] new(int), new(String), static int parseInt(String), static String toString(int,int), static int parseInt(String,int), static int MAX_VALUE, static int MIN_VALUE
  - Long [11-12] new(long), new(String), static long parseLong(String), static long MAX_VALUE, static long MIN_VALUE
  - Short [11-12] new(short), new(String)
- Runnable interface [11-30] run()
- String [3-4,5-10,5-32,6-10,6-11,6-12,6-14,6-15,6-40,7-35,7-36] int length(), boolean equals(String), String substring(int,int), String substring(int), char charAt(int), int compareTo(String), String concat(String), int indexOf(String), int indexOf(String,int), boolean endsWith(String), boolean startsWith(String), String trim(), String toLowerCase(), String toUpperCase(), int indexOf(char), int indexOf(char,int), String replace(char,char), char[] toCharArray(), new(char[]), boolean equalsIgnoreCase(String), int compareToIgnoreCase(String)
- StringBuffer [7-36,7-37] new(String), int length(), char charAt(int), setCharAt(int,char), String toString(), StringBuffer append(int,String), StringBuffer delete(int,int), StringBuffer insert(int,String), StringBuffer replace(int,int,String)

System [1-23,1-24,Int-3,4-1,5-3,5-4,5-36,6-20,6-36,7-34,7-35,10-11]
static long currentTimeMillis(), out.println(String), out.print(String),
static exit(int), static arraycopy (Object[],int,Object[],int,int)
Thread [10-11,10-12] new(Runnable), start(), static boolean interrupted(),
static sleep(int), interrupt(),
Throwable [10-9,10-33,10-34] new(String), new(), String getMessage(),
printStackTrace(), fillInStackTrace(), String toString()
Error [10-34]
AWTError [10-34]
LinkageError [10-34]
ThreadDeath [10-34]
VirtualMachineError [10-34]
Exception [6-8,10-3,10-4,10-9,10-12] new(), new(String)
InterruptedException [10-10,10-11] new(), new(String)
RuntimeException [6-8,10-3,10-12,10-13,10-35] new(), new(String)
ArithmeticException [6-8,10-3] new(), new(String)
ArrayStoreException [7-35] new(), new(String)
ClassCastException [6-12,6-31,6-39,10-3] new(), new(String)
IndexOutOfBoundsException [6-9,6-14,6-38,7-10,10-3] new(),
new(String)
ArrayIndexOutOfBoundsException [7-10] new(), new(String)
StringIndexOutOfBoundsException [7-10] new(), new(String)
NegativeArraySizeException [7-10,7-19,10-3] new(), new(String)
NumberFormatException [6-8,10-3] new(), new(String)
NullPointerException [6-8,6-21,6-39,7-14,10-3] new(), new(String)
IllegalStateException [14-6,15-19] new(), new(String)
IllegalArgumentException [15-23,15-24,15-25] new(), new(String)
UnsupportedOperationException [15-22,15-26,16-10] new(),
new(String)
SecurityException [12-31] new(), new(String)

javax.swing

JApplet (extends java.awt.Applet) [8-1,8-2,8-3,9-2,9-5,9-6]
BorderFactory [9-37] static Border createLineBorder(Color)
Box [9-35] createGlue()
BoxLayout (implements LayoutManager) [9-7,9-36] new(Container,int),
static int X_AXIS, static int Y_AXIS
ButtonGroup [9-29,9-30,9-41] new(), add(AbstractButton), remove(AbstractButton)
JComponent (extends java.awt.Container) [9-17,9-37] setBorder(Border),
setFont(Font), setIcon(ImageIcon), setToolTipText(String)
AbstractButton [9-13,9-31,9-40,9-41,11-8] addItemListener(ItemListener),
setText(String), String getText(), addActionListener(ActionListener),
setMnemonic(char), setActionCommand(String)
JButton [9-12] new(String)
JMenuItem [9-33] new(String)
JMenu [9-32,9-33] new(String), JMenuItem add(JMenuItem), add(String)
JToggleButton [9-29] boolean isSelected(), setSelected(boolean)
JCheckBox [9-29] new(String,boolean)
JRadioButton [9-29] new(String,boolean)
ImageIcon [9-37] new(String), paintIcon(Component,Graphics,int,int)
JComboBox [9-26,9-27,9-28,9-40] addActionListener(ActionListener),
new(Object[]), setSelectedIndex(int), int getSelectedIndex(),
setMaximumRowCount(int), Object getSelectedItem(),
addItem(Object), removeItemAt(int), setActionCommand(String)

JFileChooser [12-33] new(),
int showSaveDialog(null), static int APPROVE_OPTION,
static int CANCEL_OPTION, static int ERROR_OPTION,
int showOpenDialog(null), File getSelectedFile()
JFrame [9-1,9-4,9-5,9-32,9-39] new(String), String getTitle(), setTitle(String),
getContentPane(), addWindowListener(WindowListener),
JMenuBar getJMenuBar(), setJMenuBar(JMenuBar)
JLabel [9-8,9-40] new(String), setText(String), String getText()
JList [9-36] new(Object[]), setVisibleRowCount(int), int getSelectedIndex(),
Object getSelectedValue(), setSelectedIndex(int),
addListSelectionListener(EventListener)
JMenuBar [9-32] new(), new(String), JMenu add(JMenu)
JOptionPane [4-3,4-4,4-7,4-36,5-31,9-15] static showConfirmDialog(null,Object),
static String showInputDialog(Object), static int YES_OPTION,
static int showMessageDialog(null,Object), static int
YES_NO_OPTION, static int PLAIN_MESSAGE, static int
ERROR_MESSAGE, static int WARNING_MESSAGE, static int
QUESTION_MESSAGE, static int INFORMATION_MESSAGE
JPanel [9-5,9-39] new(), new(String)
JScrollPane [6-23,6-24,6-40] new(Component)
JSlider (implements SwingConstants) [9-19,9-20,9-21,9-40] new(int),
addChangeListener(ChangeListener), int getValue(),
setMinimum(int), setMaximum(int), setMinorTickSpacing(int)
JTextArea (extends JTextComponent) [6-23,6-24,6-40] new(int, int),
append(String)
JTextField (extends JTextComponent) [9-8] new(String), new(int),
setActionCommand(String), addActionListener(ActionListener)
JPasswordField [9-10] new(int), char[] getPassword()
Timer [9-23,9-24,9-41] new(int,ActionListener), start(), stop(),
setRepeats(boolean)
SwingConstants interface [9-20] static int HORIZONTAL, static int VERTICAL,
static int NORTH, static int SOUTH, static int EAST, static int WEST,
static int CENTER, static int BOTTOM, static int TOP

javax.swing.event

ChangeEvent (extends EventObject) [9-42]
ChangeListener interface [9-20,9-42] stateChanged(ChangeEvent)
ListSelectionEvent [9-36,9-37]
ListSelectionListener interface [9-36,9-37] valueChanged(ListSelectionEvent)

javax.swing.border

Border interface [9-37]

javax.swing.text

JTextComponent (extends JComponent) [9-40] setText(String), String getText()

java.applet

Applet (subclass of Panel -> Container -> Component) [9-36] init(), start(), stop(),
URL getDocumentBase(), AudioClip getAudioClip(URL,String),
AudioClip interface [9-37] play(), loop(), stop()

java.awt

BorderLayout (implements LayoutManager, SwingConstants) [9-5,9-35] new(int,int)
CardLayout (implements LayoutManager) [9-36] new(), first(Container),
last(Container), next(Container), previous(Container)
Color [8-1,8-2,8-4] new(int,int,int), static Color black, static Color cyan,
static Color darkGray, static Color lightGray, static Color magenta,
static Color green, static Color yellow, static Color red,
static Color blue, static Color white, static Color orange,
static Color gray, static Color pink
Component [8-6,8-3,8-32,9-2,9-7,9-38] paint(Graphics), int getWidth(),
int getHeight(), setSize(int,int), Graphics getGraphics(),
setVisible(boolean), setBounds(int,int,int,int),
addMouseListener(MouseListener), repaint()
Container [9-5,9-35,9-39] setLayout(LayoutManager), add(Component,int),
add(Component), LayoutManager getLayout(),
Window [9-4] addWindowListener(WindowListener)
FlowLayout (implements LayoutManager) [9-5,9-7] new()
Font [8-30] new(String,int,int), static int PLAIN, static int BOLD, static int ITALIC
Graphics [8-2,8-4,8-31] Color getColor(), setColor(Color), drawString(String,int,int),
setFont(Font)
Graphics2D [8-2,8-4,8-31] draw(Shape), fill(Shape)
GridLayout (implements LayoutManager) [9-36] new(int,int), new(int,int,int,int)
LayoutManager interface [9-5,9-35] layoutContainer(Container)
Polygon (implements Shape) [8-30] new(int[], int[], int), new(), addPoint(int,int),
translate(int,int,int)
Rectangle (implements Shape) [8-5,8-31] new(int,int,int,int), grow(int,int),
translate(int,int), setLocation(int,int), setSize(int,int)
Shape interface [8-31] boolean contains(Point2D)

