**Marine Biology
Simulation
Case Study**

COMPUTER SCIENCE

CS

The College Board is a national nonprofit membership association whose mission is to prepare, inspire, and connect students to college and opportunity. Founded in 1900, the association is composed of more than 4,200 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three million students and their parents, 22,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of equity and excellence, and that commitment is embodied in all of its programs, services, activities, and concerns.

The College Board and the Advanced Placement Program encourage teachers, AP Coordinators, and school administrators to make equitable access a guiding principle for their AP programs. The College Board is committed to the principle that all students deserve an opportunity to participate in rigorous and academically challenging courses and programs. All students who are willing to accept the challenge of a rigorous academic curriculum should be considered for admission to AP courses. The Board encourages the elimination of barriers that restrict access to AP courses for students from ethnic, racial, and socioeconomic groups that have been traditionally underrepresented in the AP Program. Schools should make every effort to ensure that their AP classes reflect the diversity of their student population.

For more information about equity and access in principle and practice, contact the National Office in New York.

Advanced Placement Program®
Computer Science

Marine Biology Simulation
Case Study

AP Computer Science
Marine Biology Simulation Case Study

Contents

Chapter 5

Marine Biology Simulation
Case Study

Introduction

This document is the report of a summer job experience by a computer science student named Pat.

Last summer I decided to get a programming job to further develop the skills I had learned in class. I was hired to modify a simulation program used by some marine biologists. I was referred to Jamie, an experienced software developer. Jamie described the existing program to me, and I continued from there. This is my report of what I learned and the code that I developed.

A *simulation* is a model of a real system. Scientists build simulations to better understand the systems they are studying, to observe and predict their behavior, or to perform experiments on them. Many real systems are difficult or impossible to observe and control, much less experiment with. It is easy, though, to run a simulation program repeatedly and to modify it to explore the effect of changes to the model.

I worked on a simulation of fish moving in a relatively small body of water, such as a lake. The modifications I added made the simulation more interesting and allowed the biologists to study more complex behavior. For example, one change I made allowed the biologists to track the fish population as fish breed and die. Another change allowed them to study what happens when there are several kinds of fish in the environment, with different patterns of movement.

This report is divided into five chapters. The first chapter describes what the program did before I began working on it. The next chapter contains Jamie's explanation of the code — how the program was implemented. The third chapter describes the set of changes I made to model fish breeding and dying. The fourth chapter describes changes that allowed the biologists to study fish with specialized patterns of movement.

After the summer was over, I continued to work for the biologists part-time. The last chapter of this report describes further changes I made that allowed the biologists to study fish in both bounded and unbounded environments.

Note about the AP Computer Science Exams:

Computer Science A: Students are expected to be familiar with the material in Chapters 1 – 4 of this case study, including the `Fish` and `Simulation` classes presented in Chapter 2, the modifications to the `Fish` class presented in Chapter 3, and the `DarterFish` and `SlowFish` classes presented in Chapter 4. Students should also be familiar with the class documentation for several other classes and interfaces from the case study (`Debug`, `Direction`, `EnvDisplay`, `Environment`, `Locatable`, `Location`, and `RandNumGenerator`) and with the subset of methods that are used in the case study from the Standard Java `ArrayList`, `Color`, and `Random` classes. A Quick Reference for these appears at the end of Chapter 2. Students taking the A Exam are not responsible for the material in Chapter 5.

Computer Science AB: Students should be familiar with the classes and class documentation listed above and the `Environment` interface and the `BoundedEnv` and `UnboundedEnv` classes presented in Chapter 5. Students should also be familiar with the class documentation for the `SquareEnvironment` abstract class.

Marine Biology Simulation Case Study

Chapter 1

Experimenting with the Marine Biology Simulation Program

Last summer I was hired to modify a simulation program used by some marine biologists. On the first day, I met with my boss who told me a little bit about the program I would be working on. It was designed to help the marine biologists study fish movement in a bounded environment, such as a lake or a bay. It was working fine for them, but they wanted to make some improvements and add some new functionality.

Jamie, an experienced programmer on the team, would be available the following day to give me a "guided tour" of the actual program code. In the meantime I could experiment with the program and start figuring out what it did. I was given some instructions for running it and told where to find several data files to use with it.

[Educational prerequisites for this chapter: The first two sections of the chapter, Experimenting with the Simulation and Looking at the Data Files, have no prerequisites. The last section, Sneaking a Peek at Some Code, assumes familiarity with constructing objects, invoking methods (including the `obj.method()` calling syntax), and `for` loops.]

Experimenting with the Simulation

The first thing I did was to determine how to compile and run the existing program.

Exercise Set 1:

1. Run the marine biology simulation program with two different data files, `fish.dat` and `manyFish.dat`, to see how the simulation works. You will need to find out from your teacher how to compile and run the program on your school's computers. You will also need to know where to find the `fish.dat` and `manyFish.dat` data files. These files describe the initial configuration of the fish in the environment. *[Note: In the `JavaMBS.zip` distribution file, the main method to run the Marine Biology Simulation is in the `MBSGUI` class in the `Code` folder. Select "Open environment file . . ." in the `file` menu to find and open an initial configuration file. The configuration files are in the `DataFiles` folder.]*

2. Run the marine biology simulation program twice with the same data file. Is the behavior of the fish in the environment the same both times?

Analysis Question Set 1:

1. How does the program appear to model the body of water (the "bounded environment")? What do the grid lines represent?

2. Where can fish be in the model? Can there be more than one fish in the same place at the same time?

3. Do all fish face the same direction? Does a fish's direction appear to matter?

4. Pick one fish and watch it for several steps. Does it move in every step? How far does it move? Does it always move in the same direction? Can it move in any direction? Is it more likely to move in one direction over another?

I ran the program several times with the different files and watched the behavior. I then made some notes and tried to deduce what the "rules" were in the program for fish movement. I found it difficult, though, to keep track of everything that was happening, and I wasn't sure that all my original conclusions were correct. I decided I needed to develop a strategy, to be more scientific in my tests. I started with the file called `onefish.dat`. This file, as its name implied, seemed to model a single fish in a small (7 x 5) environment. I created a table showing where the fish was at each timestep, which direction it was facing, and where it moved.

Exercise Set 2:

1. Run the marine biology simulation program with `onefish.dat` for five timesteps, keeping notes.

Timestep	Fish's Location	Fish's Direction	Did it Move?	In what Direction?	New Location	New Direction
1						
2						
3						
4						
5						

   How often does the fish move forward? Right? Left? Backward?

2. Compare your results with a classmate. Did your classmate have the same results? What conclusions can you draw about fish movement in a single run of the program and about different runs of the program? How confident are you that you have enough data?

Conclusions:

I made several conclusions based on my tests.

- The fish always moved in every timestep, but it didn't always move the same way.

- It always moved one space.

- It did not always move in the direction it was facing.

- Sometimes it moved forward; sometimes it moved to the side.

- I never saw it move backward in that test run.

- When it moved to the side it always changed direction.

- After every move, its new direction was always the same as the direction it moved.

- Every time I ran the program I saw different results.

Analysis Question Set 2:

1. Did your test results show the same thing? What did others in your class discover? Are five timesteps enough to come to any conclusions?

2. What if you run the program with `onefish.dat` for 10 timesteps? 20 timesteps?

Timestep	Fish's Location	Fish's Direction	Did it Move?	In what Direction?	New Location	New Direction
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						

Looking at the Data Files

I wanted to see if I could read the data files that were providing the initial configuration of fish in the environment. I decided to start with `onefish.dat`. The contents of the file are:

```
bounded 7 5
Fish 3 2 North
```

I guessed that the "bounded" referred to the "bounded environment" my boss had mentioned, and that the 7 and 5 in the first line specified the dimensions of environment, since I had already noticed that its size was 7 x 5. I thought the second line probably determined the initial location and direction of the fish, especially since I noticed that every time I ran the program with this file the fish started out facing the top of the screen.

Analysis Question Set 3:

1. If the numbers "3 2" in `onefish.dat` refer to the initial location of the fish in the environment, how are locations in an environment numbered?

Exercise Set 3:

1. Look over another of the data files and then run the program using it. Are the size of the environment and the initial locations and directions of the fish what you expected based on the analysis of `onefish.dat`?

2. Create a data file of your own and then run the program using it. Is the initial configuration of the environment what you would have expected?

Sneaking a Peek at Some Code

As I was compiling and running the program, I noticed two files called
SimpleMBSDemo1.java and SimpleMBSDemo2.java. I decided to run the first
one and discovered that it was a simpler version of the simulation program that
always ran for 15 timesteps. It wasn't as interesting as the full version of the
program, but the filename said "Simple" so I thought that I might be able to read
it and understand it even before meeting with Jamie. I decided to look at the code.

```java
public class SimpleMBSDemo1
{
    // Specify number of rows and columns in environment.
    private static final int ENV_ROWS = 10;   // rows in environment
    private static final int ENV_COLS = 10;   // columns in environment

    // Specify how many timesteps to run the simulation.
    private static final int NUM_STEPS = 15;  // number of timesteps

    // Specify the time delay for each step
    private static final int DELAY = 1000;    // delay in milliseconds

    /** Start the Marine Biology Simulation program.
     *  The String arguments (args) are not used in this application.
     **/
    public static void main(String[] args)
    {
        // Construct an empty environment and several fish in the
        // context of that environment.
        BoundedEnv env = new BoundedEnv(ENV_ROWS, ENV_COLS);
        Fish f1 = new Fish(env, new Location(2, 2));
        Fish f2 = new Fish(env, new Location(2, 3));
        Fish f3 = new Fish(env, new Location(5, 8));

        // Construct an object that knows how to draw the environment
        // with a delay; display the initial configuration of the
        // environment.
        SimpleMBSDisplay display = new SimpleMBSDisplay(env, DELAY);
        display.showEnv();

        // Run the simulation for the specified number of steps.
        for ( int i = 0; i < NUM_STEPS; i++ )
        {
            f1.act();
            f2.act();
            f3.act();
            display.showEnv();
        }
    }
}
```

I noticed that the first few lines create named constants (`ENV_ROWS`, `ENV_COLS`, `NUM_STEPS`, and `DELAY`) that can be used later in the class. However, it was the `main` method that was most interesting to me, because I knew that this is where the program execution starts. I saw that the first thing it does is to create a `BoundedEnv` object. I figured that must represent the body of water — the "bounded environment". The program then creates three `Fish` objects, passing each one the environment and a `Location` object. I couldn't remember what the initial positions of the fish had been when I ran the program, but it seemed reasonable that these `Location` objects were determining the initial positions. (I decided to rerun the program to verify this.) I wasn't as sure about why the new fish were being passed the `Environment` object, and made a note to ask Jamie when we met.

Next, the program creates an object called `display` of a class called `SimpleMBSDisplay`. This seemed to be the object that was displaying the environment on the screen. I noticed that the new `SimpleMBSDisplay` object is also passed the environment when it is constructed, but this made sense to me if its job is to display the environment. I assumed that the call

```
display.showEnv();
```

actually does the displaying. The `SimpleMBSDisplay` constructor is also passed the `DELAY` constant which, according to the comment, seemed to be the time delay between timesteps that gives the user time to see what is happening.

Finally, the main method contains a loop in which the three fish are told to "act" and then `display` is asked to show the environment again. I wondered, does "act" mean "move"?

Now I was curious about what the difference was between `SimpleMBSDemo1.java` and `SimpleMBSDemo2.java`. I ran `SimpleMBSDemo2.java`, but it seemed to do pretty much what `SimpleMBSDemo1.java` had done. (It was a little difficult to know for sure, since the behavior of every run of the program was different anyway.) So I decided to compare the code in the two files. The only differences are at the end of the two programs.

