AP Computer Science AB

Content of Reference Materials for AP CS AB Exam

Page
Appendix A – Java Quick Reference A1
Appendix B – Source Code for Visible Classes... B1
Appendix C – Black Box Classes.. C1
Appendix D – Environment Implementations.. D1
Appendix F – JMBS Quick Reference F1
Appendix G – Index for Source Code G1

Appendix A -- Java Quick Reference

`class java.lang.Object` // all classes inherit and may override these methods
- `boolean equals(Object other)`
- `String toString()`
- `int hashCode()`

`interface java.lang.Comparable`
- `int compareTo(Object other)` // return value < 0 if `this` is less than `other`
  // return value = 0 if `this` is equal to `other`
  // return value > 0 if `this` is greater than `other`

`class java.lang.Integer implements java.lang.Comparable`
- `Integer(int value)` // constructor
- `int intValue()`

`class java.lang.Double implements java.lang.Comparable`
- `Double(double value)` // constructor
- `double doubleValue()`

`class java.lang.String implements java.lang.Comparable`
- `int length()`
- `String substring(int from, int to)` // returns the substring beginning at `from`
  // and ending at `to-1`
- `String substring(int from)` // returns `substring(from, length())`
- `int indexOf(String s)` // returns the index of the first occurrence of `s;`
  // returns `-1` if not found

`class java.lang.Math`
- `static int abs(int x)`
- `static double abs(double x)`
- `static double pow(double base, double exponent)`
- `static double sqrt(double x)`

`class java.util.Random`
- `int nextInt(int n)` // returns an integer in the range from `0` to `n-1` inclusive
- `double nextDouble()` // returns a double in the range [0.0, 1.0)

`interface java.util.List`
- `int size()`
- `boolean add(Object x)` // appends `x` to end of list; returns `true`
- `Object get(int index)`
- `Object set(int index, Object x)` // replaces the element at `index` with `x`
  // returns the element formerly at the specified position
- `Iterator iterator()`
- `ListIterator listIterator()`

class java.util.ArrayList implements java.util.List

- Methods in addition to the `List` methods:
- `void add(int index, Object x)` // inserts `x` at position `index`, sliding elements
  // at position `index` and higher to the right
  // (adds 1 to their indices) and adjusts size
- `Object remove(int index)` // removes element from position `index`, sliding elements
  // at position `index + 1` and higher to the left
  // (subtracts 1 from their indices) and adjusts size
  // returns the element formerly at the specified position

class java.util.LinkedList implements java.util.List

- Methods in addition to the `List` methods:
- `void addFirst(Object x)`
- `void addLast(Object x)`
- `Object getFirst()`
- `Object getLast()`
- `Object removeFirst()`
- `Object removeLast()`

interface java.util.Set

- `boolean add(Object x)`
- `boolean contains(Object x)`
- `boolean remove(Object x)`
- `int size()`
- `Iterator iterator()`

class java.util.HashSet implements java.util.Set
class java.util.TreeSet implements java.util.Set

interface java.util.Map

- `Object put(Object key, Object value)` // associates `key` with `value`
  // returns the value formerly associated with `key`
  // or `null` if `key` was not in the map
- `Object get(Object key)`
- `Object remove(Object key)`
- `boolean containsKey(Object key)`
- `int size()`
- `Set keySet()`

class java.util.HashMap implements java.util.Map
class java.util.TreeMap implements java.util.Map

interface java.util.Iterator

- `boolean hasNext()`
- `Object next()`
- `void remove()`

interface java.util.ListIterator extends java.util.Iterator

- Methods in addition to the `Iterator` methods
- `void add(Object x)`

- `void set(Object x)`

Implementation classes for linked list and tree nodes

Unless otherwise noted, assume that a linked list implemented from the `ListNode` class does not have a dummy header node.

```
public class ListNode
{
  private Object value;
  private ListNode next;

  public ListNode(Object initValue, ListNode initNext)
    { value = initValue; next = initNext; }

  public Object getValue() { return value; }
  public ListNode getNext() { return next; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setNext(ListNode theNewNext) { next = theNewNext; }
}
```

Unless otherwise noted, assume that a tree implemented from the `TreeNode` class does not have a dummy root node.

```
public class TreeNode
{
  private Object value;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(Object initValue)
    { value = initValue; left = null; right = null; }

  public TreeNode(Object initValue, TreeNode initLeft, TreeNode initRight)
    { value = initValue; left = initLeft; right = initRight; }

  public Object getValue() { return value; }
  public TreeNode getLeft() { return left; }
  public TreeNode getRight() { return right; }

  public void setValue(Object theNewValue) { value = theNewValue; }
  public void setLeft(TreeNode theNewLeft) { left = theNewLeft; }
  public void setRight(TreeNode theNewRight) { right = theNewRight; }
}
```

Interface for stacks (* See note at end of reference)

```
public interface Stack
{
  /**
   * postcondition: returns true if stack is empty;
   *                otherwise, returns false
   */
  boolean isEmpty();

  /**
   * precondition:  stack is [e1, e2, ..., en] with n >= 0
   * postcondition: stack is [e1, e2, ..., en, x]
   */
  void push(Object x);

  /**
   * precondition:  stack is [e1, e2, ..., en] with n >= 1
   * postcondition: stack is [e1, e2, ..., e(n-1)]; returns en
   * exceptions:    throws an unchecked exception if the stack is empty
   */
  Object pop();

  /**
   * precondition:  stack is [e1, e2, ..., en] with n >= 1
   * postcondition: returns en
   * exceptions:    throws an unchecked exception if the stack is empty
   */
  Object peekTop();
}
```

Interface for queues (* See note at end of reference)

```
public interface Queue
{
  /**
   * postcondition: returns true if queue is empty;
   *                otherwise, returns false
   */
  boolean isEmpty();

  /**
   * precondition:  queue is [e1, e2, ..., en] with n >= 0
   * postcondition: queue is [e1, e2, ..., en, x]
   */
  void enqueue(Object x);

  /**
   * precondition:  queue is [e1, e2, ..., en] with n >= 1
   * postcondition: queue is [e2, ..., en]; returns e1
   * exceptions:    throws an unchecked exception if the queue is empty
   */
  Object dequeue();

  /**
   * precondition:  queue is [e1, e2, ..., en] with n >= 1
   * postcondition: returns e1
   * exceptions:    throws an unchecked exception if the queue is empty
   */
  Object peekFront();
}
```

Interface for priority queues (* See note at end of reference)

```
public interface PriorityQueue
{
  /**
   * postcondition: returns true if the number of elements in
   *                the priority queue is 0;
   *                otherwise, returns false
   */
  boolean isEmpty();

  /**
   * postcondition: x has been added to the priority queue; the number
   *               of elements in the priority queue is increased by 1.
   */
  void add(Object x);

  /**
   * postcondition: The smallest item in the priority queue is removed
   *                and returned; the number of elements in the priority
   *                queue is decreased by 1.
   * exceptions:    throws unchecked exception if priority queue is empty
   */
  Object removeMin();

  /**
   * postcondition: The smallest item in the priority queue is returned; the
   *                priority queue is unchanged
   * exceptions:    throws unchecked exception if priority queue is empty
   */
  Object peekMin();
}
```

* Note regarding use of stacks, queues, and priority queues

When a stack, queue, or priority queue object needs to be instantiated, code such as the following is used:

```
Queue q = new ListQueue();       // ListQueue implements Queue
```

Appendix B -- Source Code for Visible Classes

This appendix contains implementations of the visible core classes covered in Chapters 1 - 4: `Simulation`, `Fish`, `DarterFish`, and `SlowFish`. Information about the `Environment` interface can be found in Appendix C.