java.awt.geom

Ellipse2D (implements Shape) [8-6]
Ellipse2D.Double [8-6,8-32] new(double,double,double,double)
Line2D (implements Shape) [8-2,8-4,8-31] ptLineDist(Point2D)
Line2D.Double [8-2,8-4,8-6,8-32] new(double,double,double,double),
new(Point2D,Point2D)
Point2D (implements Shape) [8-31]
Point2D.Double [8-31] new(double,double), double getX(), double getY(),
setLocation(double,double)
Rectangle2D (implements Shape) [8-6]
Rectangle2D.Double [8-6,8-32] new(double,double,double,double)
RoundRectangle2D (implements Shape) [8-6]
RoundRectangle2D.Double [8-6,8-32]
new(double,double,double,double,double,double)

java.awt.event

ActionEvent (extends EventObject) [9-10,9-42] String getActionCommand()
ActionListener interface [9-10,9-14] actionPerformed(ActionEvent)
ItemEvent (extends EventObject) [9-31,9-42] int getStateChange(),
static int SELECTED, static int DESELECTED
ItemListener interface [9-31] itemStateChanged(ItemEvent)
MouseAdapter (implements MouseListener) [9-34]
MouseEvent (extends EventObject) [9-34,9-42] int getX(), int getY()
MouseListener interface [9-34] mouseClicked(MouseEvent),
mouseEntered(MouseEvent), mouseExited(MouseEvent),
mousePressed(MouseEvent), mouseReleased(MouseEvent),
WindowAdapter (implements WindowListener) [9-34,9-42]
WindowEvent (extends EventObject) [9-3,9-4,9-42]
WindowListener interface [9-3] windowClosing(WindowEvent) and 7 others

java.util

AbstractCollection (implements Collection) [15-35] abstract size(), abstract iterator()
AbstractList (implements List) [15-35] abstract size(), abstract get(int)
ArrayList [7-32,7-33,15-35] new(), new(Collection), int size(), Object get(int)
Vector [14-35] // a deprecated class
Stack [14-35] push(Object), Object pop(), Object peek(),
boolean empty(), int search(Object)
AbstractMap (implements Map) [16-33] abstract Set entrySet()
Arrays [13-31] static boolean equals(Object[],Object[]), static sort(Object[],int,int),
static boolean equals (anyPrimitiveType[],sameType[]),
static sort(anyPrimitiveType[]), static binarySearch(Object[],Object),
static sort(anyPrimitiveType[],int,int), static fill(Object[],Object),
static fill(anyPrimitiveType[],sameType), static sort(Object[]),
Collection interface [15-4,15-23] boolean add(Object), boolean add(Collection),
boolean isEmpty(), int size(), Iterator iterator(), clear(),
boolean contains(Object), boolean containsAll(Collection),
boolean retainAll(Collection), boolean equals(Object),
boolean remove(Object), boolean removeAll(Collection),
Object[] toArray(), Object[] toArray(Object[])
List interface (extends Collection) [15-34,15-37] Object get(int), remove(int),
set(int,Object), add(int,Object), addAll(int,Collection),
ListIterator listIterator(), ListIterator listIterator(int),
int indexOf(Object), int lastIndexOf(Object), List subList(int,int)
Set interface (extends Collection) [15-23] // unique elements, no more methods
Comparator interface [13-31] int compare(Object,Object),
boolean equals(Object,Object)
EventListener [9-20,9-41] // no methods, so this is a "tagging" interface
EventObject [9-13] Object getSource()
HashMap (implements Map) [16-27,16-33] new(), new(Map)
Iterator interface [15-16,15-17,16-9,16-23] boolean hasNext(), Object next(),
remove()
ListIterator interface [15-26,15-32,15-37] add(Object), set(Object),
int nextIndex(), boolean hasPrevious(), Object previous(),
int previousIndex()
Map interface [14-35,16-8,16-16,16-32] put(Object,Object), Object get(Object),
int size(), boolean containsKey(Object), Object remove(Object),
boolean isEmpty(), clear(), boolean containsValue(Object),
putAll(Map), Set entrySet(), Set keySet(), Collection values()
Map.Entry interface [16-30,16-33] getKey(), getValue(), setValue(), equals(Object)
NoSuchElementException (extends RuntimeException) [15-19,16-10] new(),
new(String)
Random [4-16,4-18,4-36,6-36,6-37] new(), int nextInt(), int nextInt(int),
double nextDouble(), long nextLong(), float nextFloat(),
double nextGaussian()
StringTokenizer [12-9,12-10,12-34] new(string), boolean hasMoreTokens(),
int countTokens(), String nextToken()
TreeMap (implements Map) [16-33] new(), new(Map)

java.io

File [12-33] boolean exists(), delete(), boolean canRead(), boolean canWrite()
IOException [10-7,10-9,10-12,12-5,12-7,12-31,15-2] new(), new(String)
RandomAccessFile [12-22,12-34] new(String,String), writeInt(int), int readInt(), writeChars(String), String readLine(), writeDouble(double), double readDouble(), writeChar(char), char readChar(), writeBoolean(boolean), boolean readBoolean(), close(), int length(), seek(long)
Reader [12-25] abstract read(char[],int,int), abstract close()
BufferedReader [10-7,12-2,12-3,12-4,12-33] new(FileReader), new(InputStreamReader), String readLine()
InputStreamReader [10-7,12-3,12-4] new(InputStream)
FileReader [10-7,12-2,12-3] new(String)
StreamTokenizer [12-11] new(Reader), int nextToken()
Writer [12-6,12-7,12-23] abstract write(char[],int,int), abstract flush(), abstract close()
FileWriter [12-6,12-8,12-33] new(String)
PrintWriter [12-6,12-7,12-8,12-33] new(Writer), flush(), close(), println(String), print(String)

java.io.print

PrinterAbortException [10-10] new(), new(String)

java.text

NumberFormat [6-37] static setMaximumFractionDigits(int), format(double), static setMinimumFractionDigits(int), static NumberFormat getNumberInstance()
DecimalFormat [6-38] new(String)

java.net

ServerSocket [12-31,12-32] new(int,int), accept(), close()
Socket [12-31,12-32] new(String,int), InputStream getInputStream(), OutputStream getOutputStream(), close()

Appendix F Major Programming Projects

Problem number C.N indicates chapter C and problem N for that chapter. Project descriptions marked with & are purposely ambiguous and incomplete. For them, you are to write out an analysis of the situation in the form of several questions addressed to the client to clear up ambiguities, plus answers that the client might plausibly respond with. You should write out the top-level logic design and object design before attempting implementation or lower-level design. Your instructor may ask you to submit the analysis and design for evaluation before you go on to the implementation stage.

3.1 **ShiftToEnd:** Write an application program using Vics that clears all the slots to the stack for the first sequence of slots, then puts those CDs in its last few slots. For instance, if there are 5 slots with 2 CDs in them somewhere, you should end with those CDs only in the fourth and fifth slots. Do not assume that the stack is initially empty. Extra Credit: Repeat for all sequences of slots.

3.2 **DoubleUp:** Write an application program that first clears the slots of the second sequence to the stack, then finishes with twice as many CDs in the first sequence's slots as it had to start with (or fewer if the first sequence becomes filled). Precondition: The second sequence is known to have more CDs in its slots than the first sequence has.

3.3 **FillTheFirst:** Write an application program that puts all the CDs from the second, third and fourth sequences of slots into the first sequence's empty slots. But if there are too many CDs for that, put the excess in the second sequence's empty slots (and if there are still some left over, fill in the third sequence's empty slots to the extent possible).

3.4 **MoveBackByOne:** Write a class with an instance method that moves back by one slot every CD for which, at the time execution of the method begins, (a) the CD is beyond the current slot, and (b) the CD is directly preceded by an empty slot. Test it with an appropriate main method.

3.5 **ShiftOver:** Write a class with an instance method that has a Vic parameter: Each empty slot of the parameter's sequence is to be filled by the next available CD from the executor's sequence, until the executor's sequence runs out of CDs or the parameter's sequence runs out of empty slots. Test it with an appropriate main method.

3.6 **FillSlots:** Write a class with an instance method that moves all CDs in a sequence's slots to the front of the sequence, without having more than one of its CDs on the stack at any given time. Test it with an appropriate main method.

3.7 **AddBinaries:** Write an application program that "adds" the second sequence to the first sequence, treating each sequence as if it is a binary number written in reverse order of digits, with a CD considered a 1 and the absence of a CD considered a 0. Precondition: The first sequence is longer than the first and its last slot is empty (this avoids number overflow).

4.1 **StudentGrades**: Write a complete Student class that allows you to use the statements such as the following in methods in outside classes, with the obvious meanings. Then write an application program that creates one Student object, reads in grades until a negative "grade" is seen (as a signal that the list of grades has ended), then prints out all available information about the Student:

```
Student bob = new Student ("Robert", "Newhart");
bob.storeGrade (23);          // bob scored 23 on this test
return bob.getTotalGrades();  // total of test scores to date
return bob.getAverageGrade(); // for all test scores to date
return bob.getName();         // returns "Robert Newhart"
```

4.2 **Counter:** Write a complete Counter class that allows you to use the statements such as the following in methods in outside classes, with the obvious meanings. Then write an application program that creates one Counter object, reads in a sequence of numbers until the user indicates it is time to quit, then prints out all available information about the Counter:

```
Counter sam = new Counter();
sam.recordNumber(x);       // sam adds x to its values counted
return sam.getPositives(); // number of positive numbers
                           // recorded so far
return sam.getNegatives(); // number of negative numbers
                           // recorded so far
return sam.getZeros();     // number of zeros recorded so far
```

4.3 **Dates:** Write a complete Date class with a constructor new Date(month, day, year) that creates a Date object with the given month, day and year. Use a hypothetical calendar in which every month has 30 days. A day value outside the range 1 through 30 is to be replaced by 1. For a negative month number, add N*12 to it and subtract N from the year, to make the month number 1 through 12. Make the opposite adjustment for month numbers greater than 12. Write an add method, a subtract method, a toString method, and a compareTo method. Also have two more constructors, one to construct a Date with the same values as a given Date, and another to construct a Date with only the year supplied (make it January 1). Extra credit: Use a real calendar for the number of days in each month, except ignore the possibility of leap years.