From SimpleMBSDemo1:
```
// Construct an object that knows how to draw the environment with
// a delay; display the initial configuration of the environment.
SimpleMBSDisplay display = new SimpleMBSDisplay(env, DELAY);
display.showEnv();

// Run the simulation for the specified number of steps.
for ( int i = 0; i < NUM_STEPS; i++ )
{
    f1.act();
    f2.act();
    f3.act();
    display.showEnv();
}
```

From SimpleMBSDemo2:
```
// Construct an object that knows how to draw the environment with
// a delay.
SimpleMBSDisplay display = new SimpleMBSDisplay(env, DELAY);

// Construct the simulation object. It needs to have the environment
// and the object that can draw the environment.
Simulation sim = new Simulation(env, display);

// Run the simulation for the specified number of steps.
for ( int i = 0; i < NUM_STEPS; i++ )
{
    sim.step();
}
```

I found the code for `SimpleMBSDemo2` harder to understand, and I was glad that I had looked at `SimpleMBSDemo1` first. First of all, `SimpleMBSDemo2` constructs the display object, `display`, but never asks it to show the environment. Secondly, even though `SimpleMBSDemo2` constructs three fish just as `SimpleMBSDemo1` did, it never asks them to act. Instead, it constructs an object of a new `Simulation` class, and in the loop asks it to "step." I noticed that the program passes the environment and the display object to the `Simulation` constructor, so I assumed that the `Simulation` object must ask the display object to show the environment when the `Simulation` object is constructed. I also assumed that its `step` method must ask the fish to act, although I wasn't sure how since the program never passes the fish to the simulation.

Later in the summer I created the picture below. It illustrates the behavior of a *driver* in the marine biology simulation, such as the main method in SimpleMBSDemo2. (A driver is a main method, applet, or graphical user interface that "drives" the rest of the program.)

The Driver

General Outline:

Driver
A. constructs an Environment object
B. constructs Fish objects, telling each the environment in which they live and their initial location
   i. Each Fish object adds itself to the environment as it is constructed
C. constructs an object to display the environment
D. constructs a Simulation object to run the simulation
E. repeatedly calls step in Simulation to execute each timestep

At this point I felt that I had learned about as much about the marine biology simulation program as I could without Jamie's "guided tour." I had a basic idea about how the program was working and I was ready to start exploring some of the code (like the mysterious Simulation class!) in more detail.

Marine Biology Simulation Case Study

Chapter 2

Guided Tour of the Marine Biology Simulation Implementation

Before making any modifications to the marine biology simulation, I needed to understand more thoroughly how the existing program worked. I had experimented, running it in several different ways and with different initial data, but I didn't know much about the actual implementation. My next step was to meet with Jamie, an experienced programmer who had known the original program developer and was familiar with the code. This is Jamie's "guided tour" of the simulation program.

[Educational prerequisites for this chapter: Students should be familiar with reading class documentation, constructing objects and invoking methods, the format of a class implementation (instance variables and methods), and the basic flow control constructs (conditions and loops). Students should also be familiar with 1-D Java arrays. Topics covered in this chapter include: object interaction, class documentation, class implementation, instance and class variables, interfaces, random numbers, the `this` *keyword,* `equals` *vs* `==`*, black box and code-based testing, and analyzing alternative designs. The chapter introduces and uses the following standard Java classes:* `ArrayList`*,* `Random`*, and* `Color`*. It does not, however, cover all of the* `ArrayList` *methods with which students are expected to be familiar. It touches on the difference between arrays and the* `ArrayList` *class, but does not go deeply into the difference between these basic data structures. The chapter also briefly introduces the* `protected` *keyword; chapter 4 covers this concept more thoroughly.]*

The Big Picture

The marine biology simulation case study is a simulation program designed to help marine biologists study fish movement in a small, bounded environment such as a lake or bay. For modeling purposes, the biologists think of the environment as a rectangular grid, with fish moving from cell to cell in the grid. Each cell contains zero or one fish.

	0	1	2	3	4	5	6	7	8
0				🐟					
1									
2			🐟						
3			🐟		🐟				
4									🐟

A simulation is often represented as a series of repeated *timesteps*. In this case, in each timestep of the simulation every fish has an opportunity to move to one of its immediately adjoining neighbors. Fish in this simulation only move forward and to the side; they never move backward. Thus, the fish in row 3 and column 2 (also known as location (3, 2)) in the illustration above has two neighboring locations to which it could move, the one in front of it and the one to its right (locations (3, 3) and (4, 2)). The fish in location (4, 8) has one location to which it could move, the one to its left.

One of the purposes of this simulation is to help the biologists understand how fish movement is affected by the size and density of the fish population. To do this, they run the simulation many times with different initial fish configurations. The program reads a file that specifies the size of the environment, the number of fish, and their starting locations. By changing the file read in by the program, the biologists can run the simulation with different initial conditions and see what happens.

What classes are necessary?

To model *fish* swimming in a *bounded environment*, the program has `Fish` objects and an `Environment` object. The purpose of the program is to *simulate* fish moving in the environment, so the program also has a `Simulation` object. There are a number of other useful, but less important classes, in the program, which I'll call the "utility classes." We'll get to those later, but these three are the core classes of the marine biology simulation.

What do objects of the core classes do?

The Simulation class has a step method that executes a single timestep in the simulation, in which each fish has the opportunity to move to an adjoining location in the grid. This leads to a number of design questions. Who is responsible for keeping track of the fish in the environment? Who is responsible for actually moving a fish? Who is responsible for knowing what behavior (moving, aging, breeding, dying, etc.) is required as part of the simulation? The original programmer considered several options and decided on the following design.

- The Fish class encapsulates basic information about a fish (color, location, and so on) and basic fish behavior. For now, this behavior is just moving to an adjacent cell.

- The Environment class models the rectangular grid, keeping track of the fish in the grid. It does not care what their behavior is, except when that behavior changes the number of fish in the grid or their locations. In fact, eventually the original programmer realized that there is nothing about the behavior of the environment that depends on the objects in it being fish. They could be band members marching on a field, physics particles moving in a small space, or any other object that you might want to model in a bounded, grid-like environment. So, the Environment doesn't refer to fish at all; instead, it models a grid of generic objects.

- The Simulation class represents the behavior that happens in every timestep of the simulation, fish movement in this case. Of course, the fish have to know how to do whatever it is that the simulation wants them to do, so the Simulation and the Fish classes share the responsibility for knowing what behavior is required in the simulation.

What do the core classes look like?

The heart of this simulation is the step method in the Simulation class. To run the program, though, something needs to repeatedly call step. This something is called a *driver*. It could be a Java applet, for example, or an application with a graphical user interface. It could even be a very simple main method, such as the one in SimpleMBSDemo2, that constructs a Simulation object (let's call it sim, for example) and then calls step in a loop like the one below.

```
for ( int i = 0; i < NUM_STEPS; i++ )
    sim.step();
```

The `Simulation` Class

The `Simulation` class is actually quite simple. It has two methods: a constructor and the `step` method. The class, with most of its comments and debugging statements removed, is shown below.

```
public class Simulation
{
    private Environment theEnv;
    private EnvDisplay theDisplay;

    public Simulation(Environment env, EnvDisplay display)
    {
        theEnv = env;
        theDisplay = display;
        theDisplay.showEnv();
    }

    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();
        }
        theDisplay.showEnv();
        Debug.println(theEnv.toString());
        Debug.println("——— End of Timestep ———");
    }
}
```

The constructor receives two parameters, an `Environment` object and an `EnvDisplay` object. `EnvDisplay` is one of the utility "classes" I mentioned earlier. It is actually an *interface* instead of a class. An interface is like a class, but it just specifies *what* methods should be implemented without actually implementing them. A program with an interface needs to also have at least one class that *implements* the interface, in other words, that provides implementations for all the methods that the interface specifies. In this case, the `EnvDisplay` interface specifies a single method, `showEnv`. The `SimpleMBSDisplay` class in `SimpleMBSDemo2` is one class that implements the `EnvDisplay` interface, so a `SimpleMBSDisplay` object can be passed as the `EnvDisplay` parameter to the `Simulation` constructor. There could be other classes that display the environment in other ways (such as a text-based display or a display that uses pictures rather than graphical shapes to represent the fish); as long as they implement the `EnvDisplay` interface, the `step` method will work just fine.‡

Because the `Simulation` constructor needs the environment and display, the driver (applet, main method, or user interface) that constructs the `Simulation` object must construct the environment and display objects first. The `Simulation` constructor then stores the two parameters so the `step` method can use them later and asks the `EnvDisplay` object to display the initial environment configuration.

The `step` method represents what happens in a single simulation timestep. First, it asks the environment for a list of all its objects (fish), which the environment returns as an array of `Locatable` objects. `Locatable` is another interface defined within the marine biology simulation program. Recall that an environment can contain many different kinds of objects (fish, band members, and so on). Any object stored in an environment must implement the `Locatable` interface, though. This means that it must keep track of its location in the environment, and it must provide a `location` method to report its location. The `Fish` class used by the marine biology simulation implements the `Locatable` interface, as we'll see later. A `BandMember` class or `PhysicsParticle` class would also have to implement the `Locatable` interface if their objects were going to be put in an environment.

Once the `step` method has received the list of `Locatable` objects (fish, in this case) from the environment, it loops through them, asking each one to carry out whatever actions it performs in a single timestep. The `Locatable` interface does not have an `act` method, but the `Fish` class does. Since all the `Locatable` objects in the marine biology simulation's environment are actually instances of the `Fish` class, the `step` method can *cast* the current `Locatable` object to the `Fish` class and call the `act` method associated with `Fish`. (Analysis questions in Chapter 5 consider alternatives to this design choice.) Next, the `step` method asks the `EnvDisplay` object to display the state of the environment after the timestep.

---

‡ `EnvDisplay` and the other utility classes are "black box classes" in the case study. You do not need to read or understand the implementations of these classes unless you want to; they will not be tested on the AP Exam. You should, however, be familiar with the class documentation for `EnvDisplay` and the other utility classes listed in the Quick Reference at the end of this chapter. You should also be thoroughly familiar with the core classes, including the calls they make to utility class methods, such as the `showEnv` method.

The `Simulation` constructor and `step` method also include several calls to `Debug.println`, although the code above doesn't show the calls in the constructor. The `Debug` class is another utility class. `Debug.println` is like `System.out.println`, except that the string will only be printed if debugging has been turned on. There are other methods in the `Debug` class to turn debugging on and off. Because the `step` method does not include a call to one of these methods, we can assume that debugging is off unless the method that called `step` turned it on.
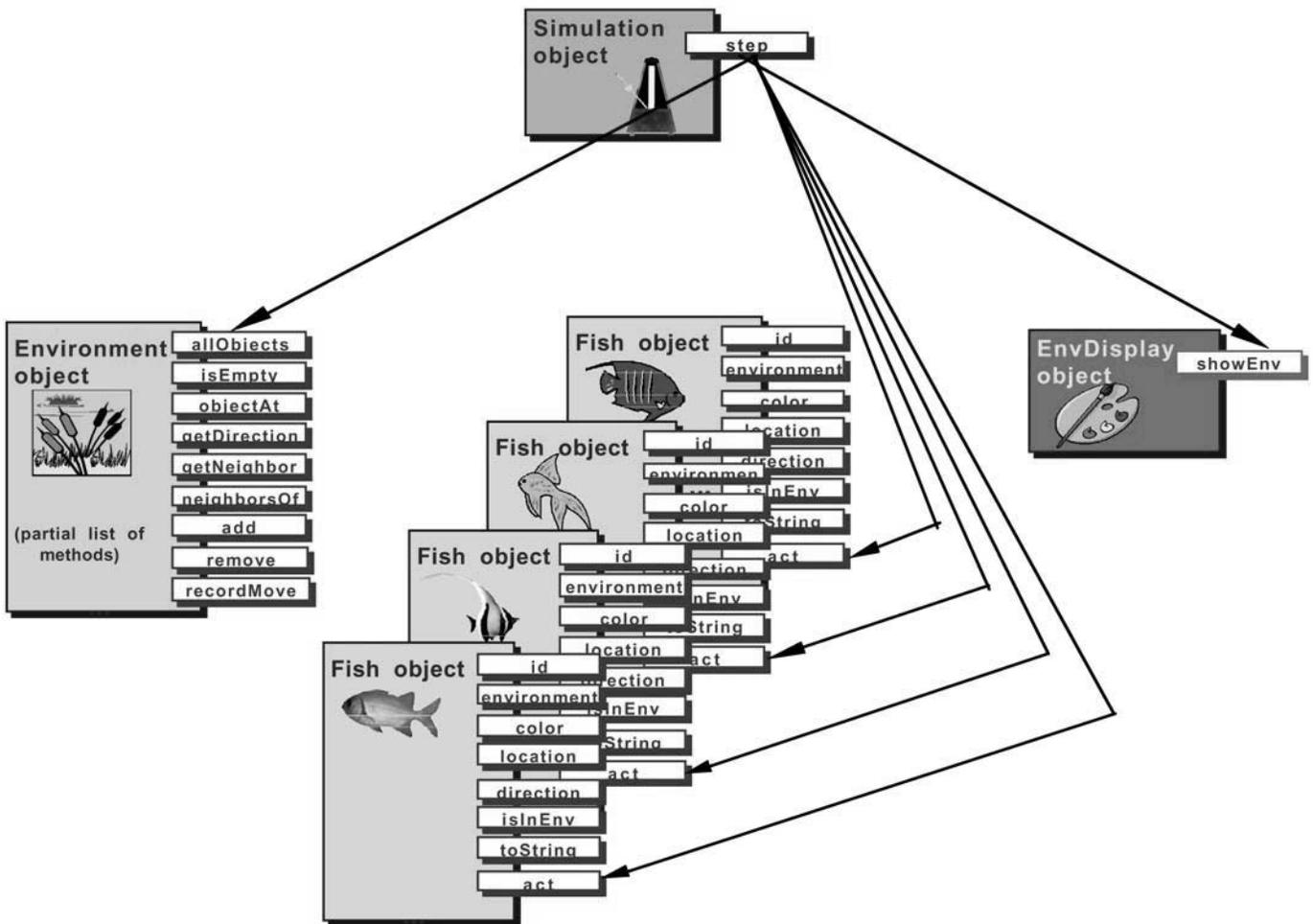
The `Environment` Interface