Simulation.java

```
/**
 *  Marine Biology Simulation:
 *  A Simulation object controls a simulation of fish
 *  movement in an Environment.
 *
 *  @version 1 July 2002
 **/

public class Simulation
{
    // Instance Variables: Encapsulated data for each simulation object
    private Environment theEnv;
    private EnvDisplay  theDisplay;

    /** Constructs a Simulation object for a particular environment.
     *  @param env     the environment on which the simulation will run
     *  @param display an object that knows how to display the environment
     **/
    public Simulation(Environment env, EnvDisplay display)
    {
        theEnv = env;
        theDisplay = display;

        // Display the initial state of the simulation.
        theDisplay.showEnv();
        Debug.println("-------- Initial Configuration --------");
        Debug.println(theEnv.toString());
        Debug.println("---------------------------------------");
    }

    /** Runs through a single step of this simulation. **/
    public void step()
    {
        // Get all the fish in the environment and ask each
        // one to perform the actions it does in a timestep.
        Locatable[] theFishes = theEnv.allObjects();
        for ( int index = 0; index < theFishes.length; index++ )
        {
            ((Fish)theFishes[index]).act();
        }

        // Display the state of the simulation after this timestep.
        theDisplay.showEnv();
        Debug.println(theEnv.toString());
        Debug.println("-------- End of Timestep --------");
    }
}
```