4.4 **DiceGame:** Write a subclass of BasicGame for which one game is the roll of two dice, reporting the two values and the total. The player wins if the roll is a 7 or 11, loses if it is 12 or 2, and ties in all other cases. Extra credit: There are no ties. If the player's roll total of X is other than 7, 11, 2, or 12, the player rolls again until a total of either 7 or X comes up. The player wins only if X comes up before 7.

4.5 **MakeThreeEqual:** Write a subclass of Vic with instance methods that (a) tell how many CDs the executor has and (b) put a given number of CDs in its first few slots (the given number is a parameter). Then write an application program for Vics that finds out how many CDs each of the first three sequences of slots has. Find the smallest number N of those three numbers. For each of the sequences that has more than N CDs in its slots, clear out all of its slots to its stack and then put back N CDs in its first few slots. For purposes of this program, N is zero if the machine has only one or two sequences of slots.

4.6 **GuessMyNumber:** Revise the GuessNumber class so that the computer lies about having chosen a number. Instead, it tells the user "too high" or "too low" depending on which will make the user take the greatest number of guesses to get it right. It only says the user is right when there is only one possible answer consistent with its previous answers and the user has guessed it.

4.7 **Mastermind4:** Revise the Mastermind game to have the user guess four digits, not three.

4.8 **GuessThatNumber:** Write a subclass of BasicGame in which the computer asks the user to choose a whole number from 1 to 100, inclusive. The program then makes a number of guesses. After each guess, the user tells whether the guess is too high, too low, or exactly right. Once the computer "knows" what the correct number is, the computer tells the user the correct number and the number of guesses tried. Have the program ask the user after each game whether she wants to play again. Design the program so that it always guesses the correct answer by the seventh try at latest. Hint: Have two instance variables that keep track of the smallest and largest values that the guess could be (initially 1 and 100). Guess halfway between them. After each wrong guess, update these smallest and largest values appropriately.

4.9 **RandomWalk:** Position five Turtles on the drawing area 100 pixels apart. Have each one draw a circle 25 pixels in radius. Then have each Turtle repeat the following 1000 times, painting as it goes: Choose one of the directions North, South, East, or West at random and move one pixel in that direction. This is called a random walk, so you should have five random-walk paths. Record the number of Turtles that ended up outside their circles. Repeat at least 20 times until you have a good estimate for the probability that a 1000-step random walk ends up more than 25 steps away from its starting point.

4.10 **CoinMatch:** Create a Die class that keeps track of the number (1 through 6) that was on top of the Die when last rolled (initialize it with 1). However, the constructor should allow any whole number of sides to the Die greater than 1. For instance, if constructed with two sides, it would in effect be a coin. Provide methods to roll the Die and to retrieve what was last on top.

Write a program that uses the Die class to create two coins, one for the user and one for the computer opponent. Flip the coins, recording a win for the user if they match on heads, a win for the computer if they match on tails, and no-win if they do not match. Report on the status after each flip. The winner of the entire game is the one who gets 10 wins first.

4.11 **DoorLock &:** A dedicated computer manages all the door locks in a building. Each door lock is a number pad beside the door. A user must enter the correct sequence of digits and then turn the handle to open the door. An alarm goes off if the user enters the wrong combination too many times in a row.

Write a complete analysis and design for this problem. Then completely implement a DoorLock class of objects that would appropriately represent a single door lock.

5.1 **MilitaryTime:** Write a class containing only class methods to do computations involving military and civilian time. Consider that 0900 is "9:00 AM" and 1420 is "2:20 PM". The military times are int values ranging from 0000 up to 2359. The class should include methods such as `toCivilian(int)` returning a String, `toMilitary(String)` returning an int, and `isMilitary(int)` returning a boolean. Also, `getHoursLater(int,int)` should tell how many whole hours the second parameter is later than the first, and `getMinsLater(int,int)` should return an answer 0 to 59 telling how many minutes the second parameter is later than the first, ignoring whole hours. You should also have the same two methods overloaded with two String parameters for civilian time. That way, you can print the following:

```
t1 + " is " + getHoursLater (t1, t2)  " hours and "
      + getMinsLater (t1, t2) " minutes after " + t2
```

5.2 **MathOp:** Add the following class methods to the MathOp class (Listing 5.1), then write a simple application program to test them all:

1) abs returns the absolute value of an int parameter.
2) sqrt returns the largest int value whose square is not greater than the int parameter. Return -1 if the parameter is negative.
3) max is overloaded: three methods that return the largest of two, three, or four int parameters.
4) min is overloaded: three methods that return the smallest of two, three, or four int parameters.
5) sum returns the sum of all int values that are equal to or between two int parameters.
6) gcd returns the greatest common divisor of two int parameters (as described in an exercise).
7) log2 returns the highest int value such that 2 to that power is not more than the int parameter.

5.3 **TwoCharacterCDs:** Revise the given Vic implementation so that the name of each CD is two characters, not one. Use "00" for the absence of a CD. For instance, `" a100itis00"` is the way you would record a five-slot sequence consisting of "a1" in the first slot, "it" in the third slot, "is" in the fourth slot, and no CD in slots two and five. A String received as input in `reset` is to have the digit for its sequence appended to each character representing a CD, e.g., if the second value `args[1]` is `"a0bc"`, you are to represent the sequence as `" a200b2c2"`.

5.4 **StringBufferVics:** Read the description of the StringBuffer class in Section 7.12. Then revise the given Vic implementation to store the stack and all sequences as StringBuffer objects, but with no change in effect.

5.5 **TicTacToe:** Write TicTacToe as a subclass of BasicGame. Have the computer pick the first available open spot each time it is its turn. Display the board as a String value with two of `'\n'` in it, to get three lines displayed. Extra credit: Be sure that the computer never loses.

5.6 **Hangman &:** Write Hangman as a subclass of BasicGame. Each word to be guessed should have five letters. Have a 250-letter String class variable that stores 50 possible words to choose from at random. Allow 9 wrong guesses before the player loses.

5.7 **GPA:** Write an application program that accepts from the user a number of String inputs representing grades in various college courses. When the user responds to showInputDialog with a value that does not have exactly either 1 or 2 characters, print the grade-point average and terminate the program. The single letters A, B, C, D, F have the values 4, 3, 2, 1, and 0. If they have a "+" or "-" after them, add or subtract 0.3 respectively. Note: You do not need to know how to calculate with numbers with decimal points in Java (type double) to do this program, but it would make the program easier to do.

5.8 **SmarterNet:** Write a subclass of the SmartNet class (Listing 5.8) with these instance methods:

1) averagePerNode returns the average number of edges per node.
2) numberOfEdges returns the total number of edges (do not forget to divide by 2).
3) averageForConnections returns the average number of edges per node connected to a given node parameter.
4) hasTriangle (Extra credit) tells whether there is any node which connects to a different node which connects to a third node (different from both of those) which connects back to the first node.

5.9 **Tums:** Implement the Tum class described in Section 3.9 similar to the String implementation of the Vic class. The following is the constructor done for you. Extra credit: Modify the constructor to replace any non-digit character by a blank.

```
public Tum (String given)
{  super();
   if (given == null)
      itsSequence = " ";
   else
      itsSequence = " " + given;
   itsPos = 1;
}
```

6.1 **BorrowMoney:** Write a program that repeatedly accepts an amount of money from the user, stopping when the user says to. After each positive input, which represents an amount borrowed, ask the user for the annual interest rate (e.g., 7.2 represents 7.2% annual rate) and the number of months over which the loan is to be paid off. Calculate and print each monthly payment, except print an error message if either input is non-positive.

Hint: First convert the interest rate to a monthly rate R (e.g., 7.2% corresponds to R = 0.006). The "infinity" monthly payment, what you would have to pay if the loan ran on forever, is R times the amount borrowed. You obviously have to pay more than that monthly to pay off the loan. The actual amount is calculated by dividing the "infinity" monthly payment by 1-(1/F), where F is the result of taking 1+R to the power equal to the number of months (use a while-loop to get the power).

6.2 **UlamValues:** Write a program that asks the user for a positive integer and then calculates the Ulam value for each of the 100 integers from that point on up (for instance, if the input is 3200, calculate the Ulam value for each number 3200 up to but not including 3300). Report the largest Ulam value seen and the number that has that Ulam value. The Ulam value for a number N is defined as the number of iterations of the following process required to get to the number 1, starting with N: Divide N in half if N is even, otherwise multiply N by 3 and then add 1.

6.3 **Calendar &:** Write a program that reads in a date as three integers Month/Day/Year. It then tells the user what day of the week that was. Have it work correctly for every date from 1/1/1753 on. It is useful to know that January 1, 1753 was a Monday, and that every year divisible by 4 except 1800, 1900, 2100, etc. is a leap year (i.e., not centesimal years except those divisible by 400). The reason for starting with 1753 is that we switched to the Gregorian calendar in 1752, losing 11 days. Extra credit: Print out a calendar for the entire month of the date requested.