`Environment` is another interface! An environment object in the marine biology simulation program models a rectangular grid that contains objects at various grid locations. Jamie told me that since I would not be modifying the environment as part of my summer project, I could treat it as a "black box." A black box object or module is one whose internal workings are hidden from the user. A classic example in the real world is a toaster; most people know how to use a toaster without ever looking inside it to see how it is wired.

Treating the environment as a black box meant that I only had to worry about the list of public operations that objects of other classes, also known as *client code*, can use. Thus, I only had to become familiar with the `Environment` interface, and not with the classes that implement it. (Thinking back to `SimpleMBSDemo1` and `SimpleMBSDemo2`, though, I could guess that `BoundedEnv` must be a class that implements this interface.)

A partial list of the public operations specified in `Environment`, including those used by methods in the `Simulation` and `Fish` classes, appears below.

Partial List of Public Methods in the `Environment` Interface

```
Direction randomDirection()
Direction getDirection(Location fromLoc, Location toLoc)
Location getNeighbor(Location fromLoc, Direction compassDir)
ArrayList neighborsOf(Location ofLoc)

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

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

other tested methods not shown; see the class documentation in the Documentation folder

The first four methods in this partial list, `randomDirection`, `getDirection`, `getNeighbor`, and `neighborsOf`, deal with navigating around the environment using locations and directions. `Location` and `Direction` are two more utility classes: a `Location` object encapsulates the row and column of a cell in the environment grid, while the `Direction` class represents a compass direction and provides several constants such as `Direction.NORTH` and `Direction.SOUTH`. A partial list of the methods and constants for these classes is shown below. The `randomDirection` method in the `Environment` interface returns a randomly chosen valid direction, such as `NORTH`, `SOUTH`, `EAST`, or `WEST`. The `getNeighbor` method takes a location, `fromLoc`, and a direction, `compassDir`, and returns the location that is the neighbor of `fromLoc` in the given direction. For example, if `fromLoc` is the location (2, 4), then `getNeighbor(fromLoc, Direction.NORTH)` would return the location (1, 4). Similarly, the `getDirection` method returns the direction required to get from one location to another. If `toLoc` is the location (3, 4), then `getDirection(fromLoc, toLoc)` would return `Direction.SOUTH`. Finally, the `neighborsOf` method returns all the neighbors of a given location in an `ArrayList` object. (`ArrayList` is a standard Java class, found in `java.util`.) The call `neighborsOf(fromLoc)`, for example, would return a list containing the locations (1, 4), (2, 5), (3, 4), and (2, 3), although not necessarily in that order.

The other methods in the `Environment` interface deal with the objects in the environment, all of which must be `Locatable` objects. (Their classes must all have a `location` method.) The `numObjects` method indicates how many `Locatable` objects there are in the environment. The `allObjects` method returns a list of them in an array; in the marine biology simulation, that array contains `Fish` objects. (`Fish` are valid `Locatable` objects.) The `isEmpty` method indicates whether a particular location in the grid is empty, while the `objectAt` method returns the object at the location. Next, there are three methods that modify the environment: `add`, which adds a new object to the environment; `remove`, which removes an object from the environment; and `recordMove`, which records the fact that an object moved from one location to another. The `recordMove` method is necessary because the environment needs to know if a `Locatable` object (a fish, in this case) thinks it is now at a different location.

Selected Public Constants and Methods in:

`Location`	`Direction`
`Location`(int row, int col)	*Constants:* NORTH, SOUTH, EAST, WEST
int `row`()	`Direction`()
int `col`()	`Direction`(String str)
boolean `equals`(Object other)	Direction `toRight`(int degrees)
	Direction `toLeft`(int degrees)
	Direction `reverse`()
	boolean `equals`(Object other)

other tested constants and methods not shown; see the class documentation in the Documentation folder

Analysis Question Set 2:

Assume that `env` is a valid 20 x 20 `BoundedEnv` object. (`BoundedEnv` is a class that implements the `Environment` interface.) Consider the following code segment.

```
Location loc1 = new Location(7, 3);
Location loc2 = new Location(7, 4);
Direction dir1 = env.getDirection(loc1, loc2);
Direction dir2 = dir1.toRight(90);
Direction dir3 = dir2.reverse();
Location loc3 = env.getNeighbor(loc1, dir3);
Location loc4 = env.getNeighbor(new Location(5, 2), dir1);
```

1. What locations would you expect in the `ArrayList` returned by a call to `env.neighborsOf(loc1)` after this code segment?

2. What should be the value of `dir1`?

3. What should be the value of `dir2`? of `dir3`?

4. What location should `loc3` refer to?

5. What location should `loc4` refer to?

6. Read the class documentation for the `Location` and `Direction` classes. What other constructors and methods do they have? How might you use their `toString` methods in a program? What do the additional `toRight` and `toLeft` methods in `Direction` do?

Exercise Set 1:

1. Write a simple driver program that constructs a `BoundedEnv` environment (similar to the one in `SimpleMBSDemo1`) and then test your answers to the Analysis Questions above. The `ArrayList`, `Location`, and `Direction` classes all implement the `toString` method to provide useful output.

2. In your driver program, use the `inDegrees` method from the `Direction` class to discover the degree representations for the `Direction` constants `NORTH`, `SOUTH`, `EAST`, and `WEST`. What is the value of `dir3` in degrees?

The `Fish` Class

A `Fish` object has several *attributes*, or features. It has an identifying number, its ID, which is useful for keeping track of the different fish in the simulation, especially during debugging. It has a color, which is used when displaying the environment. A fish also keeps track of the environment in which it lives, its location in the environment, and the direction it is facing. The `Fish` class encapsulates this basic information about a fish with its behavior. A list of the methods in the `Fish` class appears below.

Methods in the `Fish` Class

```
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 void move()
protected Location nextLocation()
protected ArrayList emptyNeighbors()
protected void changeLocation(Location newLoc)
protected void changeDirection(Direction newDir)
```

The constructors, related helper methods, and fields:

The `Fish` class has three constructor methods. All three require that the client code constructing the fish specify the environment in which the fish will live and its initial location in that environment. The second constructor allows the client code to specify the fish's direction, while the third allows client code to specify both the direction and the color. (`Color`‡ is a standard Java class found in `java.awt`.) Rather than repeating the code to initialize a new fish's state in all three constructors, the original programmer chose to have them call an internal helper method, `initialize`.

---

‡ The `Color` class is not part of the AP Computer Science Java subset, but is useful for this case study. Students will be expected to be able to use the `Color` constructor and constants documented in the Quick Reference at the end of this chapter, which will also be available as a reference during the AP Exam.

This method has four parameters (environment, location, direction, and color). The Fish constructors call methods to choose random directions and colors for fish whose direction and color were not provided by the client code. The Environment class has a randomDirection method, which the first Fish constructor uses to randomly choose a direction for the fish. Unfortunately, the Color class does not provide a randomColor method, so the original programmer implemented one in the Fish class. The code below shows the Fish constructors and the initialize method.

```java
// 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

// constructors and related helper methods
public Fish(Environment env, Location loc)
{
    initialize(env, loc, env.randomDirection(), randomColor());
}

public Fish(Environment env, Location loc, Direction dir)
{
    initialize(env, loc, dir, randomColor());
}

public Fish(Environment env, Location loc, Direction dir, Color col)
{
    initialize(env, loc, dir, col);
}

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);
}
```

The `initialize` method initializes four of the fish's five *instance variables*, `theEnv`, `myLoc`, `myDir`, and `myColor`, from its parameters. The remaining instance variable, `myID`, is initialized from the *class variable* called `nextAvailableID`. Instance variables encapsulate the data (or state) of an object, whereas all objects of the class share a class variable. Class variables are indicated by the `static` keyword. (Some textbooks use the term *field*, which is more general and refers to instance variables, class variables, and constants.) The `Fish` class variable, `nextAvailableID`, is a single integer that all `Fish` objects have access to. Each fish initializes its own ID to the current value of `nextAvailableID` and then increments it for the next fish. The first fish gets ID 1, the next one gets 2, and so on. If `nextAvailableID` were an instance variable, like `myLoc` and `myColor`, every fish would have its own copy of the variable, and incrementing the value would have no affect on the other fish. As a result, every fish would end up having the same ID. Since `nextAvailableID` is a class variable and not tied to any single object of the `Fish` class, it must be initialized when it is declared in the class rather than in the `Fish` constructor.‡

The last line in `initialize` tells the environment to add the new fish to the environment. (The `this` keyword refers to the object being constructed.) It is important that this line comes after the initialization of `myLoc` because the `add` method in `Environment` uses the object's location to place it in the environment. (Remember that all objects in an environment must be `Locatable`.) At this point the fish is fully constructed and ready to act; its instance variables are initialized and it has been placed in the appropriate location in the environment.

The `randomColor` method called by the first two `Fish` constructors generates a random color using three randomly chosen integers in the range of 0 to 255, representing the red, green, and blue aspects of the color. First, though, `randomColor` has to get a random number generator, as shown in the code on the next page. The `RandNumGenerator` class is a class that has just one method, `getInstance`. The method is `static`, meaning that, like a class variable, it is tied to the `RandNumGenerator` class, not to any particular object, and you can call it using the class: `RandNumGenerator.getInstance()`. The purpose of the `getInstance` method is to get a `Random` object (`Random` is a standard Java class found in `java.util`) that can be used to generate random numbers (or, technically, *pseudo-random* numbers). In fact, the call `RandNumGenerator.getInstance()` always returns the same `Random` object, no matter how often it is called.‡‡

---

‡ Class variables are not part of the AP Computer Science Java subset, but are useful for this case study. However, they will not be tested on the AP Exam.

‡‡ The advantage of using the `Random` object returned by `RandNumGenerator.getInstance()` rather than constructing a new `Random` object directly, is that always using the same random number generator produces a better set of random numbers. Multiple random number generators in a program can generate sequences of numbers that are too similar.

The `nextInt` method in the `Random` class takes an integer parameter, `n`, to generate a random number in the range of `0` to `n-1`, so the call `randNumGen.nextInt(256)` returns one of the 256 integers from `0` to `255`. The `randomColor` method calls `nextInt` three times to generate a color with a random amount of red, green, and blue.