Fish.java (includes breeding and dying modifications from Chapter 3)

```
import java.awt.Color;
import java.util.ArrayList;
import java.util.Random;

/**
 *  Marine Biology Simulation:
 *  A Fish object represents a fish in the Marine Biology
 *  Simulation. Each fish has a unique ID, which remains constant
 *  throughout its life.  A fish also maintains information about its
 *  location and direction in the environment.
 *
 *  Modification History:
 *  - Modified to support a dynamic population in the environment:
 *    fish can now breed and die.
 *
 *  @version 1 July 2002
 **/

public class Fish implements Locatable
{
    // Class Variable: Shared among ALL fish
    private static int nextAvailableID = 1;   // next avail unique identifier

    // Instance Variables: Encapsulated data for EACH fish
    private Environment theEnv;        // environment in which the fish lives
    private int myId;                  // unique ID for this fish
    private Location myLoc;            // fish's location
    private Direction myDir;           // fish's direction
    private Color myColor;             // fish's color
// THE FOLLOWING TWO INSTANCE VARIABLES ARE NEW IN CHAPTER 3 !!!
    private double probOfBreeding;     // defines likelihood in each timestep
    private double probOfDying;        // defines likelihood in each timestep


  // constructors and related helper methods

    /** Constructs a fish at the specified location in a given environment.
     *  The Fish is assigned a random direction and random color.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     **/
    public Fish(Environment env, Location loc)
    {
        initialize(env, loc, env.randomDirection(), randomColor());
    }
    /** Constructs a fish at the specified location and direction in a
     *  given environment.  The Fish is assigned a random color.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     *  @param dir    direction the new fish is facing
     **/
    public Fish(Environment env, Location loc, Direction dir)
    {
        initialize(env, loc, dir, randomColor());
    }

    /** Constructs a fish of the specified color at the specified location
     *  and direction.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     *  @param dir    direction the new fish is facing
     *  @param col    color of the new fish
     **/
    public Fish(Environment env, Location loc, Direction dir, Color col)
    {
        initialize(env, loc, dir, col);
    }

    /** Initializes the state of this fish.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which this fish will live
     *  @param loc    location of this fish in env
     *  @param dir    direction this fish is facing
     *  @param col    color of this fish
     **/
    private void initialize(Environment env, Location loc, Direction dir,
                            Color col)
    {
        theEnv = env;
        myId = nextAvailableID;
        nextAvailableID++;
        myLoc = loc;
        myDir = dir;
        myColor = col;
        theEnv.add(this);

        // object is at location myLoc in environment

    // THE FOLLOWING INITIALIZATIONS ARE NEW IN CHAPTER 3 !!!
        // For now, every fish is equally likely to breed or die in any given
        // timestep, although this could be individualized for each fish.
        probOfBreeding = 1.0/7.0;   // 1 in 7 chance in each timestep
        probOfDying = 1.0/5.0;      // 1 in 5 chance in each timestep
    }
    /** Generates a random color.
     *  @return       the new random color
     **/
    protected Color randomColor()
    {
        // There are 256 possibilities for the red, green, and blue attributes
        // of a color.  Generate random values for each color attribute.
        Random randNumGen = RandNumGenerator.getInstance();
        return new Color(randNumGen.nextInt(256),    // amount of red
                         randNumGen.nextInt(256),    // amount of green
                         randNumGen.nextInt(256));   // amount of blue
    }


  // accessor methods

    /** Returns this fish's ID.
     *  @return        the unique ID for this fish
     **/
    public int id()
    {
        return myId;
    }

    /** Returns this fish's environment.
     *  @return        the environment in which this fish lives
     **/
    public Environment environment()
    {
        return theEnv;
    }

    /** Returns this fish's color.
     *  @return        the color of this fish
     **/
    public Color color()
    {
        return myColor;
    }

    /** Returns this fish's location.
     *  @return        the location of this fish in the environment
     **/
    public Location location()
    {
        return myLoc;
    }

    /** Returns this fish's direction.
     *  @return        the direction in which this fish is facing
     **/
    public Direction direction()
    {
        return myDir;
    }
    /** Checks whether this fish is in an environment.
     *  @return  true if the fish is in the environment
     *           (and at the correct location); false otherwise
     **/
    public boolean isInEnv()
    {
        return environment().objectAt(location()) == this;
    }

    /** Returns a string representing key information about this fish.
     *  @return  a string indicating the fish's ID, location, and direction
     **/
    public String toString()
    {
        return id() + location().toString() + direction().toString();
    }


  // modifier method

// THE FOLLOWING METHOD IS MODIFIED FOR CHAPTER 3 !!!
//     (was originally a check for aliveness and a simple call to move)
    /** Acts for one step in the simulation.
     **/
    public void act()
    {
        // Make sure fish is alive and well in the environment -- fish
        // that have been removed from the environment shouldn't act.
        if ( ! isInEnv() )
            return;

        // Try to breed.
        if ( ! breed() )
            // Did not breed, so try to move.
            move();

        // Determine whether this fish will die in this timestep.
        Random randNumGen = RandNumGenerator.getInstance();
        if ( randNumGen.nextDouble() < probOfDying )
            die();
    }
  // internal helper methods

// THE FOLLOWING METHOD IS NEW FOR CHAPTER 3 !!!
    /** Attempts to breed into neighboring locations.
     *  @return    true if fish successfully breeds;
     *             false otherwise
     **/
    protected boolean breed()
    {
        // Determine whether this fish will try to breed in this
        // timestep.  If not, return immediately.
        Random randNumGen = RandNumGenerator.getInstance();
        if ( randNumGen.nextDouble() >= probOfBreeding )
            return false;

        // Get list of neighboring empty locations.
        ArrayList emptyNbrs = emptyNeighbors();
        Debug.print("Fish " + toString() + " attempting to breed.  ");
        Debug.println("Has neighboring locations: " + emptyNbrs.toString());

        // If there is nowhere to breed, then we're done.
        if ( emptyNbrs.size() == 0 )
        {
            Debug.println("  Did not breed.");
            return false;
        }

        // Breed to all of the empty neighboring locations.
        for ( int index = 0; index < emptyNbrs.size(); index++ )
        {
            Location loc = (Location) emptyNbrs.get(index);
            generateChild(loc);
        }

        return true;
    }

// THE FOLLOWING METHOD IS NEW FOR CHAPTER 3 !!!
    /** Creates a new fish with the color of its parent.
     *  @param loc    location of the new fish
     **/
    protected void generateChild(Location loc)
    {
        // Create new fish, which adds itself to the environment.
        Fish child = new Fish(environment(), loc,
                              environment().randomDirection(), color());
        Debug.println("  New Fish created: " + child.toString());
    }
    /** Moves this fish in its environment.
     **/
    protected void move()
    {
        // Find a location to move to.
        Debug.print("Fish " + toString() + " attempting to move.  ");
        Location nextLoc = nextLocation();

        // If the next location is different, move there.
        if ( ! nextLoc.equals(location()) )
        {
            // Move to new location.
            Location oldLoc = location();
            changeLocation(nextLoc);

            // Update direction in case fish had to turn to move.
            Direction newDir = environment().getDirection(oldLoc, nextLoc);
            changeDirection(newDir);
            Debug.println("  Moves to " + location() + direction());
        }
        else
            Debug.println("  Does not move.");
    }

    /** Finds this fish's next location.
     *  A fish may move to any empty adjacent locations except the one
     *  behind it (fish do not move backwards).  If this fish cannot
     *  move, nextLocation returns its current location.
     *  @return    the next location for this fish
     **/
    protected Location nextLocation()
    {
        // Get list of neighboring empty locations.
        ArrayList emptyNbrs = emptyNeighbors();

        // Remove the location behind, since fish do not move backwards.
        Direction oppositeDir = direction().reverse();
        Location locationBehind = environment().getNeighbor(location(),
                                                              oppositeDir);
        emptyNbrs.remove(locationBehind);
        Debug.print("Possible new locations are: " + emptyNbrs.toString());

        // If there are no valid empty neighboring locations, then we're done.
        if ( emptyNbrs.size() == 0 )
            return location();

        // Return a randomly chosen neighboring empty location.
        Random randNumGen = RandNumGenerator.getInstance();
        int randNum = randNumGen.nextInt(emptyNbrs.size());
        return (Location) emptyNbrs.get(randNum);
    }
    /** Finds empty locations adjacent to this fish.
     *  @return    an ArrayList containing neighboring empty locations
     **/
    protected ArrayList emptyNeighbors()
    {
        // Get all the neighbors of this fish, empty or not.
        ArrayList nbrs = environment().neighborsOf(location());

        // Figure out which neighbors are empty and add those to a new list.
        ArrayList emptyNbrs = new ArrayList();
        for ( int index = 0; index < nbrs.size(); index++ )
        {
            Location loc = (Location) nbrs.get(index);
            if ( environment().isEmpty(loc) )
                emptyNbrs.add(loc);
        }

        return emptyNbrs;
    }

    /** Modifies this fish's location and notifies the environment.
     *  @param  newLoc    new location value
     **/
    protected void changeLocation(Location newLoc)
    {
        // Change location and notify the environment.
        Location oldLoc = location();
        myLoc = newLoc;
        environment().recordMove(this, oldLoc);

        // object is again at location myLoc in environment
    }

    /** Modifies this fish's direction.
     *  @param  newDir    new direction value
     **/
    protected void changeDirection(Direction newDir)
    {
        // Change direction.
        myDir = newDir;
    }

// THE FOLLOWING METHOD IS NEW FOR CHAPTER 3 !!!
    /** Removes this fish from the environment.
     **/
    protected void die()
    {
        Debug.println(toString() + " about to die.");
        environment().remove(this);
    }

}
```