6.4 **RepairQueue:** Revise the RepairShop class (Listing 6.9) to use appropriately a RepairQueue class of objects that extends the Queue class. A RepairQueue has two instance variables, itsTotalTime and itsCount (remove totalTime from the CarRepair class). It has three instance methods: remove() returns a RepairOrder object and updates its instance variables, add(RepairOrder) adds one RepairOrder and updates its instance variables, and jobDescription() returns a two-line string giving the job most recently returned, the total hours for the remaining jobs, and number of jobs remaining on the queue.

6.5 **BankAccount:** Write a BankAccount class with these instance variables: itsCurrentBalance, itsOwner (a String), itsNumDepositsMadeThisMonth, and itsMinimumBalanceForThisMonth. Have methods that get these values and methods that change the owner's name. Also have methods that accept an amount for deposit or withdrawal (which must be positive numbers, otherwise the method call has no effect). Any check that brings the balance below zero is to be bounced with a $25 penalty. Add a `printMonthlyStatement()` method that prints the balance as of the end of the current month, together with fees, interest, and the values of all instance variables. The interest is 3% annually compounded monthly, paid on itsMinimumBalanceForThisMonth. The fees are (a) 15 cents per check that was cashed when the balance was below $1000, (b) 10 cents per deposit, and (c) $3 per month. However, waive all fees if the balance was never below $3000 at any point during the month.

Write an application program that creates three BankAccount objects number 1, 2, 3. It makes #1 the current account, then accepts user requests to credit or debit the current account, print a monthly statement for it, change the owner's name, or switch over to an account with a different number.

6.6 **NumbersWithCommas:** Add two independent class methods to your own library classes: One accepts a long value and returns the String equivalent with commas every three digits. The other accepts a String value that should represent an integer with commas every three digits and returns the long value corresponding to it. Make sure you do not put in any unnecessary commas or leading zeros. For extra credit

6.7 **JustifiedOutput:** Add six independent class methods to your own library classes: `rightJustify (x,10,4)` returns the string representation of the decimal value in x, with exactly four digits after the decimal point and a total of ten characters altogether (including the decimal point and leading blanks). Similarly, `rightJustify (n,10)` returns the string representation of the integer value in x, with enough leading blanks added to make a total of ten characters altogether. Add two more for `leftJustify` and two more for `centered`. Note: This meaning and order of the parameters is almost the same as what is used in the output statements that are built-in with Pascal and Fortran.

6.8 **Quadratics:** Develop a Quadratic object class whose instances represent quadratic equations. A Quadratic object should be able to return its value for a given x-value, tell whether it has any zeros, return one of those zeros (which one depends on a boolean parameter), modify itself so that its graph translates horizontally or vertically (have two parameters to specify the translation), and say where its inflection point is. Have three instance variables for its three coefficients plus one for its discriminant (b*b-4*a*c). Have two constructors, one with three coefficients as parameters and one with a String parameter that is supposed to contain three numerals. Write a main method that tests out all of the Quadratic methods.

6.9 **Shorthand:** Write a program that repeatedly accepts a line of input and produces the shorthand form of that line, defined as follows: (a) remove all vowels ('a', 'e', 'i', 'o', 'u', whether lowercase or uppercase), except (b) replace these four words: "and" by "&", "to" by "2", "you" by "U", and "for" by "4".

6.10 **LunarLander:** Design a Lander class of objects with (a) a 3-parameter constructor: its initial height in feet above the lunar surface, its initial rate of fall in feet/sec, and its gravitational pull (which for the moon is about 5 feet/sec change in velocity for each second of fall, but allow for the possibility that this program will be used for various planets too); (b) an action method with double parameter x that updates the current height and rate of fall assuming x is the number of G's of thrust applied over a 1-second period since the last update (1G is a 32 feet/sec decrease in the rate of fall relative to what it would otherwise be due to gravity alone); (c) two query methods that return the current height and rate of fall.

Next, write a main method that creates a Lander with semi-random initial values and repeatedly gets thrust requests from the user (pilot) once for each 1-second period, making the appropriate changes and reporting the status, until the Lander hits the ground. Report the speed at which it hit. Allow many re-plays with the same initial values until the user wants to change to different initial values. Note: After you learn to use graphics in Chapter Eight, add a graphical display of the current position of the lander.

Extra credit: Write a different main program that finds the best solution by repeated simulation, as follows: It applies no thrust for T seconds, then applies exactly 1G of thrust until the the Lander hits the ground or its velocity is reduced to zero. Have it try integer values of T from 1 on up until it finds the first value that has the Lander hit the ground, then repeatedly subtract 0.1 from T until it finds the first value that reduces the velocity to zero just before it hits. Print this best solution for any given inputs.

7.1 **HighLowPay:** Write a program that reads in a list of Worker data from a file. It then prints out the name and pay of the worker who has the highest pay, the second highest, the lowest, and the second lowest. In the case of ties for pay, it prints the one whose name comes alphabetically earlier. The program also prints the average pay per worker.

7.2 **FederalTax &:** Write a program that reads in a list of Worker data from a file into a WorkerList. Then it prints a report giving the name, gross pay, federal income tax, and net pay for each Worker. Calculate the federal income tax using the following approximation of the tax rate on singles: Multiply the number of exemptions by $2,800 and add $4,400 for the standard deduction. This amount is tax free. The next $27,000 that the worker earns is taxed at 15%, and all the rest is taxed at 28%.

Add an instance variable to each Worker object to record the number of exemptions he/she is entitled to. Also, the dollar amounts above are per year, so divide them by 52.2, the number of weeks in a year. Extra credit: Research and use the correct dollar amounts for the current calendar year. Also allow for married tax rates.

7.3 **Mastermind:** Revise the Mastermind methods in Listing 4.9 and 4.10 to allow the user to choose a game with two to seven digits. Have the constructor get the number 2 to 7. Then create an array for the secret digits and an array for the user's digits. Do not make any assumptions about the input being digits.

7.4 **Mancala:** Search the Internet to find the rules for the game of Mancala. Write the program as a subclass of BasicGame.

7.5 **Cypher &:** Create a Code class of objects and a program that maintains an array of several Code objects. The program accepts as input several lines, each containing a single integer followed by a string of letters and blanks. The first character of the String is either E (for encode) or D (for decode). For an E, the program is to output the encoded form of the rest of the String. For a D, the program is to output the decoded form of the rest of the String. Each Code object represents a simple substitution cypher, in which each of the 26 letters is replaced by the corresponding character specified by the code. Each Code object has an ID number; the integer input tells the ID number of the Code which is to be used.

Example: If Code #1 is to add 2 to the Unicode value of each character, then "1E the quick brown fox" causes an output of "vjg swkem dtqyp hqz", and "1D uewbbz ftkxg" causes an output of "scuzzy drive" (x, y, and z encode into z, a, and b respectively -- you subtract 26 if adding the number takes it past z or Z). The object representing Code #1 should have a String instance variable with the value "cdefghijklmnopqrstuvwxyzab", giving the encoded form of the alphabet.

7.6 **FormLetter &:** Write a program that reads in a plain text file containing what should be a form letter. Any use of `\Name` in the letter should be replaced by a user-supplied value, and similarly for `\Address` and `\Date`. If the form letter is supposed to contain the backslash character itself, it uses `\\` to indicate one instance of the backslash to be printed. The form letter with the substitutions should be printed.
Extra credit: First read another file that supplies a list of substitutions to be made. For instance, if the second file contained the following, the program would produce two corresponding form letters:

```
\Name       Ralph Kramden
\Address    5 Main Street
\Date       January 5, 2003
\Name       Walter Mitty
\Address    27 Milquetoast Avenue
\Date       January 7, 2003
```

7.7 **Networks:** Write the Node and Position classes to go with the Network class described in Section 7.8. Then replace the Network constructor to read data from a file named "networks.txt" in a reasonable form, with each Node's name and connections given on one line. Always use the same file name; the user can replace that file to use different network data.

Extra credit: Add an Edge class with an instance variable of Node type, which stores the Node that it connects to. The two-dimensional array should be an array of Edges, not Nodes. An Edge should have three instance methods: getNode, getMark, and setMark (the latter two analogous to those for Nodes). The getNext method for Positions should contain one statement: return itsList[itsPos].getNode(). A Position should have an additional method getEdge that returns itsList[itsPos].

7.8 **NongraphicalVics**: Complete the development of a non-graphical simulator for Vics begun in Section 7.8. So that a user can see what the status is at any point in a program, have the constructor and the putCD and takeCD methods print to System.out a description of the complete current status of the executor, including the status of the stack.

7.9 **Sets**: Implement a class for which each instance models a mathematical set of objects. This class has some similarities to the Queue class. The Set class should have the following instance methods:

```
public boolean add (Object ob); // add ob only if no Object that
    // equals it is already in the Set; tell whether the Set changed.
public boolean remove (Object ob); // remove the Object that equals
    // ob, but only if it is there; tell whether the Set changed.
public boolean contains (Object ob);   // tell whether some Object in
    // the Set equals ob
public Set union (Set par);   // return the union of the two sets
public Set intersection (Set par);   // return their intersection
public Set difference (Set par);   // return the Set of all Objects
    // that are in the executor but not in par
```

Also implement size(), isEmpty(), equals(), and toString() with the normal meanings, plus a constructor to make an empty Set and a constructor to make a copy of a Set parameter.