```
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
}
```

One thing I noticed about `randomColor` (and the other helper methods further down in the `Fish` class) is that it is declared `protected`, rather than `public` or `private`. I was not familiar with the `protected` keyword. Jamie told me that it would be useful when I started creating new types of fish, but that in the meantime I could pretend that `randomColor` and the other helper methods were `private`. Like `private` methods, they are internal methods provided to help methods in the `Fish` class get their job done, and are not meant to be used by external client code.‡

---

‡ The `protected` keyword is not part of the AP Computer Science Java subset. Students will be expected to be able to use and redefine `protected` methods on the AP Exam in the context of the case study, but will not be tested on the visibility rules of `protected` methods. Note that although Jamie told Pat to pretend that the `protected` methods are `private`, Java compilers will not actually keep external client code in the marine biology program from accessing these methods. See the Specialized Fish chapter and the Teacher's Manual for a more detailed discussion of the `protected` keyword and its use in the case study.

Analysis Question Set 3:

Assume that `env` is a valid 20 x 20 `BoundedEnv` object. Consider the following code segment.

```
Location loc1 = new Location(7, 3);
Location loc2 = new Location(2, 6);
Location loc3 = new Location(4, 8);
Fish f1 = new Fish(env, loc1);
Fish f2 = new Fish(env, loc2);
```

1. What should be the return value of `env.numObjects()` after this code segment?

2. What should be the return value of `env.allObjects()`?

3. What should be the return value of `env.isEmpty(loc1)`?

4. What should be the return value of `env.isEmpty(loc3)`?

5. What should be the return value of `env.objectAt(loc2)`?

6. Read the class documentation for the `Environment` interface. What should be the return value of `env.objectAt(loc3)`?

7. Based on what you know about the `Fish` constructors, does it make sense to add a fish directly to the environment from client code using the `add` method in `Environment`?

8. Why isn't the `initialize` method a `public` method?

9. Could the reference to the environment (`theEnv`) have been a class variable rather than an instance variable?

Exercise Set 2:

1. Write a simple driver program that constructs a `BoundedEnv` environment (similar to the one in `SimpleMBSDemo1`) and then tests your answers to the Analysis Questions on the previous page. The `Fish` and `Location` classes implement the `toString` method to provide useful output. To print the objects in a Java array, though, you will need to step through the array and get the string representation for each individually. (Remember that the `Simulation step` method has an example of stepping through an array returned by the `allObjects` method.)

2. Add a few more fish to your program and verify that `numObjects` and `allObjects` behave as you expect. Remove a few fish using the `Environment remove` method and test `numObjects` and `allObjects` again. Then test `isEmpty`; does it behave as you expect if you pass it the location of one of the fish you removed?

3. Add one of your fish to the environment using the `add` method in the `Environment` class. What happens? Why?

Simple accessor methods in `Fish`:

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

The simplest methods in the `Fish` class are those that correspond to its conceptual attributes: `id`, `environment`, `color`, `location`, and `direction`. For example, the code for the `id` method appears below.

```
public int id()
{
    return myId;
}
```

The `location` method is particularly important, because it is this method and the `"implements Locatable"` phrase at the beginning of the `Fish` class definition that make `Fish` valid `Locatable` objects. In other words, the `location` method and the `"implements Locatable"` phrase make `Fish` valid objects to put in an environment.

The other two accessor methods are `isInEnv` and `toString`. The `toString` method is a typical Java method that captures basic information about a fish and puts it in a string for display or debugging purposes. The `isInEnv` method, though, is more interesting. It tests whether the fish is in the environment and at the location where it thinks it is. This should always be the case, and if it isn't the case, the fish is in an *inconsistent* state. What does it mean, for example, to ask a fish for its location in the environment with the `location` method if the fish isn't in an environment at all? What does it mean to ask the fish to move in that situation? The `isInEnv` method provides a way to test whether the fish is in a consistent state by asking the environment for the object at the location where the fish thinks it should be and testing that the found object is the fish itself. (The keyword `this` in a Java method refers to the object on which the method was invoked.)

```
public boolean isInEnv()
{
    return ( environment().objectAt(location()) == this );
}
```

Analysis Question Set 4:

1. Does a fish start out in a consistent or inconsistent state? In other words, is it in a consistent state immediately after it is constructed?

2. How could a fish get into an inconsistent state? (Hint: look at the methods available in the `Environment` interface.)

Fish movement methods — `act` and its helper methods:

```
public void act()
protected void move()
protected Location nextLocation()
protected ArrayList emptyNeighbors()
protected void changeLocation(Location newLoc)
protected void changeDirection(Direction newDir)
```

The most important method in the `Fish` class from the client code's perspective is the `act` method. This is the method in which a fish does whatever action is appropriate for a single timestep in the simulation. The code for the `act` method, though, actually does very little. It first checks that the fish is still alive and well in the environment (in other words, in a consistent state) and then, if it is, calls the internal, protected `move` method.

```
public void act()
{
    if ( isInEnv() )
        move();
}
```

The `move` method first looks for a location to which the fish can move. The `nextLocation` method returns the next location the fish should move to; if the fish can't move, `nextLocation` returns the current location. If the new location is not equal to the current location (the fish can move), `move` calls another helper method, `changeLocation`, to actually move there. It also changes direction to reflect the direction it moved. For example, if a fish facing north at location (4, 7) were to move west to location (4, 6), it would change its direction to show that it moved west. The code for the `move` method, with debugging messages and comments removed, is shown below. It uses the `equals` method in the `Location` class rather than the `==` operator because it only cares whether the row and column values are the same, which is what the `equals` method checks, not whether the two locations are exactly the same `Location` object.

```
protected void move()
{
    Location nextLoc = nextLocation();

    if ( ! nextLoc.equals(location()) )
    {
        Location oldLoc = location();
        changeLocation(nextLoc);

        Direction newDir = environment().getDirection(oldLoc, nextLoc);
        changeDirection(newDir);
    }
}
```

The picture below illustrates the behavior of the `Fish act` and `move` methods.

Fish object

	id
	environment
	color
randomColor	location
move	direction
nextLocation	isInEnv
emptyNeighbors	toString
changeLocation	
changeDirection	act
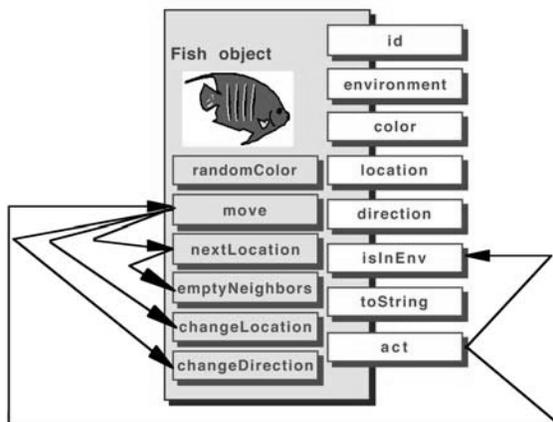
General Outline:

Fish act **method**
A. calls isInEnv to verify that fish is still in environment
B. calls move, which
   i. calls nextLocation to decide where to move, which
     a. calls emptyNeighbors to find empty neighboring locations
     b. randomly chooses one of those neighboring locations to move to
   ii. calls changeLocation to move there
   iii. decides which direction to face
   iv. calls changeDirection to face that direction

The `nextLocation` method called by `move` finds the next location for the fish. A fish can move to the cell immediately in front of it or to either side of it, so long as the cell it is moving to is in the environment and empty. The first thing `nextLocation` does, therefore, is to get a list of all the neighboring locations that are in the environment and that are empty. It calls the `emptyNeighbors` helper method to do this. Then it removes the location behind the fish from the list, since the fish is not allowed to move backward. Finally, `nextLocation` randomly chooses one of the valid empty locations and returns it (or returns the current location, if the fish can't move). The *pseudo-code* below summarizes the activities of the `nextLocation` method.

```
Pseudo-code for nextLocation method

get list of neighboring empty locations (by calling emptyNeighbors())
remove the location behind the fish
if there are any empty neighboring locations
    return a randomly chosen one
else
    return the current location
```

The `emptyNeighbors` method, shown below, is pretty straightforward, so we'll look at it before continuing on with `nextLocation`. In `emptyNeighbors`, the fish first asks the environment for a list of all its neighboring locations. Since the neighboring locations are not necessarily empty, `emptyNeighbors` constructs a new list and copies all the neighbors that happen to be empty into the new list. The `emptyNeighbors` method doesn't know in advance how many empty neighbors there will be, so the `emptyNbrs` list is an `ArrayList`, a list that can grow and shrink after it is created. (`ArrayList` is a standard Java class found in `java.util`.) This is the list that `emptyNeighbors` returns.

```
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;
}
```

At this point I was curious, so I asked Jamie why the `neighborsOf` method in `Environment` returns an `ArrayList` rather than an array. It seemed to me that `neighborsOf` would always know how many locations it was returning. Jamie pointed out that in a bounded environment some of a location's neighbors would be out-of-bounds. The `neighborsOf` method only returns valid neighbors, so `nbrs.size()` in `emptyNeighbors` does not always evaluate to 4. Since `neighborsOf` only returns locations that are in the environment, we know that `emptyNeighbors` does too.

What happens once `emptyNeighbors` returns the list of empty neighboring locations to `nextLocation`? Since fish cannot move backward, `nextLocation` calculates the "backward" direction and removes the neighbor in that direction, if it is in the list. To do this, it calls the `remove` method in `ArrayList` that takes an object (in this case a `Location` object) as a parameter, finds an equivalent object in the list, and removes it.‡

```
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);
}
```

---

‡ There are two `remove` methods in `ArrayList`. One takes an integer index as a parameter and removes whatever object is at that index; the other takes an object as its parameter and finds and removes the equivalent object from the list. The second `remove` method is not part of the AP Computer Science Java subset, but is useful for this case study. It may be tested on the AP Exam in the context of the case study.

As long as there are any empty neighboring positions left in the list, the fish randomly chooses one of them as its new position. As in the `randomColor` method, `RandNumGenerator.getInstance()` returns the one instance of the `java.util.Random` class being used in the simulation program. Recall that the `nextInt` method takes an integer parameter, `n`, and generates a random number in the range of `0` to `n-1`. For instance, if the size of the `emptyNbrs` list is 3, then `randNum` will be set to one of the three numbers `0`, `1`, or `2`. The `nextLocation` method then uses that random number as the index into the `emptyNbrs` list and calls the `get` method in `ArrayList` to retrieve the location at the specified index. An `ArrayList` stores its objects generically as `Object` instances, so the location in the `emptyNbrs` list must be cast to the `Location` class before `nextLocation` can return it to the `move` method as the newly chosen location.

Analysis Question Set 7:

Consider the following bounded environments.
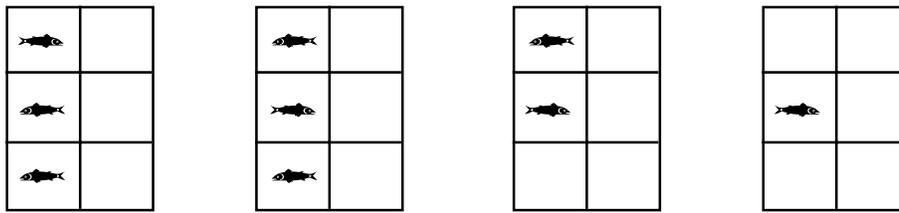
1. How many neighboring locations would the `Environment neighborsOf` method return for location $(1, 0)$ in each environment?

2. How many neighboring locations would `emptyNeighbors` return for the fish in location $(1, 0)$ in each environment?

3. What are the possible locations that `nextLocation` might return for the fish in location $(1, 0)$ in each environment?

We already saw that once the `move` method knows where it wants to move, it calls `changeLocation` to actually move there and `changeDirection` to actually change the direction. These methods change the `myLoc` and `myDir` instance variables. The `changeLocation` method also notifies the environment to update itself. This is necessary because the environment keeps track of the locations of all its objects, and it is critical that the fish and environment agree where the fish is. To do this, the fish passes both itself and its old location to the `recordMove` method in `Environment`. It does not need to pass its new location to `recordMove`; the environment knows that the object is `Locatable`, so it asks the object for the new location.