DarterFish.java

```
import java.awt.Color;

/**
 *  Marine Biology Simulation:
 *  The DarterFish class represents a fish in the Marine
 *  Biology Simulation that darts forward two spaces if it can, moves
 *  forward one space if it  can't move two, and reverses direction
 *  (without moving) if it cannot  move forward.  It can only "see" an
 *  empty location two cells away if the cell in between is empty also.
 *  In other words, if both the cell in front of the darter and the cell
 *  in front of that cell are empty, the darter fish will move forward
 *  two spaces.  If only the cell in front of the darter is empty, it
 *  will move there.  If neither forward cell is empty, the fish will turn
 *  around, changing its direction but not its location.
 *
 *  DarterFish objects inherit instance variables and much
 *  of their behavior from the Fish class.
 *
 *  @version 1 July 2002
 **/

public class DarterFish extends Fish
{

  // constructors

    /** Constructs a darter fish at the specified location in a
     *  given environment.   This darter is colored yellow.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     **/
    public DarterFish(Environment env, Location loc)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, env.randomDirection(), Color.yellow);
    }

    /** Constructs a darter fish at the specified location and direction in a
     *  given environment.   This darter is colored yellow.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     *  @param dir    direction the new fish is facing
     **/
    public DarterFish(Environment env, Location loc, Direction dir)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, dir, Color.yellow);
    }
    /** Constructs a darter fish of the specified color at the specified
     *  location and direction.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     *  @param dir    direction the new fish is facing
     *  @param col    color of the new fish
     **/
    public DarterFish(Environment env, Location loc, Direction dir, Color col)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, dir, col);
    }


  // redefined methods

    /** Creates a new darter fish.
     *  @param loc    location of the new fish
     **/
    protected void generateChild(Location loc)
    {
        // Create new fish, which adds itself to the environment.
        DarterFish child = new DarterFish(environment(), loc,
                                          environment().randomDirection(),
                                          color());
        Debug.println("  New DarterFish created: " + child.toString());
    }

    /** Moves this fish in its environment.
     *  A darter fish darts forward (as specified in nextLocation) if
     *  possible, or reverses direction (without moving) if it cannot move
     *  forward.
     **/
    protected void move()
    {
        // Find a location to move to.
        Debug.print("DarterFish " + toString() + " attempting to move.  ");
        Location nextLoc = nextLocation();

        // If the next location is different, move there.
        if ( ! nextLoc.equals(location()) )
        {
            changeLocation(nextLoc);
            Debug.println("  Moves to " + location());
        }
        else
        {
            // Otherwise, reverse direction.
            changeDirection(direction().reverse());
            Debug.println("  Now facing " + direction());
        }
    }
    /** Finds this fish's next location.
     *  A darter fish darts forward two spaces if it can, otherwise it
     *  tries to move forward one space.  A darter fish can only move
     *  to empty locations, and it can only move two spaces forward if
     *  the intervening space is empty.  If the darter fish cannot move
     *  forward, nextLocation returns the fish's current
     *  location.
     *  @return    the next location for this fish
     **/
    protected Location nextLocation()
    {
        Environment env = environment();
        Location oneInFront = env.getNeighbor(location(), direction());
        Location twoInFront = env.getNeighbor(oneInFront, direction());
        Debug.println("  Location in front is empty? " +
                        env.isEmpty(oneInFront));
        Debug.println("  Location in front of that is empty? " +
                        env.isEmpty(twoInFront));
        if ( env.isEmpty(oneInFront) )
        {
            if ( env.isEmpty(twoInFront) )
                return twoInFront;
            else
                return oneInFront;
        }

        // Only get here if there isn't a valid location to move to.
        Debug.println("  Darter is blocked.");
        return location();
    }

}
```

SlowFish.java

```
import java.awt.Color;
import java.util.ArrayList;
import java.util.Random;

/**
 *  Marine Biology Simulation:
 *  The SlowFish class represents a fish in the Marine Biology
 *  Simulation that moves very slowly.  It moves so slowly that it only has
 *  a 1 in 5 chance of moving out of its current cell into an adjacent cell
 *  in any given timestep in the simulation.  When it does move beyond its
 *  own cell, its movement behavior is the same as for objects of the
 *  Fish class.
 *
 *  SlowFish objects inherit instance variables and much of
 *  their behavior from the Fish class.
 *
 *  @version 1 July 2002
 **/

public class SlowFish extends Fish
{
    // Instance Variables: Encapsulated data for EACH slow fish
    private double probOfMoving;    // defines likelihood in each timestep


  // constructors

    /** Constructs a slow fish at the specified location in a
     *  given environment.   This slow fish is colored red.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     **/
    public SlowFish(Environment env, Location loc)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, env.randomDirection(), Color.red);

        // Define the likelihood that a slow fish will move in any given
        // timestep.  For now this is the same value for all slow fish.
        probOfMoving = 1.0/5.0;       // 1 in 5 chance in each timestep
    }
    /** Constructs a slow fish at the specified location and direction in a
     *  given environment.   This slow fish is colored red.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     *  @param dir    direction the new fish is facing
     **/
    public SlowFish(Environment env, Location loc, Direction dir)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, dir, Color.red);

        // Define the likelihood that a slow fish will move in any given
        // timestep.  For now this is the same value for all slow fish.
        probOfMoving = 1.0/5.0;       // 1 in 5 chance in each timestep
    }

    /** Constructs a slow fish of the specified color at the specified
     *  location and direction.
     *  (Precondition: parameters are non-null; loc is valid
     *  for env.)
     *  @param env    environment in which fish will live
     *  @param loc    location of the new fish in env
     *  @param dir    direction the new fish is facing
     *  @param col    color of the new fish
     **/
    public SlowFish(Environment env, Location loc, Direction dir, Color col)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, dir, col);

        // Define the likelihood that a slow fish will move in any given
        // timestep.  For now this is the same value for all slow fish.
        probOfMoving = 1.0/5.0;       // 1 in 5 chance in each timestep
    }


  // redefined methods

    /** Creates a new slow fish.
     *  @param loc    location of the new fish
     **/
    protected void generateChild(Location loc)
    {
        // Create new fish, which adds itself to the environment.
        SlowFish child = new SlowFish(environment(), loc,
                                      environment().randomDirection(),
                                      color());
        Debug.println("  New SlowFish created: " + child.toString());
    }
    /** Finds this fish's next location.  A slow fish moves so
     *  slowly that it may not move out of its current cell in
     *  the environment.
     **/
    protected Location nextLocation()
    {
        // There's only a small chance that a slow fish will actually
        // move in any given timestep, defined by probOfMoving.
        Random randNumGen = RandNumGenerator.getInstance();
        if ( randNumGen.nextDouble() < probOfMoving )
            return super.nextLocation();
        else
        {
            Debug.println("SlowFish " + toString() +
                          " not attempting to move.");
            return location();
        }
    }

}
```

Appendix C -- Black Box Classes

This appendix contains summary class documentation for the `Environment` interface and the MBS utility classes covered in Chapters 1 - 4 (`Debug`, `Direction`, `EnvDisplay`, `Locatable`, `Location`, and `RandNumGenerator`) and the `SquareEnvironment` class from Chaper 5.

Environment interface

`public int` **`numRows`**`()`

Returns number of rows in this environment (-1 if the environment is unbounded).

`public int` **`numCols`**`()`

Returns number of columns in this environment (-1 if the environment is unbounded).

`public boolean` **`isValid`**`(Location loc)`

Returns `true` if `loc` is valid in this environment; otherwise returns `false`.

`public int` **`numCellSides`**`()`

Returns the number of sides around each cell.

`public int` **`numAdjacentNeighbors`**`()`

Returns the number of adjacent neighbors around each cell.

`public Direction` **`randomDirection`**`()`

Generates a random direction. The direction returned by `randomDirection` reflects the direction from a cell in the environment to one of its adjacent neighbors.

`public Direction` **`getDirection`**`(Location fromLoc, Location toLoc)`

Returns the direction from one location to another.

`public Location` **`getNeighbor`**`(Location fromLoc, Direction compassDir)`

Returns the adjacent neighboring location of a location in the specified direction (whether valid or invalid).

`public java.util.ArrayList` **`neighborsOf`**`(Location ofLoc)`

Returns the adjacent neighboring locations of a specified location. Only neighbors that are valid locations in the environment will be included.