7.10 **WeatherChannel:** The Weather Channel has hired your software development group to create software that accepts from the keyboard a sequence of temperature readings. These readings are taken at noon, 6pm, midnight, and 6am each day. You may have thousands of temperature readings, but you will not have more than 200 different temperature values. The software must, at any time during data entry when requested, (a) display the different temperatures in ascending order, with the number of occurrences of each, (b) tell the largest temperature seen so far, (c) tell whether a specified temperature has occurred before, or (d) tell the temperature value that has occurred most often (in case of ties, tell the largest such value).

Write this program. Use a DataSet class with two instance variables: private int itsSize; private Info[ ] itsItem. Have a separate DataSet instance method to perform each of the operations described in (a)-(d). Use a nested class of objects, defined inside the DataSet class as follows:

```
private static class Info
{  public int itsTemp;
   public int itsCount = 1;
   public Info (int par)
   {  itsTemp = par;
   }
}
```

Restriction: Design the DataSet class as a partially-filled array so that, if the following method were part of that DataSet class, it would remove all temperature values less than the given temperature value:

```
public boolean erase (int cutoff)
{  int keep = 0;
   while (keep < itsSize && itsItem[keep].itsTemp < cutoff)
      keep++;
   if (keep > 0)
      for (int k = 0;  k < itsSize - keep;  k++)
         itsItem[k] = itsItem[k + keep];
   itsSize -= keep;
}
```

Extra credit: Revise the main logic so that the user can handle ten DataSet objects instead of just one. The user can specify a number 1..10 to indicate which DataSet is being handled by all future requests to add, display, etc., until the user specifies an new number 1..10. Use an array of 10 DataSets.

7.11 **FlashCards:** Write a program that reads in a text file with two words on each line, the first in English and the second in Spanish (or whatever other language you prefer). This program allows the user to practice vocabulary items in a foreign language. As the pairs of words are read in, store each pair in a Vocabulary object, which you then put in a WordList object, a partially-filled array of Vocabulary items that you always keep in alphabetical order of English words, regardless of their order in the file. However, whenever one pair has the same English word as an earlier pair, replace the earlier pair with the later one. Restriction: Vocabulary objects should be mentioned only within the WordList class.

After reading to the end of the file, the main method lets the user repeatedly indicate one of the following choices until EXIT is chosen (you may use either letters or digits to indicate the choices):
(a) EXIT means quit the program.
(b) TO_ENGLISH means list the English words one at a time, waiting until the user presses ENTER, then display the corresponding Spanish word. However, if the user presses CANCEL in response to a display of the Spanish word, remove that Vocabulary item permanently from the list.
(c) FROM_ENGLISH is the opposite of (b): list Spanish words, then display the corresponding English.
(d) TRANSLATE means accept one English word from the user and display its translation.

If you have studied buttons and textfields, use a button for the three options (a), (b), (c) and a textfield to enter the English word for option (d).

7.12 **CourseSections:** Write a CourseSection class as follows: It has only these instance methods: `getMales()`, `getFemales()`, `getName()`, `toString()`, `changeMales(int)`, and `changeFemales(int)`; all instance variables are private. The first four methods listed simply return a value. The two change methods do nothing if the parameter is not in the range -2 to +2 inclusive; otherwise they change the number of people registered by the amount specified. The one CourseSection constructor should have one parameter, which is one line from the file.

The text file `data.txt` has a group of 1 to 20 lines of the form <males females sectionName> each representing one CourseSection object (all sectionNames different; males and females are ints; sectionName is String with no blanks; one space separation between parts). The text file `changes.dat` has a group of an unknown number of lines of the form <change sectionName> (change is one of "+m", "-m", "+f", or "-f", meaning "add 1 male", "remove 1 male", "add 1 female", "remove 1 female"; sectionName is one of those that appeared in data.dat).

Write an application that reads in the `data.dat` file, stores the information in an array, makes all the changes specified by the `changes.dat` file to the CourseSection objects, then prints representations of the revised CourseSection objects.

Sample data.dat file	Sample changes.dat file	Output for these files
17 4 CS152-71	+f CS151-02	17 4 CS152-71
12 13 CS151-02	-m CS253-01	13 14 CS151-02
5 15 CS253-01	-f CS253-01	4 14 CS253-01
	+m CS151-02	

Avoid all possible crashes, for bad input or otherwise, treating contrary-to-spec input lines as if those lines were not there. Use a second class of objects CourseSectionList for all list operations.

7.13 **OneDimensionalRandomWalk:** (Read the earlier RandomWalk project description to get the idea) One trial consists of a person walking 200 steps. Each step is either east or west, with equal probability. Record the final location of the person in an array. Repeat for 1000 trials. Print out the number of persons at each possible location, except ignore those more than 25 steps away from the starting point. Discuss the relation of the numbers you get to Pascal's Triangle.

7.14 **TwoDimensionalRandomWalk:** Same as the previous problem, but each step is either north or south or east or west. Record the final location of the person in a 2-dimensional array, ignoring those people who are more than 25 steps away from the starting point. Instead of printing out the whole 51-by-51 array, print the number of people N total steps away from the starting point for each value of N from 0 to 25. Extra credit: Do it for a 3-dimensional array (think of it as representing fish in a fish tank).

7.15 **PayWorkers &:** Write a program that reads in a file describing workers and also a file giving the five numbers for each worker that tell the number of hours worked each day. Accept keyboard input that updates these work hours for the next day. Produce a new file giving the updated five numbers (use redirection to create the new file).

7.16 **WorkersAges:** Some workers may have the same name. Write a program that reads in a file describing workers and then prints out each name that occurs more than once. Make sure you only print a name once, even if it occurs three or four times.

7.17 **EvilCarRentals &:** A car rental company charges its customers a $150 penalty if they go over 80 mph in the rental car and $100 if they go outside the designated driving area. Write a program that takes input from a text file (which is actually a remote terminal connected to GPS) for which each line gives the time and <x,y> position of one of up to ten rental cars (three double values, with positional coordinates measured in feet from an arbitrary reference point) as well as its VIN (a String). Each time a penalty is incurred, write out the time and nature of the offense and its VIN. Design an appropriate Car class of objects. The designated driving area is specified as two opposite corners of a rectangular area. Extra credit: The designated driving area is a quadrilateral.

7.18 **SieveOfErosthenes:** Create a class of objects as follows: The constructor has a positive int parameter that tells it how large an array to construct; initial each component a[k] contains the value k (you will ignore the first two components throughout). One method finds the lowest value not previously established to be a prime, notes that it is now known to be a prime, and overwrites in the array each multiple of that prime with zero. Another method tells whether all non-zero values in the array are known to be prime. Another method removes all zeros from the array by moving non-zeros towards the front of the array. And another method reports the number of primes found so far.

Then write a main method that calculates the percentage of primes in the first N integers for N having the values 1000, 2000, 3000, 4000, and 5000.

7.19 **Supermarket &:** A supermarket executive asks you to decide whether adding several 10-item or less lanes will improve customer satisfaction. You have data describing a typical store day's customer demand, in the form of a text file with 2 numbers per line. Each pair represents one customer: the arrival time at a checkout lane (measured in seconds since opening time) and the number of items purchased. The file ends with the pair 0 0 as a sentinel. History indicates that checkout processing requires an average of 5 seconds per item plus 30 seconds of overhead (for collecting the money, etc.). The executive specifies that the "dissatisfaction rating" of a particular choice of express lines is to be the amount of time spent waiting in line, averaged over all customers. [problem taken from Pascal: Problem-Solving and Programming With Style, by Dr. William C. Jones, Jr., copyright 1986 by Harper & Row.]

7.20 **SalesReport &:** A telemarketing firm wants us to create a sales-report program to process the monthly sales figures of each salesperson. The data available is a list of salesperson names, sales figures for the month, and hours spent on the phone in that month. The program is to compute the median sales figure and list the name of the salesperson with the highest sales per hour (so he/she can be given a bonus) and the name of the sales-person with the lowest sales per hour (so he/she can be fired). However, the bonus requires being more than 20% above the median and working at least 100 hours in the month. And a salesperson is fired only if he/she was at least (a) 20% below the median if working at most 50 hours in the month, (b) 25% below the median if working at least 150 hours in the month, (c) on a sliding scale between 20% and 25% for working between 50 and 150 hours in the month.

7.21 **ArrayLists:** Rewrite all the programs in Chapter Seven using ArrayLists as described at the end of Chapter Seven, instead of array brackets.

8.1 **GradeHistogram**: Develop a class with the following methods:

```
public GradeHistogram ();  // constructor; set all values to 0
public void setGrades (String grades);
   // count how many of each of A,B,C,D,F appears in grades
public char mode();     // query which of the capital letters
              // A,B,C,D,F has the highest frequency
public void showHistogram (Graphic page, int xcor, int ycor);
   // display histogram for the counts of the five characters
   // A,B,C,D,F with (xcor, ycor) as its bottom-left corner
```

Test with an applet or frame having this coding for the paint() method:

```
public void paint (Graphics page)
{  GradeHistogram cs1 = new GradeHistogram();
   cs1.setGrades ("ABC ABCD ABCBc  eABCF");
   System.out.println ("The mode is " + cs1.mode());
   cs1.showHistogram (page, 50, 110);
   cs1.showHistogram (page, 180, 120);
   cs1.setGrades ("AAAFFFFCCBBB");
   cs1.showHistogram (page, 50, 300);
}  //=====================
```

The histogram itself lists the five capital letters horizontally. Above each letter is a column of N rectangles, where N is the number of times that letter appeared in the most recent grades parameter. Each rectangle is 5 pixels high and 10 pixels wide.