```
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
}

protected void changeDirection(Direction newDir)
{
    myDir = newDir;
}
```

Note that the `isInEnv`, `nextLocation`, `emptyNeighbors`, `changeLocation`, and `changeDirection` methods use calls to the fish's own accessor methods, `direction()`, `location()`, and `environment()`, rather than accessing `myDir`, `myLoc`, or `theEnv` directly. This is good programming practice, even though it's a little harder to read, because it limits the number of places in the code that depend on the internal representation of the object's data. (It also turned out to be useful later when implementing other types of fish.) In the statements that modify the fish's location and direction, though, `changeLocation` and `changeDirection` set `myLoc` and `myDir` directly, since they can't use accessor methods to modify values.

Analysis Question Set 8:

1. Consider the fish in location (2, 2) in the example at the beginning of this chapter. Step through the `move` and `nextLocation` methods for this fish. What will `emptyNeighbors` return to `nextLocation`? What will `oppositeDir` be set to? Where might the fish move?

2. Now that you've seen all the methods in the `Fish` class, consider modifying it to keep track of a fish's age. What changes would you need to make to the `Fish` class?

Exercise Set 4:

1. Modify the `Fish` class to add a public `changeColor` method. The method should take the new color as a parameter.

2. Modify `SimpleMBSDemo2` to test your new `changeColor` method. Construct new fish using the third `Fish` constructor, specifying the color of each fish as you construct it. Change the color of at least one fish at some point in the simulation. For example, you might modify the simulation loop so that one of your fish changes to `Color.red` at the beginning of even numbered timesteps and `Color.yellow` at the beginning of odd ones. The Java `Color` class provides a number of constant color values, such as `red`, `orange`, `yellow`, `green`, `blue`, and `magenta`. To use these constants, you will need to import `java.awt.Color` into `SimpleMBSDemo2.java` and you will need to refer to the colors by their "full names" (`Color.red`, and so on).

Test Plan

A crucial element in any software development project is a well-defined test plan. The test plan should consist of test cases derived from the problem specification (known as "black box test cases" because they treat the entire program as a black box) and test cases derived from an analysis of the code. The test plan should also include the expected results for every test case. It is very important to identify the expected results for each test *before* running it, or else subtle errors may not be caught.

Testing Programs with Random Behavior

Programs with random behavior can be difficult to test. Every time the program is run you get different results. This makes it difficult to say what the expected results of a given test case should be and, therefore, whether the actual results are correct. Another difficulty is verifying that the probabilities of various behaviors are correct. For example, a fish with two neighboring empty locations should move to each one half the time, but that does not mean that in actual tests it will go to each location exactly half the time. It might go to one slightly less than half the time and go to the other slightly more than half the time and still be exhibiting correct behavior. The challenge is to analyze test results and determine whether they demonstrate the appropriate probabilities.

One technique for testing programs using random numbers is to *seed* the random number generator. This will cause the generator to create the same sequence of pseudo-random numbers every time the program is run, leading to predictable results. For example, the initial configuration file specifies the initial location and direction of each fish, but not its color. If we seed the random number generator, then the initial colors of the fish and their behavior in each timestep will be the same every time we run the simulation. If a given fish is constructed facing south and has empty neighboring locations to its south and east, it will always move either south or east in the first time step.

Another technique is to run the program many times, without seeding the random number generator. Each run will yield different results, but the accumulation of results will demonstrate whether the probabilistic behavior is as expected. For example, a fish facing south with empty neighboring locations to its south and east will move south approximately half the time and move east the rest of the time.

A third technique is to include many different test cases in each run and see if the numerous cases create the range of expected behavior. For example, if we have many different south-facing fish with empty neighboring locations to the south and east, approximately half of them should move south and half should move east in any given timestep. This technique can be combined with a seeded random number generator to

test probabilistic behavior with predictable results. Although the behavior of all the fish will be the same every time we run the simulation, with enough fish and enough timesteps we can test many different test cases and see a range of behaviors. An added benefit is that after the first run we can predict exactly which fish will show which behavior in which timestep. This predictability makes it easier to test that a program modification does what we want (and doesn't break anything else).

Black Box Test Cases

The problem specification defined the following requirements for the simulation.

- No cell should have more than one fish.
- In each timestep, each fish should
    - move to a randomly-chosen adjacent empty location, but
    - never move backward.
- Initial configuration information for the environment should be provided in a file.

The second and third requirements yield the following test cases and informal expected results after a single timestep. Many of the tests regarding the initial configuration file are *boundary tests*, testing extreme conditions like an empty file or a completely full environment.

- An empty file or one that contains invalid data (such as just one dimension for the environment) should generate an error.
- A valid file with valid environment dimensions but no fish should result in an empty environment, but no errors.
- A file with a single fish should run with no errors. (The behavior of the fish will depend on its starting location.)
- A file with two or more fish in the same location should generate an error.
- A file with a fish in every location in the environment (but only one fish in every location) should run with no errors. None of the fish should move.
- A fish with no empty locations around it should stay where it is.
- A fish with a single adjacent empty location in front or to the side of it should always move there.
- A fish with a single adjacent empty location behind it should stay where it is.
- A fish with two adjacent empty locations in front or to the side of it should move to either of its neighboring empty locations with equal likelihood, never staying where it is and never moving backward.
- A fish with three adjacent empty locations in front and to the side of it should move to any of its three neighboring empty locations with equal likelihood, never staying where it is and never moving backward.

These test cases lead to actual test runs, such as one with an empty configuration file, a file with valid environment dimensions but no fish, and a file with as many fish as there are locations in the environment. The fish configuration at the beginning of this chapter, repeated here, could form the basis of another test run. This would test the case of a fish with a single valid location to which it could move (the fish in the lower right corner), and the cases of fish with two and three valid neighboring locations. The `fish.dat` file could form the basis of another test run.

Test Cases Based on Code

Additional test cases may come from analysis of the code, once it is written. For example, consider the `act` and `emptyNeighbors` methods in the `Fish` class. The black box test cases do not cover the test case from the `act` method in which the method asks a fish that has been removed from the environment to act. Similarly, consider the loop with the embedded conditional statement in `emptyNeighbors`. The tests we need include a case where execution does not enter the loop (there are no neighboring locations, empty or not), a case where it loops through once (one neighbor), a case where it loops through multiple times (multiple neighbors), a case where execution enters both the loop and the body of the `if` statement (a neighboring location that is empty), and a case where the `if` statement is false (at least one neighboring location that is not empty). The black box test cases cover the last three cases, but not the first two. To test the situation in which a fish has absolutely no neighbors requires a test run with a 1 x 1 environment with a fish in the only cell.

Analysis Question Set 9:

1. Some exceptional cases only happen if the program itself is wrong. Others might occur even if the program is correct; if, for example, the format of the initial configuration file is wrong. Identify at least one exceptional case that can easily be tested by modifying the initial configuration file.

2. Which black box test cases cover the last three cases for the loop with an embedded conditional statement in `emptyNeighbors` (multiple neighboring locations, an empty neighboring location, a non-empty neighboring location)?

3. What kind of environment would you construct to test the situation in which a fish has exactly one valid neighboring location?

The `Debug` Class

In addition to observing the actions of the simulation by manipulating the initial configuration file and noting the results after each timestep, we can watch the simulation execute by running it with a debugger or, at a less detailed level, by calling the `Debug.println` method in the program. The `Debug` class is another utility class, some of whose methods are shown below.

Selected Public Methods in the `Debug` Class

```
static void turnOn()
static void restoreState()
static void print(String message)
static void println(String message)
```

other tested methods not shown; see the class documentation in the Documentation folder

All `Debug` methods are `static`, meaning that they are not tied to any particular `Debug` object, and it is not necessary to create a `Debug` object to call them. The `print` and `println` methods are like `System.out.print` and `System.out.println`, except that they will print only if debugging has been turned on. To trace what is happening during a fish move, for example, we could add

```
Debug.turnOn();
```

to the beginning of the `Fish move` method. The debugging output will show all of the neighboring locations around a fish (empty or not) and indicate whether the fish is stuck or which location it is moving to if it moves. Seeing this information for a number of fish over a number of moves can help us verify that the neighborhoods are correct and that the fish are moving in various directions in the proportions we expect. Before the `move` method returns, a call to

```
Debug.restoreState();
```

restores debugging to whatever it was before the call to `turnOn`. If debugging had been off, the call to `restoreState` will turn it off again. If debugging had already been on, the call to `restoreState` will keep it on.

Exercise Set 5:

1. Run the simulation. Set the seed to any arbitrary value (for example, 17). Run the simulation several times with the same configuration file and the same seed value to make sure that every run displays the same behavior. Run the simulation several times with the same configuration file but with different seed values; does every run display the same behavior, or different behavior? *[Reminder: In the distributed version of the case study, the class containing the main method is* `MBSGUI`*. To set the seed, select "Use fixed seed . . ." from the* `Seed` *menu.]*

2. Turn on debugging in the `Simulation step` method right before the first line with a `Debug.println` statement. Restore the debugging state immediately after the calls to `Debug.println`. Run the program. What does the output tell you?

3. Run the simulation with the `fish.dat` initial configuration file. Make a record of the test cases covered by the first 5 timesteps of your run, including the actual results.

4. If you're running a user interface that has a Save function, save a copy of the results after 5 timesteps. Give the file a descriptive name, like `after5steps.dat`. Run the program for 10 timesteps and save the results in another file with a descriptive name.

5. Looking at your results from each timestep in the test in Exercise 3, can you determine the order in which the environment's `allObjects` method returns the fish? How does that order influence how many empty neighbors there are around each fish?

6. Look at the `fish.dat` initial fish configuration file. The first line specifies that the environment is bounded and then gives its dimensions (number of rows followed by number of columns). Each line after that has the class name of an object in the environment (`Fish`, in this case) and its location (row and column) and direction. Design test data to test black box test cases that have not already been covered by `fish.dat`.

7. Determine the test cases required to test the `move`, `nextLocation`, and `emptyNeighbors` methods in the `Fish` class, based on their code. Have your test cases been tested by the black box test cases?

What alternative designs did the original programmer consider?

The implementation of the marine biology simulation program flows from a number of design decisions made early on. The original programmer considered other design possibilities as well, all of which address several fundamental questions. Who is responsible for keeping track of the fish in the environment? Who is responsible for actually moving a fish? Who is responsible for knowing what behavior (moving, aging, breeding, dying, etc.) is required as part of the simulation? Consider, for example, the following possible scenarios.

- Whatever method calls the `Simulation step` method could pass it a list of all the fish, and then the simulation could ask the fish to move. In this case, the object whose method calls `step` would have to keep track of all the fish in the environment, or would have to ask the environment for the list to pass to the `step` method. The `Simulation` class would be responsible for knowing what behavior is part of the simulation, and the `Fish` class would be responsible for knowing how to move a fish.

- The simulation could keep track of all the fish in the environment and ask them to move. Again, the `Simulation` class would be responsible for knowing what behavior is part of the simulation, and the `Fish` class would be responsible for knowing how to move a fish.

- The simulation could keep track of both the environment and all the fish in the environment. It could pass the fish to the environment and ask it to move them. In this case, the `Environment` class would be responsible for knowing what behavior is part of the simulation and how to move a fish.

- The simulation could keep track of the environment (or be passed it as a parameter), and ask it to move all the fish. In this case, the `Environment` class would be responsible for keeping track of the fish, knowing what behavior is required in the simulation, and knowing how to move a fish.

- The simulation could keep track of the environment (or be passed it as a parameter), and ask it for all the fish. Then the simulation could ask the fish to move. In this case, the `Environment` class would be responsible for keeping track of the fish in the environment and providing a list of them to other objects when asked. The `Simulation` object would be responsible for knowing what behavior is required as part of the simulation. The `Fish` class would be responsible for knowing how to move a fish.

Analysis Question Set 10:

1. Which scenario above corresponds to the design chosen by the original programmer?

2. Pick a different scenario and describe how the specifications, or lists of `public` methods, for the three core classes, `Simulation`, `Environment`, and `Fish`, would be different under that scenario.

3. Which of the scenarios above do you think represent particularly good design choices? Why?

4. Do you think any of the scenarios above would be a poor design choice? Why?

5. Do any of the designs lead to classes that are general enough that they could be used in other applications? Do any of the designs lead to classes that could not be used in other applications?

Quick Reference for Core Classes and Interfaces

This quick reference lists the constructors and methods associated with the core classes described in this chapter. Public methods are in normal type. *Private and protected methods are in italics.* (Complete class documentation for the marine biology simulation classes can be found in the `Documentation` folder.)