`public int` **`numObjects`**`()`

Returns the number of objects in this environment.

`public Locatable[]` **`allObjects`**`()`

Returns all the objects in this environment.

`public boolean` **`isEmpty`**`(Location loc)`

Returns `true` if `loc` is a valid location in the context of this environment and is empty; `false` otherwise.

`public Locatable` **`objectAt`**`(Location loc)`

Returns the object at location `loc`; null if `loc` is not in the environment or is empty.

`public void` **`add`**`(Locatable obj)`

Adds a new object to this environment at the location it specifies.
(Precondition: `obj.location()` is a valid empty location.)

`public void` **`remove`**`(Locatable obj)`

Removes the object from this environment.
(Precondition: `obj` is in this environment.)

`public void` **`recordMove`**`(Locatable obj, Location oldLoc)`

Updates this environment to reflect the fact that an object moved.
(Precondition: `obj.location()` is a valid location and there is no other object there.
Postcondition: `obj` is at the appropriate location (`obj.location()`), and either `oldLoc` is equal to `obj.location()` (there was no movement) or `oldLoc` is empty.)

Debug class

`public static boolean` **`isOn`**`()`

Checks whether debugging is on (not necessary when using `Debug.print` and `Debug.println`).

`public static boolean` **`isOff`**`()`

Checks whether debugging is off (not necessary when using `Debug.print` and `Debug.println`).

`public static void` **`turnOn`**`()`

Turns debugging on.

`public static void` **`turnOff`**`()`

Turns debugging off.

`public static void` **`restoreState`**`()`

Restores the previous debugging state. If there is no previous state to restore, `restoreState` turns debugging off.

`public static void` **`print`**`(String message)`

Prints debugging message without appending a newline character at the end. If debugging is turned on, `message` is printed to `System.out` without a newline.

`public static void` **`println`**`(String message)`

Prints debugging message, appending a newline character at the end. If debugging is turned on, `message` is printed to `System.out` followed by a newline.

Direction class

Public class constants: `NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST,`
`WEST, NORTHWEST`

Note: these are class constants of type `Direction` (e.g., `Direction.NORTH`) that represent compass directions in degrees (0°, 45°, 90°, 135°, 180°, 225°, 270°, 315°, respectively)

`public` **`Direction`**`()`

Constructs a default `Direction` object facing North.

`public` **`Direction`**`(int degrees)`

Constructs a `Direction` object - initial compass direction in degrees.

`public` **`Direction`**`(String str)`

Constructs a `Direction` object - compass direction specified as a string, e.g. "North".

`public int` **`inDegrees`**`()`

Returns this direction value in degrees.

`public boolean` **`equals`**`(Object other)`

Returns `true` if `other` has the same number of degrees as this direction; `false` otherwise.

`public Direction` **`toRight`**`()`

Returns the direction that is a quarter turn to the right of this Direction object.

`public Direction` **`toRight`**`(int deg)`

Returns the direction that is `deg` degrees to the right of this `Direction` object.

`public Direction` **`toLeft`**`()`

Returns the direction that is a quarter turn to the left of this `Direction` object.

`public Direction` **`toLeft`**`(int deg)`

Returns the direction that is `deg` degrees to the left of this `Direction` object.

`public Direction` **`reverse`**`()`

Returns the direction that is the reverse of this `Direction` object.

`public String` **`toString`**`()`

Returns a string indicating the direction.

`public static Direction` **`randomDirection`**`()`

Returns a random direction in the range of [0, 360) degrees.

EnvDisplay interface

`public void` **`showEnv`**`()`

Shows the current state of the environment.

Locatable interface

`public Location` **`location`**`()`

Returns the location of this object.

Location class (implements Comparable)

`public` **`Location`**`(int row, int col)`

Constructs a `Location` object.

`public int` **`row`**`()`

Returns the row coordinate of this location.

`public int` **`col`**`()`

Returns the column coordinate of this location.

`public boolean` **`equals`**`(Object other)`

Returns `true` if `other` is at the same row and column as the current location; `false` otherwise.

`public int` **`hashCode`**`()`

Generates a hash code for this location.

`public int` **`compareTo`**`(Object other)`

Compares this location to `other` for ordering. Returns a negative integer, zero, or a positive integer as this location is less than, equal to, or greater than `other`. Locations are ordered in row-major order. (Precondition: `other` is a `Location` object.)

`public String` **`toString`**`()`

Returns a string indicating the row and column of the location in (row, col) format.

RandNumGenerator class

`public static java.util.Random` **`getInstance`**`()`

Returns a random number generator. Always returns the same `Random` object to provide a better sequence of random numbers.

SquareEnvironment Abstract Class (implements Environment) (also found in Appendix D)

`public` **`SquareEnvironment`**`()`

Constructs a `SquareEnvironment` object in which cells have four adjacent neighbors.

`public` **`SquareEnvironment`**`(boolean includeDiagonalNeighbors)`

Constructs a `SquareEnvironment` object in which cells have four or eight adjacent neighbors. If `includeDiagonalNeighbors` is `true`, cells have eight adjacent neighbors -- the immediately adjacent neighbors on all four sides and the four neighbors on the diagonals. If `includeDiagonalNeighbors` is `false`, cells have only the four neighbors they would have in an environment created with the default `SquareEnvironment` constructor.

`public int` **`numCellSides`**`()`

Returns the number of sides around each cell.

`public int` **`numAdjacentNeighbors`**`()`

Returns the number of adjacent neighbors around each cell.

`public Direction` **`randomDirection`**`()`

Returns a random direction in the range of [0, 360) degrees.

`public Direction` **`getDirection`**`(Location fromLoc, Location toLoc)`

Returns the direction from `fromLoc` to `toLoc`.

`public Location` **`getNeighbor`**`(Location fromLoc, Direction compassDir)`

Returns the adjacent neighbor (whether valid or invalid) of a location in the specified direction.

`public java.util.ArrayList` **`neighborsOf`**`(Location ofLoc)`

Returns the adjacent neighbors of a specified location. Only neighbors that are valid locations in the environment will be included.

Appendix D -- Environment Implementations

This appendix contains source code for the Environment implementations covered in Chapter 5: `BoundedEnv` and `UnboundedEnv`. It also contains summary class documentation for the black box `SquareEnvironment` class.