8.2 **SlotMachine &:** Write an applet or frame that simulates a slot machine with its spinning wheels. It should have three windows with ten possible symbols for each window. Pay off for getting all three the same, paying the most if all three are stars. Choose minor payoffs for matching two symbols. Give the house an overall profit of between 2% and 5% (pay by credit card). Use animations and random numbers.

8.3 **DrawClock:** Write an applet or frame that reads in a time such as 3:15 or 10:45 using showInputDialog, then draws a clock with a big hand and little hand pointing to the right time. A minimal drawing is a circle plus two lines of different length extending from the center of the circle.

Alternative problem, for teaching young children to tell time: Repeatedly choose a clock time at random and draw the clock that shows that time. Ask the young user to say what time it shows, then say whether it is right (within three minutes of the time that is indicated).

8.4 **ScissorsPaperRock:** Complete the applet for colliding balls described in Section 8.8: Define three subclasses of the Ball class with different drawImage methods, different hitsOther methods, and running counts of each kind of object. Extra credit: Have a dying Ball slowly drift down to the bottom as it deteriorates, and be reborn when it gets there. Or add a button or slider to affect the movements.

8.5 **MazeDrawer:** Write a program that draws a solvable maze. If e.g. you choose to create a 40-by-30 maze, you could proceed as follows: Think of a rectangular area as being divided into 30 rows of 40 small squares. Note that each of the 1200 small squares has 4 walls, most of which are shared between two squares, so there are somewhat over 2400 walls that could be drawn in. First draw a meandering path from the lower-left corner to the upper-right corner. Then repeatedly choose a wall to fill in that does not block the path you chose. Fill in about 40% of the possible walls. Then print the maze, with a special marker on the starting (lower-left) and ending (upper-right) corners. Allow the user to print it both without the path marked (the challenge) and with the path marked (the solution).

8.6 **Yahtzee &:** Design and implement a program that supervises 2 to 4 players of Yahtzee, keeping records as needed.

8.7 **Elevators &:** Design an implement a program that simulates the movements of 3 elevators in a 20-story building. Be sure to have Elevator objects (each knows the floor it is on and the direction it is moving) and Person objects (each knows the floor it is on and the floor it wants to go to). Assume that one time unit is the time to move an elevator up or down one floor or half the time an elevator spends loading and/or unloading at one floor. Assume for simplicity that elevators never become full. Include a description of a reasonably efficient processing for deciding which elevators go where, in such a way that no person waits for more than 100 time units to get an elevator. Have Person objects appear at the elevator landings at random, on average once each 5 time units.

9.1 **Scrolling Marquee:** Write a program that creates a full-screen frame with a scrolling marquee. Initially it says "Hello World!". Have a textfield in the bottom-right corner where the user can enter a new phrase to be scrolled. Alternative: Make an applet.

First display the given String of characters at the far right of the screen at a randomly chosen height. Most of the characters will not show because they are off the screen to the right. Then move the String a few pixels to the left and repaint. Repeat until the String disappears off to the left. Then choose another random height and repeat. Extra credit: keep changing the font shape and size.

9.2 **GraphicalGames:** Write a game-playing program with graphical user interface: Display a main frame that allows the user to choose one of the three games GuessNumber, Nim, or Mastermind described in Chapter Four. Revise each of these games to have input from a textfield or a slider that stays on the screen through the game rather than using JOptionPane.

9.3 **AddressBook &:** Write a program with a graphical user interface that allows the user to create, update, and search a personal address book (names, phone numbers, email, addresses, birthdays), including the option of listing certain subsets in order.

APP-37

9.4 **ShoppingCart &:** Write an applet or application program that behaves much as the shopping cart applets for online shopping. Read a file to get the information about products. The main screen should have at least three categories of products to click on and display the total money due so far. Each category should have from 3 to 10 items in it that a buyer can choose.

9.5 **Solitaire:** Use a frame and graphical components for this program. The program shuffles a standard 52-card deck and deals it in 4 rows of 13 cards. The player repeatedly clicks on two adjacent cards of the same suit or of the same rank (the last card on one row is considered adjacent to the first card on the next row below). If the two cards clicked meet this criterion, the program removes those two and moves the rest closer together to fill up the gap. Play stops when no legal move is left. The score is the number of cards removed, so the highest score possible is 52.

Card objects should have two characters for instance variables: the rank from the 13 char values {A,2,3,4,5,6,7,8,9,T,J,K,Q} and the suit from the four char values {C,D,H,S}. A random shuffle of the cards can be obtained as follows: for each int value k from 0 to 51 inclusive do...
{ r = a random int from k through 51; swap the card at index r with the card at index k;}.

9.6 **PatiencePoker:** This is similar to the preceding solitaire game but more challenging and interesting. The program shuffles the standard 52-card deck and shows 25 cards, one card at a time. After each card is shown, the player must put it in a 5-by-5 grid before seeing the next card. When all 25 cards have been placed (one card in each of the 25 parts of the grid), the program announces the player's score, which is the sum of the scores for the ten poker hands that show (5 in the horizontal rows and 5 in the vertical columns). The score is 2 for a pair, 5 for two pairs, and 10/15/20/25/50/75 for three-of-a-kind, straight, flush, full house, four-of-a-kind, and straight-flush, respectively. Hint: You can most easily tell what a 5-card hand counts as if you first count the number of cards that have the same rank as a card later in the hand: 1 means a pair, 2 means two-pair, 3 means three-of-a-kind, etc.

9.7 **VendingMachine &:** Design software to control a candy vending machine. It will need a Selection object (responds to the customer pressing a button to choose one particular product bin), a CoinSlot object (responds to the customer entering a particular denomination of coin or bill), a ReturnMoney object (responds to the customer pressing the `returnMyMoney` button), a MoneyCounter object (tracks the amount entered and the coins available to make change), and a list of Bin objects (one per kind of product; tracks the price and availability of the product). Allow for refills and service by the manager.

9.8 **Bingo &:** Write a program that lets two players play Bingo. Each gets two different cards, chosen from a store of ten cards. Call letter/number pairs (a different letter/number pair each time) until one of the cards wins (5 in a line, any of 12 possible). Keep track of total winnings. Include Card objects, LetterNumber objects, and Player objects.

Additional problems for Chapter Nine: Most of the problems specified for Chapter Seven can be written using a Graphical User Interface instead of what they describe.

10.1 **GraphicalStocks:** Revise the Investor software to display the output in a text area and get input from a textfield and buttons. Add a chart to track the progress of the investments.

10.2 **Speculator &:** Modify the Investor software to have assets follow trends. One day's multiplier is to be 70% random and 30% calculated from the trend for the previous few days. Research how to calculate a 9-day Exponential Moving Average and use that for the trend. The description of an asset should give the current trend number on a scale from 0 to 10, 10 being rapidly rising and 0 being rapidly falling.

10.3 **StockRebalancer:** Modify the Investor software as follows: Write a program that accepts from the user five percentages -- non-negative integers that add up to 100. Create a Portfolio that has these percentages in each of the five asset classes. Also ask for an initial balance, a monthly deposit, and a number of years. Example: If the input is 10, 20, 40, 15, 15, the user will have 10% in the money market, 20% in 2-year bonds, 40% in large-cap stock, and 15% in each of small-cap stock and emerging markets. If the user also specifies $80,000, -300, and 20, that means that the user starts with an initial balance of $80,000, spends $300 a month from the account, and this continues for 20 years.

Run a simulation for that many years with that monthly deposit, rebalancing the total assets among the five asset classes to the original proportions once each month. In the example, you would start with $80,000 and, at the end of each month for 240 months, subtract $300 and then shift assets so that you have 10% in the money market, 20% in 2-year bonds, etc. Repeat the simulation 99 times and list (a) the highest outcome, (b) the lowest outcome, and (c) the average outcome (geometric average). The purpose of the exercise is to show the user what the range of outcomes would be if the user were to use this "constant-ratio asset allocation" method, ignoring any trending in the markets.

Extra credit: Keep track of all 99 results in an array of values. Each time you add a value, insert it in increasing order. After all 99 repetitions have been made, list the first, tenth, fiftieth, ninetieth, and ninety-ninth values. This gives the user an even better idea of the possible outcomes.

11.1 **Multiplication:** Write an application program that tests out all the services offered by the VeryLong class. Complete all methods in the VeryLong class (the hard one is the multiply method). Extra credit: Add a method to find the square root of a VeryLong value.

11.2 **UltraLong**: Write an UltraLong subclass of Numeric, like VeryLong except the array can have any positive number of components. For instance, adding two values requires allowing for one having a longer array than the other and never produces the wrong answer just because the result is too large. Store the most significant part of the number in the highest-indexed component, not in component zero. That highest-indexed component is to have a positive int value when the array has more than one component. Keep the meaning of the constructor with three long parameters. You may disable the multiply method. Extra credit: Include the multiply method. A second constructor is the following:

```
public UltraLong (int mantissa, int expo)
{  if (mantissa < 0 || expo < 0)
   {  itsItem = new int [1];  // just zero
   }else
   {  long mant = mantissa;
      for (int i = expo % 9;  i > 0;  i--)
         mant *= 10;

      if (mant < LONGBILL)
      {  itsItem = new int [expo / 9 + 1];
         itsItem[itsItem.length - 1] = (int) mant;
      }else
      {  itsItem = new int [expo / 9 + 2];
         itsItem[itsItem.length -1] = (int) (mant / LONGBILL);
         itsItem[itsItem.length -2] = (int) (mant % LONGBILL);
      }
   }
}  //======================
```

11.3 **PolarCoordinates:** Add three methods to the Complex class: a class method `fromPolar(r,t)` returns the Complex object whose polar coordinates are radius r and angle t; instance methods `getRadius()` and `getAngle()` return those two attributes.