```
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)
```

```
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 void move()
protected Location nextLocation()
protected ArrayList emptyNeighbors()
protected void changeLocation(Location newLoc)
protected void changeDirection(Direction newDir)
```

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

This quick reference lists the public constants, constructors, and methods associated with the utility classes described in this chapter. The marine biology simulation program also uses subsets of the following standard Java classes: `java.util.ArrayList`, `java.util.Random`, and `java.awt.Color`. (Complete class documentation for the marine biology simulation classes can be found in the `Documentation` folder).

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(Object other)
Direction toRight()
Direction toRight(int degrees)
Direction toLeft()
Direction toLeft(int degrees)
Direction reverse()
String toString()
static Direction randomDirection()
```

The following are not tested:
```
FULL_CIRCLE
int hashCode()
Direction roundedDir(int numDirections,
          Direction startingDir)
```

```
EnvDisplay Interface

void showEnv()
```

```
Locatable Interface

Location location()
```

```
Location Class

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

The following is not tested:
```
int hashCode()
```

```
RandNumGenerator Class

static Random getInstance()
```

Java Library Utility Classes

```
java.util.ArrayList Class (Partial)

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

Marine Biology Simulation Case Study

Chapter 3

Creating a Dynamic Population

After Jamie described the existing implementation of the marine biology simulation, I looked over the code some more to be sure I understood it. Then I talked to the marine biologists about what modifications they wanted in the program. They decided that the first modification would be to create a dynamic, or changing, population, with new fish being born and other fish dying as the program ran.

Problem Specification

The marine biologists wanted the simulation to model more complex fish behavior. They asked me to modify the simulation so that, in any given timestep, a fish might breed, move, or die. We talked about keeping track of a fish's age and having the likelihood of breeding or dying depend on that, but the biologists decided that they didn't need that level of sophistication right away. Instead they decided to build the model around some probabilities, giving fish a certain chance of breeding and a certain chance of dying in any given timestep. After some analysis of experimental data they had gathered from a sample real-world environment, the marine biologists specified that a fish should:

- have a 1 in 7 chance of breeding,
- breed into all its empty neighboring locations if it does breed,
- attempt to move to an empty neighboring location when it does not breed,
- never move backwards (unchanged from previous version), and
- have a 1 in 5 chance of dying after it has bred or moved.

Design and Implementation

My first step was to write *pseudo-code* for the `act` method that described what I thought it should do. Rather than putting the code for breeding and dying in the `act` method, I decided to break out these activities into separate `breed` and `die` methods. I also decided to deal with the "error condition" (or at least unexpected condition) of a fish that has been removed from the environment at the very beginning of the method. This is a common practice.

```
      if the fish is no longer in the environment
         do nothing (return immediately)

     if this is the 1 in 7 chance of breeding
         call the breed method
     else
         call the move method

     if this is the 1 in 5 chance of dying
         call the die method
```

Then I went on to implement the `breed` method. Since the `breed` and `die` methods should be called from `act` and not from methods outside the class I wanted to make them `private`, but I decided to follow the lead of the original programmer and make them `protected`, just like the `move` method. I still didn't understand what the difference between `private` and `protected` was, but Jamie assured me that this would become clearer when I started creating new kinds of fish.

According to the problem specification, a fish should breed into all of its empty neighboring locations. This means that the first step is the same as in the `nextLocation` method — getting a list of the empty neighboring locations — although, in this case, there's no reason to remove the location behind the fish from the list. As in `nextLocation`, if there are no empty neighboring locations then we're done with the method. Once we have the list of empty neighbors, we want to add a new fish to all of them rather than move to just one of them. This means looping through the list, constructing a new `Fish` object for each empty location and adding it to the environment. I reviewed the `Environment` interface and found the `add` method, but then I remembered that fish add themselves to the environment as they are constructed so I wouldn't need to do that separately. Here's the first version of the code.

```
protected void breed()       // first draft!
{
    // Get list of neighboring empty locations.
    ArrayList emptyNbrs = emptyNeighbors();

    // If there is nowhere to breed, then we're done.
    if ( emptyNbrs.size() == 0 )
        return;

    // Breed to all of the empty neighboring locations.
    for ( int index = 0; index < emptyNbrs.size(); index++ )
    {
        // Create new fish, which adds itself to the environment.
        Location loc = (Location) emptyNbrs.get(index);
        Fish child = new Fish(environment(), loc);
        Debug.println("  New Fish created: " + child.toString());
    }
}
```

Looking at this code I realized that a fish's 1 in 7 chance of breeding is not the only reason why it might not breed. If there are no empty neighboring locations, then it can't breed either. This means that the fish's chance of breeding is actually less than 1 in 7. I talked to the marine biologists about this, and they decided that a 1 in 7 chance of *attempting* to breed was fine. I then decided that, rather than have some of the logic about whether a fish breeds or not in the `act` method and some in the `breed` method, I would put both tests in `breed` and have it return a `boolean` value indicating whether the fish successfully bred or not.

In mathematics, probabilities are represented as real numbers from 0.0 to 1.0, where 0.0 means there is no probability that something will happen and 1.0 means that it will always happen. The probability corresponding to a 1 in 7 chance is the mathematical value 1/7 (approximately 0.143). Therefore, the test for whether the fish should attempt to breed or not involves randomly picking a real number (a `double`) in the range of 0.0 to 1.0 (using the `nextDouble` method in `java.util.Random`), and comparing it to the mathematical value 1/7. There is a 1 in 7 chance that the randomly chosen value will be less than 1/7 and a 6 out of 7 chance that it will be greater than or equal to 1/7. Similarly, there would be a 3 out of 4 chance that the randomly chosen value would be less than 3/4, or 0.75.

I decided to store the probability of breeding and the probability of dying in instance variables of the `Fish` class rather than as hard-coded constants, both to give them meaningful names and because this would provide more flexibility for the future. For example, the biologists might decide that each fish should have its own chance of breeding, which could be set in the `Fish` constructor. For now, though, I decided to initialize both instance variables in the `initialize` method.

On Jamie's advice, I made one other change to the `breed` method. I took out the code that actually creates a new fish and put it in a separate method, `generateChild`. Jamie told me that doing this would make it easier for me to develop new kinds of fish later. To make it clearer which fish had bred, and which spaces it had bred into, I decided to use the four-parameter `Fish` constructor to give newborn fish their parent's color. In other words, a red fish's children will also be red, their children will be red, and so on.

The code below shows the new instance variables and the modified `breed` method, with additions or changes to breed in boldface.

```
private double probOfBreeding;      // defines likelihood in
                                    // each timestep
private double probOfDying;         // defines likelihood in
                                    // each timestep


protected boolean breed()      // 2nd draft!
{
    // 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();

    // If there is nowhere to breed, then we're done.
    if ( emptyNbrs.size() == 0 )
        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;
}


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());
}
```

Because I had moved the test of whether to breed from `act` to `breed`, I needed to update my pseudo-code for the `act` method.

```
Updated pseudo-code for act method
    if the fish is no longer in the environment
        do nothing (return immediately)

    attempt to breed by calling the breed method
    if the fish did not breed
        call the move method to attempt to move

    if this is the 1 in 5 chance of dying
        call the die method
```

The last method called by `act` is the `die` method. When a fish is constructed, it initializes its instance variables and adds itself to the environment. When a fish dies, it must remove itself from the environment. It does not need to "uninitialize" its instance variables, nor does it need to "destroy" itself; this cleanup is handled by the Java garbage collector once there are no references to the fish left in the program. I reviewed the `Environment` documentation and verified how to call its `remove` method before writing this simple method.

```java
protected void die()
{
    environment().remove(this);
}
```

Analysis Question Set 1:

1. Is the reference in the environment the only reference to a fish?

2. Pat could have changed the `move` method so that it, like `breed`, would return a `boolean` value indicating whether or not the fish successfully moved. What advantages might there be to this design change? Are there any disadvantages?

Once the `breed` and `die` methods were implemented, I modified the `act` method based on the pseudo-code I had written earlier. I decided to implement the chance of dying, like the chance of breeding, as a `double` instance variable in the range from 0.0 to 1.0, and initialize it in the `initialize` method. The code, with modifications in boldface, is shown below. Then I was ready to test my modifications.

```
public void act()          // modified from Chapter 1!
{
    // 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();
}
```

Testing

With some help and advice from Jamie, I developed the black box test cases by starting with the test cases from the original program and modifying them to reflect the probabilities of breeding and dying. The list of modified test cases is below, with additions shown in boldface.

- An empty file or one that contains invalid data (such as just one dimension for the environment) should generate an error.
- A valid file with valid environment dimensions but no fish should result in an empty environment, but no errors.
- A file with a single fish should run with no errors. (The behavior of the fish will depend on its starting location.)
- A file with two or more fish in the same location should generate an error.
- A file with a fish in every location in the environment (but only one fish in every location) should run with no errors. **Even in the first timestep, some fish may move into spaces vacated by dying neighbors.**
- A fish with no empty locations around it should stay where it is.
- **A fish should attempt to breed approximately one-seventh of the time. If it has no empty neighboring locations, it does not breed successfully. When a fish breeds, it should breed into all of its empty neighboring locations. The newborn fish should not act during the timestep when they are created.**
- A fish with a single adjacent empty location in front or to the side of it should always move there **if it doesn't breed**.
- A fish with a single adjacent empty location behind it should stay where it is.
- A fish with two adjacent empty locations in front or to the side of it should **either breed (see above) or should move. If it moves, it should** move to either of its neighboring empty locations with equal likelihood, never staying where it is and never moving backward.
- A fish with three adjacent empty locations in front and to the side of it should **either breed (see above) or should move. If it moves, it should** move to any of its three neighboring empty locations with equal likelihood, never staying where it is and never moving backward.
- **Whether a fish breeds, moves, or stays still, it has a one in five chance of dying. In other words, in each timestep approximately one-fifth of the fish should die.**

I also analyzed the code in the `act`, `breed`, and `die` methods to identify additional tests. For example, test cases from `breed` include a fish that attempts to breed but has no valid, empty neighboring locations and fish that breed into 1, 2, 3, and 4 locations.

To get predictable results as I ran my tests, I seeded the random number generator with the same seed I had used when running the original test runs (see Exercise 1 in Exercise Set 5 of Chapter 2). Next, I turned on debugging at the beginning of the `step` method in the `Simulation` class (and restored it before returning) to help trace fish activity throughout the simulation. Finally, I ran the program with the `fish.dat` initial configuration file. The first thing I realized was that the debugging messages weren't as useful as they could have been; I hadn't added any debug messages to the `breed` or `die` methods. I decided to add debugging messages to my new methods so I could trace all simulation behavior more easily, not just fish movement.

After a number of timesteps it looked like the probabilities of moving in various directions were as I expected. I saw fish with only one potential new location move to that empty location, fish with two potential new locations move to either the first or second empty location with approximately equal likelihood, and fish with three potential new locations move to one of the three empty locations, again with approximately equal likelihood. I could also verify that fish that bred were breeding into all neighboring empty locations. It was a little harder, though, to determine whether fish were breeding and dying with the correct probabilities. I decided to turn off debugging for the move method to cut down on the number of trace messages and then run the simulation for more timesteps. I did this by adding a call to `Debug.turnOff` at the beginning of the `move` method and a call to `Debug.restoreState` at the end of the method. (The `turnOff` method is an additional `Debug` method that was not listed in the partial list of methods in Chapter 2. It does appear, however, in the Quick Reference and the full class documentation for the `Debug` class.) The results from my first test of five timesteps are below.

	1	2	3	4	5	Total
Number of fish actions	6	9	15	18	20	68
Number of breed attempts	1	2	3	3	3	12
Number of deaths	1	0	2	2	3	8

The percentage of fish in all timesteps that attempted to breed was 17.6% (12 attempts at breeding in 68 calls to the `act` method); the percentage that died was 11.8%. This was close to what I expected for the 1 in 7 chance of breeding (14.3%), although it did not correspond very well to the 1 in 5 chance of dying (20%). I ran the test again with 10 timesteps; the additional timesteps are shown below. This time the numbers were much closer to what I expected: 13.9% of fish in all timesteps attempted to breed (not always successfully), while 18.3% died.