BoundedEnv.java

```
/**
 *  Marine Biology Simulation:
 *  The BoundedEnv class models a bounded, two-dimensional,
 *  grid-like  environment containing locatable objects.  For example,
 *  it could be an environment of fish for a marine biology simulation.
 *
 *  @version 1 July 2002
 **/

public class BoundedEnv extends SquareEnvironment
{
    // Instance Variables: Encapsulated data for each BoundedEnv object
    private Locatable[][] theGrid;  // grid representing the environment
    private int objectCount;        // # of objects in current environment


  // constructors

    /** Constructs an empty BoundedEnv object with the given dimensions.
     *  (Precondition: rows > 0 and cols > 0.)
     *  @param rows        number of rows in BoundedEnv
     *  @param cols        number of columns in BoundedEnv
     **/
    public BoundedEnv(int rows, int cols)
    {
        // Construct and initialize inherited attributes.
        super();

        theGrid = new Locatable[rows][cols];
        objectCount = 0;
    }


  // accessor methods

    /** Returns number of rows in the environment.
     *  @return   the number of rows, or -1 if this environment is unbounded
     **/
    public int numRows()
    {
        return theGrid.length;
    }
```

```java
    /** Returns number of columns in the environment.
     *  @return   the number of columns, or -1 if this environment is unbounded
     **/
    public int numCols()
    {
        // Note: according to the constructor precondition, numRows() > 0, so
        // theGrid[0] is non-null.
        return theGrid[0].length;
    }

    /** Verifies whether a location is valid in this environment.
     *  @param  loc    location to check
     *  @return true if loc is valid; false otherwise
     **/
    public boolean isValid(Location loc)
    {
        if ( loc == null )
            return false;

        return (0 <= loc.row() && loc.row() < numRows()) &&
               (0 <= loc.col() && loc.col() < numCols());
    }

    /** Returns the number of objects in this environment.
     *  @return   the number of objects
     **/
    public int numObjects()
    {
        return objectCount;
    }

    /** Returns all the objects in this environment.
     *  @return    an array of all the environment objects
     **/
    public Locatable[] allObjects()
    {
        Locatable[] theObjects = new Locatable[numObjects()];
        int tempObjectCount = 0;

        // Look at all grid locations.
        for ( int r = 0; r < numRows(); r++ )
        {
            for ( int c = 0; c < numCols(); c++ )
            {
                // If there's an object at this location, put it in the array.
                Locatable obj = theGrid[r][c];
                if ( obj != null )
                {
                    theObjects[tempObjectCount] = obj;
                    tempObjectCount++;
                }
            }
        }

        return theObjects;
    }
    /** Determines whether a specific location in this environment is empty.
     *  @param loc  the location to test
     *  @return     true if loc is a valid location in the context of this
     *              environment and is empty; false otherwise
     **/
    public boolean isEmpty(Location loc)
    {
        return isValid(loc) && objectAt(loc) == null;
    }

    /** Returns the object at a specific location in this environment.
     *  @param loc    the location in which to look
     *  @return       the object at location loc;
     *                null if loc is not in the environment or is empty
     **/
    public Locatable objectAt(Location loc)
    {
        if ( ! isValid(loc) )
            return null;

        return theGrid[loc.row()][loc.col()];
    }

    /** Creates a single string representing all the objects in this
     *  environment (not necessarily in any particular order).
     *  @return    a string indicating all the objects in this environment
     **/
    public String toString()
    {
        Locatable[] theObjects = allObjects();
        String s = "Environment contains " + numObjects() + " objects: ";
        for ( int index = 0; index < theObjects.length; index++ )
            s += theObjects[index].toString() + " ";
        return s;
    }


  // modifier methods

    /** Adds a new object to this environment at the location it specifies.
     *  (Precondition: obj.location() is a valid empty location.)
     *  @param obj the new object to be added
     *  @throws    IllegalArgumentException if the precondition is not met
     **/
    public void add(Locatable obj)
    {
        // Check precondition.  Location should be empty.
        Location loc = obj.location();
        if ( ! isEmpty(loc) )
            throw new IllegalArgumentException("Location " + loc +
                                        " is not a valid empty location");

        // Add object to the environment.
        theGrid[loc.row()][loc.col()] = obj;
        objectCount++;
    }
    /** Removes the object from this environment.
     *  (Precondition: obj is in this environment.)
     *  @param obj     the object to be removed
     *  @throws    IllegalArgumentException if the precondition is not met
     **/
    public void remove(Locatable obj)
    {
        // Make sure that the object is there to remove.
        Location loc = obj.location();
        if ( objectAt(loc) != obj )
            throw new IllegalArgumentException("Cannot remove " +
                                               obj + "; not there");

        // Remove the object from the grid.
        theGrid[loc.row()][loc.col()] = null;
        objectCount--;
    }

    /** Updates this environment to reflect the fact that an object moved.
     *  (Precondition: obj.location() is a valid location and there is no
     *  other object there.
     *  Postcondition: obj is at the appropriate location (obj.location()),
     *  and either oldLoc is equal to obj.location() (there was no movement)
     *  or oldLoc is empty.)
     *  @param obj       the object that moved
     *  @param oldLoc    the previous location of obj
     *  @throws    IllegalArgumentException if the precondition is not met
     **/
    public void recordMove(Locatable obj, Location oldLoc)
    {
        // Simplest case: There was no movement.
        Location newLoc = obj.location();
        if ( newLoc.equals(oldLoc) )
            return;

        // Otherwise, oldLoc should contain the object that is
        //   moving and the new location should be empty.
        Locatable foundObject = objectAt(oldLoc);
        if ( ! (foundObject == obj && isEmpty(newLoc)) )
            throw new IllegalArgumentException("Precondition violation moving "
                + obj + " from " + oldLoc);

        // Move the object to the proper location in the grid.
        theGrid[newLoc.row()][newLoc.col()] = obj;
        theGrid[oldLoc.row()][oldLoc.col()] = null;
    }

}
```