11.4 **AlgebraicIntegers:** Implement a new subclass of Numeric: An AlgInt object represents a number of the form a + b*s where a and b are integers and s is a square root of 3. Extra credit: Add a method that tells whether a given AlgInt value is a prime number in the sense of having exactly two positive AlgInt divisors. Note that 13 is not an AlgInt prime because 13 = (4 + s) * (4 - s).

11.5 **Polynomials:** Implement a new subclass of Numeric: A Polynomial object represents a polynomial with double coefficients and no term of higher degree than 9. A polynomial is to be considered positive when its leading coefficient is positive. Extra credit: add methods to evaluate the polynomial for a given x-value and to find the derivative of the polynomial.

11.6 **BooleanAlgebra:** Implement a new subclass of Numeric: A Group object represents a finite set of positive integers. The sum, difference, and product of two Groups corresponds to set union, set subtraction, and set intersection. Hint: Store as an array of positive ints in increasing order.

12.1 **Concordance:** Write a program that reads in a text file and produces a concordance. This is a listing of many lines of text, one for each occurrence of a principal word in the original text file. Each line contains the principal word plus the three words that come before it and the three words that come after it in the original text. The listing is in alphabetical order of principal words. The listings for any one principal word are in the order they occur in the original text. For example, if you define a principal word to be any word of six or more letters, a concordance of the first three sentences of this paragraph is as follows:

```
words that come before it and the
and produces a concordance.  This is a
file.  Each line contains the principal word
This is a listing of many lines
one for each occurrence of a principal
word in the original text file.  Each
it in the   original text.  The listing
occurrence of a principal word in the
line contains the principal word plus the
text file and produces a concordance.  This
Write a program that reads in
```

Define a principal word to be any word of N or more letters, where N is obtained from the command line (with a default of six). For a text file of any substantial size, this concordance can become quite large. It would be quite inefficient to keep inserting new lines in an array of tens of thousands of lines, keeping them all in order. So you are to use an array of 26 LineList objects, one LineList object for each letter of the alphabet. The component numbered 0 will contain the principal words beginning with a, the component numbered 1 will contain the b's (the first line in the example above), the component numbered 2 will contain the c's (the second and third lines in the example above), etc. Each LineList object is to be an array of objects that each have two instance variables: one is to be the String of three words that come before the principal word, and the other is to be the String of the remaining four words beginning with the principal word. This makes it easy to test the ordering and keep the array in increasing order.

Extra credit: Allow the user to specify that a principal word is any word of N or more letters, plus any word on a given list of words, minus any word on a second given list of words (all of these specifications are obtained from a separate input file). Also, allow the user to specify that the principal word be boldfaced or that each listing be in inverted order, illustrated by the following:

```
before it and the             -- words that come
concordance.  This is a       -- and produces a
contains the principal word   -- file.  Each line
```

Extra credit #2: Do not include any words from another sentence (a sentence ends with a period or exclamation mark or question mark or colon). If that shortens the phrase on one end, use four words from the other end (unless that would go outside the current sentence).

12.2 **ExamGrades &:** Write a program to handle exam grades for one group of students. Maintain a permanent file of Student data where each Student object includes five exam grades (integers). Also maintain a permanent file of Exam data where each of five Exam objects includes the correct answers to the exam, the number of people who got each question right (an array of non-negative integer values), and the highest and lowest grades on the exam.

From time to time, read in a file containing student answers to a single exam, one line per student. Use this data to update the Student and Exam files. Use appropriate methods in the Student and Exam classes. Write an application program that would be useful to a teacher keeping track of student grades for the semester.

12.3 **MergeFiles:** You have a file that holds five to ten times as much data as you can get into RAM at one time. For instance, it may have several hundred thousand data values, but RAM can only hold 40,000. You are to produce a new file that contains that data in sorted order. Do it as follows: Read 40,000 data values into RAM and sort them in some reasonable way, then write them to file#0. Repeat for 40,000 sorted values into file#1. Continue until you have perhaps file#8 only partially filled. These files should be in an array of files. This completes Phase 1.

Phase 2 is to read the first (smallest) value from each file into an array of (in this example) 9 data values (one per file). Find the smallest. Write it to the endResult file. Replace it by the next data value from its file. Repeat until done. Hint: For testing purposes, just put 40 in each file.

12.4 **RandomAccess:** The company that wants email tracked as described in Chapter Twelve has a thousand employees. For each possibility of one employee sending email to another, the company wants you to store the subject lines of the five most recent emails for analysis (up to 80 characters per subject line). Implement the EmailSet class using a random-access file, since this amount of data is beyond the RAM capacity of the computer being used.

12.5 **VendingMachine &:** Design and implement software to control a candy vending machine. It will need to interact with a Slot object (tells what coins or bills are entered), a Money object (tells what coins are available as change), a Selection object (tells what candy the customer chooses, and whether the returnMyMoney button has been pressed), and an array of Bin objects (one per kind of candy item). Allow for refills and service by the manager. Also design a GUI simulation so it can be tested.

13.1 **TimingSorts:** Write a program that chooses 3000 Double values at random, sorts them by each of the selectionSort and the insertionSort methods, and reports the elapsed time. Call a method to do this 20 times and give the average of the 20 for each sorting method. Then call the method again for 6,000 values, then again for 12,000 values. Write an explanation of how your results confirm or contradict the expected big-oh behavior of these sorting algorithms. Extra credit: Include the Merge Sort algorithm.

13.2 **RiseAndFall:** Implement the "rise-and-fall" logic for the MergeSort described at the end of the section on the MergeSort.

13.3 **MostlySorted:** Modify the MergeSort logic so that it executes much faster when the elements in the array are nearly in order already. Specifically, you always check before merging whether the elements are already in order. If so (the highest element in the lower group is less than or equal to the lowest element in the upper group), then do not do any merging. Have the private sorting methods return a boolean value: true if they actually merged, false if they did not move the groups as they were supposed to (as thus the group is left in the wrong array). Adjust the logic for this returned value.

14.1 **StockTransactions &:** Write out the full program for the buying and selling of company stock described in Section 14.1.

14.2 **MatchingParentheses:** Write a program that reads in several lines of input and for each one, tells whether its grouping symbols are nested properly, as described in Section 14.1. Ignore everything on a line except for `( ) [ ] { }`. Store symbols on a stack until matched. Extra credit: Ignore everything that is inside a pair of quotes and everything that comes after `//` on the line.

14.3 **Queue:** Write a complete implementation of QueueADT using the "avoid-all-moves" approach described in Section 14.2.

14.4 **DoubleEndedQueue:** Write a complete array implementation of a class that allows the queue operations plus adding to the front of the list and deleting from the rear of the list. Use the concept shown in Listing 14.3. However, start with `itsFront` initially `itsItem.length / 2` so you can add to the array in either direction. When you get to one end of the array, shift all the values back to the middle of the array (unless the array is full).

14.5 **DMV:** The Department of Motor Vehicles in a large city has 26 clerks at 26 windows, one for each letter in the alphabet. Each customer who comes in signs up by last name at a computer terminal. Each time a clerk finishes with a customer, the clerk presses a button, which causes this computer to display the next customer's last name on a screen above the clerk (that is, the next customer whose first letter of the last name matches the clerk). Write the software to control the computer with a GUI interface having three panels: one for sign-in, one for 26 buttons, and one for 26 displays.

14.6 **UnorderedPriQue:** Write a complete implementation of PriQue using a partially-filled array of Store objects. The Store class is nested inside the UnorderedPriQue class. A Store object has two instance variables: One is the element being stored, the other is an int value that counts the total number of values added to the priority queue so far. The following is the `add` method; you have to write the rest of the class including Store. When you remove a value and there are several with the same highest priority, you can easily find the one that was added earliest. And you can simply replace it with the first value in the list. Alternative problem: Do this with a linked list rather than a partially-filled array.

```
public void add (Object ob)
{  if (itsSize == itsItem.length)
      // same as in ArrayStack's push method, Listing 14.2
   itsNumAdded++;
   itsItem[itsSize] = new Store (ob, itsNumAdded);
   itsSize++;
}  //======================
```

14.7 **PostfixCalculator &:** Write a program that acts as a calculator using postfix notation. The JFrame should have 10 buttons with the 10 digits on them, a button with a decimal point on it, a button with ~ on it, a button with PUSH on it, and four "operator" buttons with +, -, *, and / on them. A label displays the digits in the currently-entered number. Pressing the PUSH key pushes the number currently on display onto a stack of numeric values. Pressing a digit button appends it to the displayed number, unless it has been pushed, in which case pressing a digit button makes that digit the first and only digit in the displayed number. Pressing the decimal point appends it to the displayed number if it does not already have a decimal point. The ~ button only has an effect when pressed as the first entry for a number; it represents a negative sign (as opposed to the - button which represents subtraction of two numbers).

Pressing an operator button has the push effect if the displayed number has not yet been pushed. It then pops the two top values off the stack and displays the result (the sum, difference, product, or quotient depending on the operator symbol). Finally, it pushes the result. In addition, a JTextArea shows all the values currently on the stack and a CLEAR button clears the display.