	6	7	8	9	10	Total
Number of fish actions	26	31	31	36	38	230
Number of breed attempts	3	3	5	4	5	32
Number of deaths	4	8	7	6	9	42

Finally, I ran the test with 20 timesteps. In this test run, 14.2% of fish in all timesteps attempted to breed (not always successfully), while 19.6% died.

	11	12	13	14	15	16	17	18	19	20	Total
Number of fish actions	10	20	20	21	23	32	38	41	46	45	379
Number of breed attempts	5	2	2	2	6	9	5	6	5	5	53
Number of deaths	3	4	3	4	6	8	7	6	12	10	78

Analysis Question Set 2:

1. Would you have created separate `breed` and `die` methods? Why or why not?

2. Consider the design choice to put the test for whether a fish should attempt to breed in the `breed` method. If there are no empty neighboring locations to breed into, then a fish wouldn't be able to move either. Does that affect whether the decision was a good design choice? Should the test for whether the fish should die have been moved into the `die` method?

3. The call to `breed` could have been

   ```
   boolean bred = breed();

   if ( ! bred )
    ...
   ```

   What advantages or disadvantages do you see in these two ways of calling this method?

Exercise Set 1:

1. Run the marine biology simulation with the modified `Fish` class to see how the behavior has changed. (Use the class provided in the `DynamicPopulation` folder or follow the directions in `FishModsForChap3.txt`.)
    [Reminder: In the distributed version of the case study, the class containing the main method is `MBSGUI`.]

2. If you're running a user interface that has a "Save" function, save a copy of the results after 5 timesteps. Use the same seed you used in Chapter 2 (see Exercise 4 in Exercise Set 5). Give the file a descriptive name, like `chap3after5steps.dat`. Run the program for 10 timesteps and save the results in another file with a descriptive name. Compare the files you saved in Chapter 2 with the files you just saved. Are the fish movements the same? Why or why not?

3. Modify the `generateChild` method to construct children with random colors. Run the simulation again to see how the behavior has changed.

4. Modify the `initialize` method to take two new parameters representing the chance of breeding and the chance of dying. Modify the existing `Fish` constructors to pass the mathematical values `1.0/7.0` and `1.0/5.0` to `initialize`. The values are now hard-coded in the constructors rather than in `initialize`.

5. Implement a new `Fish` constructor that also takes two new parameters representing the chance of breeding and the chance of dying. Use `SimpleMBSDemo2` to test your new constructor. Construct fish with different probabilities of breeding and dying, and note what differences in behavior you observe. For example, what happens when you construct fish whose probability of breeding is 0.05 and probability of dying is 0.1? What happens when you construct fish whose probabilities of breeding and dying are both 0.0? Remember that if you are specifying colors in the driver (`MBSSimpleDemo2`), you will need to import `java.awt.Color`. (See Exercise 2 in Exercise Set 4 of Chapter 2.)

6. Introduce a new instance variable in `Fish` keeping track of how many times a fish bred. Initialize it in the constructor and increment it in the `breed` method. Modify the debugging statement in the `die` method to print the number of times the fish bred. Run your simulation for 20 timesteps. What is the maximum number of times a fish bred in your test run? What is the minimum number? Are these values you would have expected given the probability of breeding in each timestep?

7. Introduce a new instance variable keeping track of a fish's age. Initialize it in the constructor and increment it in the `act` method. Modify the debugging statement in the `die` method to print the age of the fish when it dies. Run your simulation for 20 timesteps. What is the oldest age at which a fish died in your test run? What is the youngest age at which a fish died? Are these values what you would have expected given the probability of dying in each timestep?

8. What if the chance of breeding and dying depended on the age of a fish? Change the fish's behavior so that it never breeds until it is 3 units old, after which it has a 1 in 3 chance of breeding. Its chance of dying, on the other hand, increases steadily with its age: at age 1 it has a 1 in 10 chance of dying, while at age 7 it has a 7 in 10 chance of dying.

Modified Quick Reference for Fish Class

This quick reference lists the constructors and methods associated with the modified Fish class described in this chapter. Public methods are in regular type. *Private and protected methods are in italics.* (Complete class documentation for the marine biology simulation classes can be found in the Documentation folder.)

```
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()
```

Marine Biology Simulation Case Study

Chapter 4

Specialized Fish

After I demonstrated the dynamic population version of the simulation to the marine biologists to make sure it met their needs, we talked about what modification should be made next. They wanted me to create several new kinds of fish with specialized patterns of movement to see what effect that would have on the simulation.

Problem Specification

The marine biologists decided that they would like to start by adding two new kinds of fish to the simulation: fish that dart forward whenever possible and slow-moving fish. These new types of fish would share all the attributes already defined for the `Fish` class except that their movement behavior would be different. In particular, the biologists decided that:

- A darter fish darts two cells forward if both the first and second cells in front of it are empty. If the first cell is empty but the second cell is not, then the darter fish moves forward only one space. If the first cell is not empty, then the darter reverses its direction but does not change its location. Like objects of the `Fish` class, darters never move in the same timestep as breeding.

- A slow fish moves so slowly that, even when it does not breed, 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. Like objects of the `Fish` class, slow fish never move in the same timestep as breeding and never move backward.

Design Issues

Although normal fish, darters, and slow fish exhibit different behavior when moving, they also share many similarities. For example, the `id`, `color`, `location`, and `direction` accessor methods have nothing to do with the particular type of fish being modeled. At the abstract level of the `act` method, deciding when to breed, when to move, and when to die, the different kinds of fish have the same behavior. The

particulars of how they breed and die are almost identical; the only difference is what type of fish they generate when breeding. The primary difference among these three kinds of fish, though, is how they move.

Rather than repeat the code for all the methods with the same behavior, I wanted to use *inheritance* to implement the specialized kinds of fish. My idea was to create new `DarterFish` and `SlowFish` subclasses that would *extend* the `Fish` class. This means that an object of the `DarterFish` (or `SlowFish`) class would inherit certain data and behavior defined in the `Fish` superclass, such as the accessor methods and the `act`, `breed`, and `die` methods. It could also *redefine* the behavior of some superclass methods by providing new implementations in the subclass. (This is also known as *overriding* the superclass method.) For example, the `DarterFish` class could redefine the `generateChild` method to create darter fish rather than normal fish, and redefine the `move` method (or its helper methods) to move in a different way.

One question I had was, if a darter or slow fish used the `act` method inherited from the `Fish` class, how would it know to use the redefined `generateChild` and `move` methods from the `DarterFish` or `SlowFish` class, instead of the methods from the `Fish` class? I decided to ask Jamie. The answer was *dynamic binding* (sometimes called *polymorphism*, according to Jamie). Here's how it works. The `Simulation` object asks a particular fish (darter, slow, or normal) to act. Let's say, for the sake of simplicity, that the object is a darter fish.

- The `DarterFish` class doesn't have an `act` method defined in it, so it inherits the generic `act` method from `Fish`. (We say that the call to `act` is *dynamically bound* to the `act` method in the `Fish` class.)
- The object that is acting, though, is still a `DarterFish` object. When the `act` method (inherited from `Fish`) calls `move`, it is actually the darter fish executing the `act` method that is invoking the `move` method on itself. The `DarterFish` class does have a `move` method defined in it, so that is the one that's executed. (The call to `move` is dynamically bound to the redefined `move` method in the `DarterFish` class.)
- Depending on how it is implemented, the redefined `move` method could call another internal method, like `location` or `emptyNeighbors`. We'll see that in `DarterFish` the redefined `move` method calls `nextLocation` (which it redefines) and `location`, `direction`, `changeLocation`, and `changeDirection` (which it does not redefine). This means that the call to `nextLocation` will be dynamically bound to the redefined method in the `DarterFish` class, but the calls to `location`, `direction`, `changeLocation`, and `changeDirection` will be dynamically bound to the inherited methods from the `Fish` class.

The picture below shows an object of the `DarterFish` class, with the redefined methods overlaying the methods they override. (`Protected` methods, which are meant to be used internally, are shown inside the box representing the `DarterFish` object.)

DarterFish
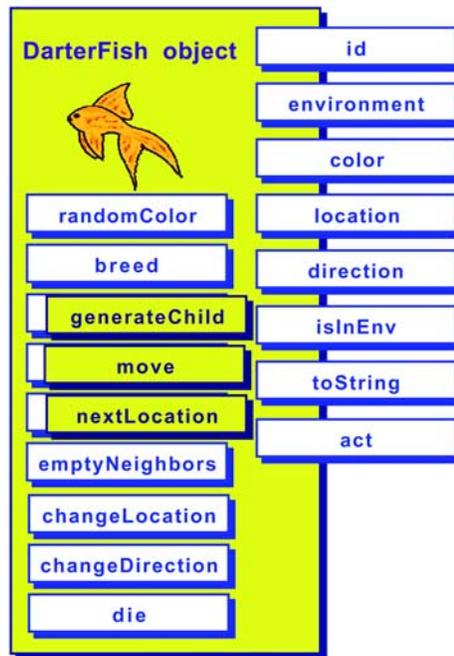
One of the assumptions in this explanation of dynamic binding is that the darter or slow fish has access privileges to the inherited methods (like `act` and `location`) and that inherited methods (like `act`) have access to redefined methods (like `move`). This would not be the case if these methods were private. For example, if the `move` method were private in `Fish` and `DarterFish`, then the `act` method in `Fish` would always use the `move` method from the `Fish` class, even for a darter fish, because that is the only `move` method to which it would have access. Similarly, if the `changeLocation` method were private in `Fish`, a redefined `move` method in a subclass would not be allowed to invoke it. The `protected` keyword allows inherited methods in superclasses to call methods that dynamically bind to methods in subclasses, and allows methods in subclasses to call inherited methods in superclasses. The `public` keyword would allow this also, but the `protected` keyword is an indication that the access is not meant to be open to all classes. Unfortunately, Java does not guarantee that objects of other classes, like the `Simulation` class, do not make use of `protected` methods. This meant that I needed to be very careful to check for myself that I used `protected` methods only in subclasses, as intended, and that I did not use them in client code.

Once I understood dynamic binding and how I should use protected methods in the marine biology simulation program, I felt comfortable implementing `DarterFish` and `SlowFish` as subclasses of the `Fish` class.

Darter Fish

Implementation of the `DarterFish` Class

The first thing I needed to do was create the empty `DarterFish` subclass, specifying that it extends the `Fish` class.

```
public class DarterFish extends Fish
{
}
```

A subclass inherits its superclass's data, can inherit or redefine its methods, and can define new data and methods. In the case of `DarterFish`, I knew I wanted to inherit most of the `Fish` methods but redefine the `move` method. According to the specification, a darter can only move forward. It moves two spaces forward if it can, and one space forward if it can't move two spaces. If it can't move at all, because the cell in front of it is not empty, then it reverses its direction without moving.

I decided that my first step would be to modify the `nextLocation` method, which defines how the fish chooses where to move. My redefined method finds the location in front of the darter (in other words, the neighbor of the current location in the same direction that the fish is facing) and the location in front of that (the neighbor of the one in front, in the same direction). The new `nextLocation` method then checks whether those spaces are empty. If neither location is empty, `nextLocation` returns the darter's current location because it was unable to move. (My logic for deciding when a darter is unable to move was incorrect, though, as I discovered later.)

The code below shows the first draft of my redefined `nextLocation` method, without debugging messages.

```
protected Location nextLocation()      // first draft!
                                       // (warning: buggy!)
{
    Environment env = environment();
    Location oneInFront = env.getNeighbor(location(), direction());
    Location twoInFront = env.getNeighbor(oneInFront, direction());
    if ( env.isEmpty(twoInFront) )
        return twoInFront;
    else if ( env.isEmpty(oneInFront) )
        return oneInFront;
    else
        return location();      // can't move, stay in
                                // current location
}
```

I also needed to write a new `move` method for `DarterFish`, so that if the darter did not change location, then it reversed its direction. Here is the new `move` method without debugging messages.

```
protected void move()
{
    // Find a location to move to.
    Location nextLoc = nextLocation();

    // If the next location is different, move there.
    if ( ! nextLoc.equals(location()) )
    {
        changeLocation(nextLoc);
    }
    else
    {
        // Otherwise, reverse direction.
        changeDirection(direction().reverse());
    }
}
```

The `move` method for `DarterFish` was simple to write because the logical structure is the same as the `move` method for `Fish`. The code is simpler, though, because when a darter moves, its direction does not change.