UnboundedEnv.java

```
import java.util.ArrayList;

/**
 *  Marine Biology Simulation:
 *  The UnboundedEnv class models an unbounded, two-dimensional,
 *  grid-like environment containing locatable objects.  For example, it
 *  could be an environment of fish for a marine biology simulation.
 *
 *  Modification History:
 *  - Created to support multiple environment representations: this class
 *    represents a second implementation of the Environment interface.
 *
 *  @version 1 July 2002
 **/

public class UnboundedEnv extends SquareEnvironment
{
    // Instance Variables: Encapsulated data for each UnboundedEnv object
    private ArrayList objectList;   // list of Locatable objects in environment


  // constructors

    /** Constructs an empty UnboundedEnv object.
     **/
    public UnboundedEnv()
    {
        // Construct and initialize inherited attributes.
        super();

        objectList = new ArrayList();
    }


  // accessor methods

    /** Returns number of rows in this environment.
     *  @return   the number of rows, or -1 if the environment is unbounded
     **/
    public int numRows()
    {
        return -1;
    }

    /** Returns number of columns in this environment.
     *  @return   the number of columns, or -1 if the environment is unbounded
     **/
    public int numCols()
    {
        return -1;
    }
    /** Verifies whether a location is valid in this environment.
     *  @param  loc    location to check
     *  @return true if loc is valid;
     *          false otherwise
     **/
    public boolean isValid(Location loc)
    {
        // All non-null locations are valid in an unbounded environment.
        return loc != null;
    }

    /** Returns the number of objects in this environment.
     *  @return   the number of objects
     **/
    public int numObjects()
    {
        return objectList.size();
    }

    /** Returns all the objects in this environment.
     *  @return    an array of all the environment objects
     **/
    public Locatable[] allObjects()
    {
        Locatable[] objectArray = new Locatable[objectList.size()];

        // Put all the environment objects in the list.
        for ( int index = 0; index < objectList.size(); index++ )
        {
            objectArray[index] = (Locatable) objectList.get(index);
        }

        return objectArray;
    }

    /** Determines whether a specific location in this environment is empty.
     *  @param loc  the location to test
     *  @return     true if loc is a valid location in the context of this
     *              environment and is empty; false otherwise
     **/
    public boolean isEmpty(Location loc)
    {
        return (objectAt(loc) == null);
    }

    /** Returns the object at a specific location in this environment.
     *  @param loc    the location in which to look
     *  @return       the object at location loc; null if loc is empty
     **/
    public Locatable objectAt(Location loc)
    {
        int index = indexOf(loc);
        if ( index == -1 )
            return null;

        return (Locatable) objectList.get(index);
    }
    /** Creates a single string representing all the objects in this
     *  environment (not necessarily in any particular order).
     *  @return    a string indicating all the objects in this environment
     **/
    public String toString()
    {
        Locatable[] theObjects = allObjects();
        String s = "Environment contains " + numObjects() + " objects: ";
        for ( int index = 0; index < theObjects.length; index++ )
            s += theObjects[index].toString() + " ";
        return s;
    }


  // modifier methods

    /** Adds a new object to this environment at the location it specifies.
     *  (Precondition: obj.location() is a valid empty location.)
     *  @param obj the new object to be added
     *  @throws    IllegalArgumentException if the precondition is not met
     **/
    public void add(Locatable obj)
    {
        // Check precondition.  Location should be empty.
        Location loc = obj.location();
        if ( ! isEmpty(loc) )
            throw new IllegalArgumentException("Location " + loc +
                                        " is not a valid empty location");

        // Add object to the environment.
        objectList.add(obj);
    }

    /** Removes the object from this environment.
     *  (Precondition: obj is in this environment.)
     *  @param obj     the object to be removed
     *  @throws    IllegalArgumentException if the precondition is not met
     **/
    public void remove(Locatable obj)
    {
        // Find the index of the object to remove.
        int index = indexOf(obj.location());
        if ( index == -1 )
            throw new IllegalArgumentException("Cannot remove " +
                                               obj + "; not there");

        // Remove the object.
        objectList.remove(index);
    }
    /** Updates this environment to reflect the fact that an object moved.
     *  (Precondition: obj.location() is a valid location and there is no
     *  other object there.
     *  Postcondition: obj is at the appropriate location (obj.location()),
     *  and either oldLoc is equal to obj.location() (there was no movement) or
     *  oldLoc is empty.)
     *  @param obj       the object that moved
     *  @param oldLoc    the previous location of obj
     *  @throws    IllegalArgumentException if the precondition is not met
     **/
    public void recordMove(Locatable obj, Location oldLoc)
    {
        int objectsAtOldLoc = 0;
        int objectsAtNewLoc = 0;

        // Look through the list to find how many objects are at old
        // and new locations.
        Location newLoc = obj.location();
        for ( int index = 0; index < objectList.size(); index++ )
        {
            Locatable thisObj = (Locatable) objectList.get(index);
            if ( thisObj.location().equals(oldLoc) )
                objectsAtOldLoc++;
            if ( thisObj.location().equals(newLoc) )
                objectsAtNewLoc++;
        }

        // There should be one object at newLoc.  If oldLoc equals
        // newLoc, there should be one at oldLoc; otherwise, there
        // should be none.
        if ( ! ( objectsAtNewLoc == 1 &&
                 ( oldLoc.equals(newLoc) || objectsAtOldLoc == 0 ) ) )
        {
            throw new IllegalArgumentException("Precondition violation moving "
                + obj + " from " + oldLoc);
        }
    }
```

```
  // internal helper method

    /** Get the index of the object at the specified location.
     *  @param loc    the location in which to look
     *  @return       the index of the object at location loc
     *                if there is one; -1 otherwise
     **/
    protected int indexOf(Location loc)
    {
        // Look through the list to find the object at the given location.
        for ( int index = 0; index < objectList.size(); index++ )
        {
            Locatable obj = (Locatable) objectList.get(index);
            if ( obj.location().equals(loc) )
            {
                // Found the object -- return its index.
                return index;
            }
        }

        // No such object found.
        return -1;
    }

}
```

SquareEnvironment Abstract Class (implements Environment) (also found in Appendix C)

`public` **`SquareEnvironment`**`()`

Constructs a `SquareEnvironment` object in which cells have four adjacent neighbors.

`public` **`SquareEnvironment`**`(boolean includeDiagonalNeighbors)`

Constructs a `SquareEnvironment` object in which cells have four or eight adjacent neighbors. If `includeDiagonalNeighbors` is `true`, cells have eight adjacent neighbors -- the immediately adjacent neighbors on all four sides and the four neighbors on the diagonals. If `includeDiagonalNeighbors` is `false`, cells have only the four neighbors they would have in an environment created with the default `SquareEnvironment` constructor.

`public int` **`numCellSides`**`()`

Returns the number of sides around each cell.

`public int` **`numAdjacentNeighbors`**`()`

Returns the number of adjacent neighbors around each cell.

`public Direction` **`randomDirection`**`()`

Returns a random direction in the range of [0, 360) degrees.

`public Direction` **`getDirection`**`(Location fromLoc, Location toLoc)`

Returns the direction from `fromLoc` to `toLoc`.

`public Location` **`getNeighbor`**`(Location fromLoc, Direction compassDir)`

Returns the adjacent neighbor (whether valid or invalid) of a location in the specified direction.

`public java.util.ArrayList` **`neighborsOf`**`(Location ofLoc)`

Returns the adjacent neighbors of a specified location. Only neighbors that are valid locations in the environment will be included.