14.8 **Anagrams:** Write a program that accepts as input two positive integers N and n and then prints out 24 anagrams of the first N letters of the alphabet starting with the nth anagram. Use the logic illustrated in Section 14.1, in which you keep the letters of the current anagram in a stack in reverse order and move from one anagram to the next with the help of just one queue. The anagrams are numbered in alphabetical order, so for N = 6, the first is abcde, the second is abced, the third is abdce, etc. The reason for the restriction to 24 is that printing all of them would print N-factorial lines, which is 5040 when N is just 7. Add to your stack class a method that returns the toString() value of all the elements on the stack in reverse order. Extra credit: Write an alternative version in which you keep the current anagram in a queue in reverse order and move from one anagram to the next with the help of just one stack.

14.9 **RadixSort:** Write a program that reads in a file containing one word per line and prints them out in alphabetical order, as follows: First add to NodeQueue a method `public void add (NodeQueue queue)` for which the executor adds the contents of queue after its own elements, keeping the same relative order. Read the file into a single NodeQueue. Apply the logic of the radix sort described at the end of Chapter Fourteen, one pass per character, with an array of 26 NodeQueues. Only use the first ten characters, so make ten passes that put values in the array of NodeQueues and then back into a single NodeQueue. Finally, apply an insertion sort to take care of the very few cases of two words that have the same first ten characters.

15.1 **BasicSequence:** Write out the full BasicSequence and BasicSequenceIterator classes. Include a constructor with a Collection parameter. Write an application program that tests almost all of the methods.

15.2 **LinkedWorkerList:** Rewrite the entire WorkerList class (Listings 7.8 and 7.9) to use a linked list of Nodes rather than an array.

15.3 **TrackRear:** Rewrite the NodeSequence and NodeSequenceIterator classes so that a NodeSequence object has another instance variable itsLast, which is always the last node in the sequence (or null if the sequence is empty). Use it to make the add method much more efficient.

15.4 **Positions:** Write a Position interface that extends Iterator (with `remove` as an unsupported operation) by adding `getNext()` to return the element that `next()` would return, but without moving further in the sequence. Define them so that the standard loop for arrays translates as shown:

```
for (k = 0;  k < itsSize;  k++)
   {x = itsItem[k];...}
for (p = position(); p.hasNext(); p.next())
   {x = p.getNext();... }
```

Position also has `setNext(anObject)` to replace the value that `getNext()` returns by `anObject`; `addNext(anObject)` to insert `anObject` before what `getNext()` returns and make it the value that `getNext()` returns (inserting at the end if `hasNext()` is `false`); and `removeNext()` to remove the value that `getNext()` returns. Define `next()` to be the equivalent of `getNext(); moveOn()`. Write out a full implementation of Position as a inner class of NodeSequence, possibly with the modifications for the next problem. Allow a call of any method whenever `hasNext()` is true and allow `addNext` anytime. Write an essay comparing this approach with having `add`, `remove`, and `set` added to Iterator.

15.5 **OrderedSequence:** Write a Collection implementation that throws an Exception if some object is added that is not Comparable. Store all objects in a linked list in increasing order as determined by compareTo. Include a constructor that makes an ordered sequence from a Collection parameter with unordered Comparable values.

15.6 **HeaderNode:** Rewrite the NodeSequence and NodeSequenceIterator classes so that a NodeSequence's `itsFirst` value acts as a dummy header node with no data, as follows: Rename `itsFirst` as `itsNext` throughout. Delete the `itsNext` field from the nested Node class and make Node a subclass of NodeSequence (as well as being nested in it). Then the first element of the abstract sequence is in `itsNext.itsData`, except a NodeSequence's `itsNext` is null if the sequence has no elements. Then the `add` method is greatly simplified by replacing its call of `addLater` by a recursive call of `add` and eliminating the `addLater` method entirely.

The real improvement is in the ListIterator methods. The iterator does not create its own dummy header node; instead, it initializes `itsPos = NodeSequence.this` (which requires that itsPos be declared as a NodeSequence). Simplify the coding of the rest of the ListIterator methods where possible. Extra credit: (a) Add `push`, `pop`, and `peekTop` methods so NodeSequence implements StackADT; do not call on iterator methods or other methods in the NodeSequence class. (b) Use recursion in place of every loop.

15.7 **InternalSorts:** Add two methods to the NodeSequence class, one to do a SelectionSort and one to do an InsertionSort. Revise the TimingSorts project for Chapter Thirteen to compare these two linked-list sorting methods on execution times. Hint for the InsertionSort: Denote the first node as `lastGood`, since it is the last node in the initial sorted list of one element. As you insert nodes into the sorted list, keep `lastGood` always indicating the last node in the sorted list. So the `bad` node you are to insert next is always the one after `lastGood` and there are three cases to code: Either `bad` is inserted after `lastGood` (that is easy) or `bad` is removed from where it is and then either inserted as the first node on the list or else inserted later in the list.

15.8 **DoublyLinkedList:** Write a List implementation, together with a full implementation of ListIterator, using a doubly-linked list as for TwoWaySequence and TwoWaySequenceIterator, except do not have a header node. Hint: Include a private method `find(indexInt)` to return the Node corresponding to `indexInt`.

15.9 **Word Chains:** Your program should read in a list of four-letter words, one per line, from a text file, and store it in a Collection object. Then repeat the following as long as the user wants to continue: Ask the user for two four-letter words (which may or may not be in the file) and either (a) print some chain starting from the first word and ending with the second word, or (b) tell the user that no exists. A chain is defined to be a sequence of words, all from the file except possibly the two inputs, such that each differs in only one letter from the one before it in the sequence. Example: If the user's input is "mail" and "down", you could produce mail -> main -> fain -> fawn -> dawn -> down if the four intermediate words are in the file.

Hint: For each pair of inputs, first make a copy x of the file's collection. Then there is a chain from firstInput to secondInput only if there is a someWord one letter different from firstInput for which there is a chain from someWord to secondInput. Each time you find a value for someWord, remove it from your copy before seeing whether you can make a chain to secondInput. Use recursion.

16.1 **Parser:** Write the Parser class with the `parseInput` method described in Chapter Sixteen, capable of processing the subset of the Scheme language described in that chapter. Add a main method you can use to test it: Repeatedly get a line of input and say whether it is acceptable.

16.2 **OrderedArrayMap:** Write an array implementation of AbstractMap in which the Entry objects are kept in ascending order of keys at all times. Have itsKey be of a class that implements the Comparable interface (i.e., has the usual compareTo method). Use binary search for the containsKey method. When the array becomes full, expand it by 50%. Add another map method named display that prints to System.out all the values in order (this is useful for debugging purposes).

Also write a main method that creates two maps, named `oneTime` and `manyTimes`, that store word/definition pairs, both objects being Strings. The main method should then read in a sequence of word/definition pairs and store in `oneTime` those words that appear once in the input, but in `manyTimes` those words that appear more than once in the input. When input is complete, print all the word/definition pairs in both maps in order with appropriate titles. Hint: The first time a word appears in the input, put it in `oneTime`; the second time it appears, take it out of `oneTime` and put it in `manyTimes`.

16.3 **TimingLinkedSorts:** Same as Problem 13.1, TimingSorts, except implement all three sorting methods on linked lists with trailer nodes rather than arrays. Hint for Insertion Sort: Start a new list containing only the first data value. Go through the rest of the original list one value at a time and insert it into the new list where it goes.

16.4 **HeaderListMap:** The NodeMap class has N+1 nodes whenever it contains N entries, with the last node (the empty list) having null for its entry. That is a trailer node. Write a HeaderNodeMap class that implements a Mapping as a linked list with N+1 nodes whenever it contains N entries, except that it is the first node (the "header node") that has `null` for its entry. Hint: `containsKey` should set `thePosition` to the node before the one that contains the entry that was found, so `put` can insert a node after `thePosition's` node containing the entry to be added.

16.5 **MapIt:** Write the remove method for the MapIt class for NodeMap. Add in fail-fast protection: If the program ever calls directly on a NodeMap object X's remove or put operation after a particular MapIt object is constructed for X, and thereafter calls some MapIt method for that particular MapIt object, the method throws an IllegalStateException. This prevents something like getting the next value using the Iterator and removing it using the NodeMap remove method and then calling next again, thereby confusing the Iterator object. Hint: Add an instance variable to the NodeMap class that counts the total number of puts and removes that have been executed on a given NodeMap object.

16.6 **BinaryDeletion:** Write the remove method for the binary search tree implementation of AbstractMap.

17.1 **Pathfinder:** Write an alternative MapIt class nested in the BinTree class, as follows: Each iterator keeps track of a stack of BinTree objects, which are all the BinTrees seen on the path through the tree starting from the root and going down to the current position in the tree.

17.2 **RedBlackRemove:** Write the remove method for the red-black tree logic used for Listing 17.10.

17.3 **AVL:** Revise the entire BinTree implementation of Mapping to use the AVL tree logic.

17.4 **Btree**: Write a Btree class to fully implement the Mapping iterface using B-trees. Assume the existence of two random-access files, one an index file organized as a B-tree and the other the data file containing the actual data.

Many More Problems: Hundreds more problems are available from the Association for Computing Machinery website http://acm.uva.es/problemset/. These are contest problems. You may submit solutions to be evaluated by a non-human judge.