Appendix F -- JMBS Quick Reference

Quick Reference for Core Classes and Interfaces

Simulation Class

```
public Simulation(Environment env, EnvDisplay display)
public void step()
```

Environment Interface

```
public int numRows()
public int numCols()

public boolean isValid(Location loc)
public int numCellSides()
public int numAdjacentNeighbors()
public Direction randomDirection()
public Direction getDirection(Location fromLoc, Location toLoc)
public Location getNeighbor(Location fromLoc, Direction compassDir)
public ArrayList neighborsOf(Location ofLoc)

public int numObjects()
public Locatable[] allObjects()
public boolean isEmpty(Location loc)
public Locatable objectAt(Location loc)

public void add(Locatable obj)
public void remove(Locatable obj)
public void recordMove(Locatable obj, Location oldLoc)
```

SquareEnvironment Abstract Class (implements Environment)

```
public SquareEnvironment()
public SquareEnvironment(boolean includeDiagonalNeighbors)

public int numCellSides()
public int numAdjacentNeighbors()
public Direction randomDirection()
public Direction getDirection(Location fromLoc, Location toLoc)
public Location getNeighbor(Location fromLoc, Direction compassDir)
public ArrayList neighborsOf(Location ofLoc)
```

BoundedEnv Class(extends SquareEnvironment)

```
public BoundedEnv(int rows, int cols)

public int numRows()
public int numCols()

public boolean isValid(Location loc)

public int numObjects()
public Locatable[] allObjects()
public boolean isEmpty(Location loc)
public Locatable objectAt(Location loc)
public String toString()

public void add(Locatable obj)
public void remove(Locatable obj)
public void recordMove(Locatable obj, Location oldLoc)
```

UnboundedEnv Class(extends SquareEnvironment)

```
public UnboundedEnv()

public int numRows()
public int numCols()

public boolean isValid(Location loc)

public int numObjects()
public Locatable[] allObjects()
public boolean isEmpty(Location loc)
public Locatable objectAt(Location loc)
public String toString()

public void add(Locatable obj)
public void remove(Locatable obj)
public void recordMove(Locatable obj, Location oldLoc)

protected int indexOf(Location loc)
```

Quick Reference for Fish Class

Fish Class (implements Locatable)

```
public Fish(Environment env, Location loc)
public Fish(Environment env, Location loc, Direction dir)
public Fish(Environment env, Location loc, Direction dir, Color col)
private void initialize(Environment env, Location loc, Direction dir,
                        Color col)
protected Color randomColor()

public int id()
public Environment environment()
public Color color()
public Location location()
public Direction direction()
public boolean isInEnv()
public String toString()

public void act()

protected boolean breed()
protected void generateChild(Location loc)
protected void move()
protected Location nextLocation()
protected ArrayList emptyNeighbors()
protected void changeLocation(Location newLoc)
protected void changeDirection(Direction newDir)
protected void die()
```

Quick Reference for Specialized Fish Subclasses

DarterFish Class (extends Fish)

```
public DarterFish(Environment env, Location loc)
public DarterFish(Environment env, Location loc, Direction dir)
public DarterFish(Environment env, Location loc, Direction dir, Color col)

protected void generateChild(Location loc)
protected void move()
protected Location nextLocation()
```

SlowFish Class (extends Fish)

```
public SlowFish(Environment env, Location loc)
public SlowFish(Environment env, Location loc, Direction dir)
public SlowFish(Environment env, Location loc, Direction dir, Color col)

protected void generateChild(Location loc)
protected Location nextLocation()
```

Quick Reference for Utility Classes and Interfaces (public constants, constructors, and methods only)

Case Study Utility Classes and Interfaces

Debug Class

```
static boolean isOn()
static boolean isOff()
static void turnOn()
static void turnOff()
static void restoreState()
static void print(String message)
static void println(String message)
```

Direction Class

```
NORTH, EAST, SOUTH, WEST, NORTHEAST,
NORTHWEST, SOUTHEAST, SOUTHWEST

Direction()
Direction(int degrees)
Direction(String str)
int inDegrees()
boolean equals(Oject other)
Direction toRight()
Direction toRight(int degrees)
Direction toLeft()
Direction toLeft(int degrees)
Direction reverse()
String toString()
static Direction randomDirection()
```

EnvDisplay Interface

```
void showEnv()
```

Locatable Interface

```
Location location()
```

Location Class
(implements Comparable)

```
Location(int row, int col)
int row()
int col()
boolean equals(Object other)
int compareTo(Object other)
String toString()
int hashCode()
```

RandNumGenerator Class

```
static Random getInstance()
```

Java Library Utility Classes

`java.util.ArrayList` Class(Partial)

```
boolean add(Object obj)
void add(int index, Object obj)
Object get(int index)
Object remove(int index)
boolean remove(Object obj)
Object set(int index, Object obj)
int size()
```

`java.awt.Color` Class(Partial)

```
black, blue, cyan, gray, green,
magenta, orange, pink, red,
white, yellow

Color(int r, int g, int b)
```

`java.util.Random` Class(Partial)

```
int nextInt(int n)
double nextDouble()
```

Appendix G -- Index for Source Code

This appendix provides an index for the Java source code found in Appendix B and Appendix D. Methods are grouped in alternating gray and white blocks based on the page on which they are listed.

Simulation.java

`Simulation(Environment env, EnvDisplay display)` B1
`step()` B1

Fish.java

`Fish(Environment env, Location loc)` B2
`Fish(Environment env, Location loc, Direction dir)` B3
`Fish(Environment env, Location loc, Direction dir, Color col)` B3
`initialize(Environment env, Location loc, Direction dir, Color col)` B3
`randomColor()` B4
`id()` B4
`environment()` B4
`color()` B4
`location()` B4
`direction()` B4
`isInEnv()` B5
`toString()` B5
`act()` B5
`breed()` B6
`generateChild(Location loc)` B6
`move()` B7
`nextLocation()` B7
`emptyNeighbors()` B8
`changeLocation(Location newLoc)` B8
`changeDirection(Direction newDir)` B8
`die()` B8

DarterFish.java

`DarterFish(Environment env, Location loc)` B9
`DarterFish(Environment env, Location loc, Direction dir)` B9
`DarterFish(Environment env, Location loc, Direction dir, Color col)` B10
`generateChild(Location loc)` B10
`move()` B10
`nextLocation()` B11

SlowFish.java

SlowFish(Environment env, Location loc)	B12
SlowFish(Environment env, Location loc, Direction dir)	B13
SlowFish(Environment env, Location loc, Direction dir, Color col)	B13
generateChild(Location loc)	B13
nextLocation()	B14

BoundedEnv.java

BoundedEnv(int rows, int cols)	D1
numRows()	D1
numCols()	D2
isValid(Location loc)	D2
numObjects()	D2
allObjects()	D2
isEmpty(Location loc)	D3
objectAt(Location loc)	D3
toString()	D3
add(Locatable obj)	D3
remove(Locatable obj)	D4
recordMove(Locatable obj, Location oldLoc)	D4

UnboundedEnv.java

UnboundedEnv()	D5
numRows()	D5
numCols()	D5
isValid(Location loc)	D6
numObjects()	D6
allObjects()	D6
isEmpty(Location loc)	D6
objectAt(Location loc)	D6
toString()	D7
add(Locatable obj)	D7
remove(Locatable obj)	D7
recordMove(Locatable obj, Location oldLoc)	D8
indexOf(Location loc)	D9