I also needed to write one or more constructors for the `DarterFish` class, because constructors are not inherited like other methods. Each class must explicitly define its own constructors. The original `Fish` class has three constructors: one that specifies the environment and initial location, one that also specifies the initial direction, and a third that specifies environment, location, direction, and a color. For testing purposes I decided to make all darter fish yellow. This was fine with the biologists; they had primarily been using color to make the simulation more interesting, not to represent meaningful information. At first I thought that because all darters would be the same color, I didn't need to provide the third constructor. Then I decided to go ahead and provide it anyway, in case the biologists wanted to define darters with other colors later.

Below is the code for the first constructor. The only thing it has to do is to call the appropriate superclass constructor (using the `super` keyword) to initialize the attributes inherited from `Fish`, such as the location, direction, and color. The expression to get a random direction is the same as that found in the two-parameter `Fish` constructor. The code for the other constructors is practically the same, so it is not shown here.

```
public DarterFish(Environment env, Location loc)
{
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, env.randomDirection(), Color.yellow);
}
```

Finally, I redefined the generateChild method to construct a new DarterFish rather than a new Fish object, as shown below (without the debugging message). Everything else about the method is the same. With the redefined generateChild method, I did not need to copy or redefine the rest of fish breeding behavior but could inherit it from the Fish class.

```
protected void generateChild(Location loc)
{
    // Create new fish, which adds itself to the environment.
    DarterFish child = new DarterFish(environment(), loc,
                        environment().randomDirection(), color());
}
```

Analysis Question Set 1:

1. What is the logic error in Pat's first draft of the nextLocation method?

2. A darter can only swim east and west or north and south. How might we change the darter so that it usually continues east and west, or north and south, but occasionally switches?

Testing the DarterFish Class

The new code for the DarterFish class seemed pretty simple, so I thought I would test it by just running it a number of times and seeing if the behavior seemed right. It did, and I was about to move on to the SlowFish class when I noticed something odd. One of the darters had hopped right over another fish! I ran the program a few more times and saw the same behavior again.

I turned on debugging at the beginning of the step method in the Simulation class (and restored it before returning) to help trace fish activity throughout the simulation. To help clarify when and where darters were moving and when they were blocked, I added some debugging statements to the nextLocation method in DarterFish. I ran the program and analyzed my results more carefully and realized that there were two different situations that could cause these results, one of which was an error. One situation was that the darter could have moved forward through an empty cell to get to the second cell, and then another fish could have slipped from the side into the first empty cell. This would be acceptable behavior, according to the program specification.

The second situation was that the darter could have hopped over a fish to get to an empty cell beyond it, because I had written `nextLocation` incorrectly. The code I had written did not check that **both** the cell immediately in front of the darter and the cell beyond that were empty. The corrected code appears below.

```
protected Location nextLocation()        // corrected code!
{
    Environment env = environment();
    Location oneInFront = env.getNeighbor(location(), direction());
    Location twoInFront = env.getNeighbor(oneInFront, direction());
    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.
    return location();
}
```

Because I had almost missed the error in `nextLocation`, I decided to go back to testing more thoroughly, as the original programmer had done. I first developed black box test cases to test the `DarterFish` class, basing them on the test cases for the `Fish` class. Only the new or modified test cases are shown below.

- A file with a single darter fish should run with no errors. (The behavior of the fish will depend on its starting location and direction.)
- A file with two or more darter fish, or with one darter and one non-darter fish, in the same location should generate an error.
- A file with either normal or darter fish in every location in the environment (but only one in every location) should run with no errors. Whether they may move or not depends on whether breeding and dying have been implemented (see the appropriate test case from either Chapter 2 or Chapter 3). Any darters that do not move should reverse direction.
- A darter that does not breed and that has two empty neighboring locations in front of it should always move forward two spaces. A darter that does not breed and that has only one empty neighboring location in front of it should always move into that location. A darter that does not breed and does not change location should reverse its direction. This leads to a visual pattern that is easy to spot — darter fish appear to pace back and forth in the bounded environment.
- All darter fish should be yellow. (This is actually based on an implementation decision, not on the original problem specification.)

I then considered the code in the new `move` and `nextLocation` methods to see if I needed to develop additional test cases. It seemed that the black box cases listed above would cover the new code. I decided to start my testing by rerunning my previous tests with normal fish to verify that the results were the same and that my modifications had not broken working code. This kind of testing is known as *regression testing*.

As I had with my previous test runs, I seeded the random number generator to get predictable results (see Exercise 1 in Exercise Set 5 of Chapter 2). For the regression tests it was important to use the same seed, and there was no reason to change the seed for the new tests. I had already turned on debugging to find the bug in `nextLocation`. Finally, I made a copy of `fish.dat` and changed the six fish to six darter fish. I thought that if I used the same seed and the same initial configuration of fish, then I would see the same behavior for breeding and dying, although the movement would be different. Then I ran the program. This time the darters moved exactly where I expected them to move in each timestep.

To test breeding and dying, which are probabilistic, I needed to keep statistics for a number of timesteps, as I had done for the normal fish. My results are below.

	1	2	3	4	5	6	7	8	9	10	Total after 10	Total after 20
Number of darter actions	6	9	11	16	16	19	23	31	42	45	218	733
Number of breed attempts	1	1	2	1	2	2	4	6	7	4	30	114
Number of deaths	1	1	3	2	1	2	3	6	11	9	39	139

After ten timesteps, the percentage of darters in all timesteps that had attempted to breed was 13.8% (30 attempts at breeding in 218 calls to the `act` method); the percentage that had died was 17.9%. This corresponded reasonably well to the 1 in 7 chance of breeding (14.3%) and the 1 in 5 chance of dying (20%) specified in the problem description. I continued the test up to 20 timesteps. This time the percentages were 15.6% for breeding and 19% for dying.

What surprised me, though, was that the numbers recorded for breeding and dying darter fish were different from the earlier tests with normal fish, even though the probabilities remained the same, the initial configuration was the same (except for the name of the class), and I had used the same seed. I realized that the difference was that the original breeding and dying fish use random numbers for movement as well as for breeding and dying, but darters do not. Consequently, the random numbers used for breeding and dying are different for the two populations of fish.

Exercise Set 1:

1. Draw two diagrams illustrating each of the situations Pat discovered in which a darter could hop over, or appear to hop over, another fish.

2. Run the marine biology simulation with the `darter.dat` and the `darterAndNormalFish.dat` initial configuration files to see how the behavior is different. (The difference is more obvious if you use the original `Fish` class rather than the one from the breeding and dying chapter or if the probabilities of breeding and dying are both set to zero. Or you can temporarily comment out the lines of code in the `Fish act` method that deal with breeding and dying.) *[Reminder: In the distributed version of the case study, the class containing the main method is MBSGUI. You can edit `MBSGUI.java` and follow the directions in the comments to make darter fish appear different from normal fish or to include darter fish as an option when creating a new environment using the graphical user interface.]*

3. If you're running a user interface that has a "Save" function, run the simulation with the `fish.dat` configuration file and save a copy of the results after 5 timesteps. Use the same seed you used in Chapter 3 (see Exercise 2 in Exercise Set 1). Give the file a descriptive name, like `chap4after5steps.dat`. Run the program for 10 timesteps and save the results in another file with a descriptive name. Compare the files you saved in Chapter 3 with the files you just saved. Are the fish movements the same? Why or why not? Run the program with the `darter.dat` configuration file and save the results in a file for future regression testing. (Be sure to give the file a name that will allow you to identify it later.)

4. If you have added constructors to the `Fish` class in addition to the three original constructors from Chapter 2, analyze which of these constructors should be added to the `DarterFish` class as well. Add them.

5. Redefine the `toString` method in `DarterFish` to clarify that this is a darter. This makes it easier to keep track of darters and normal fish in the debugging output. Turn on debugging in the `step` method in `Simulation`, as you did in Chapter 3, and run the simulation again. This will let you observe the changed behavior at a greater level of detail.

6. Choose a different seed for the random number generator and rerun your tests. What effect does this have on the behavior of the simulation?

7. The darters always move east and west or always move north and south. Create a subclass of the `DarterFish` class, called `TurningDarter`, that behaves like `DarterFish` except that there is a probability of 0.1 that a turning darter turns right or left (each with equal probability) before it tries to move forward. To use the `Random` class, you will need to import `java.util.Random`.

Slow Fish

Implementation of the `SlowFish` Class

The behavior of a slow fish is very similar to the behavior of a normal fish, so again I knew I wanted to inherit most of the `Fish` methods but redefine how (and when) it moved. According to the specification, a slow fish 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, however, it moves just like any other fish of the `Fish` class.

To test whether the fish should move, I knew I would randomly pick a number and compare it to the probability of moving. The first design decision I had to make, though, was whether to put that test in the `move` method or in the `nextLocation` method. If I put it in the `move` method, then 4 out of 5 times it would do nothing. If I put it in the `nextLocation` method, then 4 out of 5 times it would return the current location. Since it seemed like the test could go in either place, I decided to put it in the lower-level, more specific method, `nextLocation`.

I decided to store the probability of moving, the mathematical value 1/5 represented as a `double`, in an instance variable, just as I had with the probabilities of breeding and dying. Then I redefined the `nextLocation` method to pick a `double` randomly in the range of 0.0 to 1.0 (using the `nextDouble` method in `java.util.Random`), and compare it to the instance variable representing the probability of moving. If the randomly chosen number is less than the probability, the slow fish chooses a new location in the usual way, otherwise it returns the current location. To choose a new location, the slow fish calls `super.nextLocation()`, which executes the `nextLocation` method in the `Fish` superclass. (Without the `super` keyword, the call to `nextLocation` in the first `return` statement would be a recursive call to the `nextLocation` method in `SlowFish`. The `super` keyword forces the call to use the inherited `nextLocation` method, which in this case is defined in `Fish`.)

The code below shows the new instance variable and the redefined `nextLocation` method without debugging messages.

```
// Instance Variables: Encapsulated data for EACH slow fish
private double probOfMoving;    // defines likelihood in each
                                // timestep

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
        return location();
}
```

The `SlowFish` class also needed new constructors, not only because constructors aren't inherited like other methods, but also because I needed to initialize the `probOfMoving` instance variable. Each `SlowFish` constructor calls the four-parameter superclass constructor (using the `super` keyword) to initialize the instance variables inherited from `Fish`. To make slow fish easier to spot when testing, I decided to make them red. The code below shows the two-parameter `SlowFish` constructor; the other constructors are similar.

```
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
}
```

Finally, I redefined the `generateChild` method to construct a new `SlowFish` rather than a new `Fish`, just as I had for `DarterFish`.

Testing the `SlowFish` Class

As usual, I developed the black box test cases for the `SlowFish` class based on the test cases for the `Fish` class. Only the new or modified test cases are shown below.

- A file with a single slow fish should run with no errors. (The behavior of the fish will depend on its starting location.)
- A file with two or more slow fish, or with one slow and one other fish, in the same location should generate an error.
- A file with a fish in every location in the environment (but only one in every location) should run with no errors, regardless of the types of the fish. Whether they may move or not depends on whether breeding and dying have been implemented (see the appropriate test case from either Chapter 2 or Chapter 3).
- A slow fish that does not breed and that has one or more empty neighboring locations in front of it or to its sides should move to one of its neighbors approximately 20% of the time. In other words, in each timestep approximately one fifth of the slow fish that don't breed should move. When a slow fish does move, it should have an equal probability of moving to each of its valid empty neighbors.

I then considered the code in the `SlowFish nextLocation` method to see if I needed to develop additional test cases. It seemed that my final black box case would cover the new code. I would still need to include all the test cases for normal fish when testing

slow fish, though, because of the call to `super.nextLocation()`. As usual, I decided to start with regression testing (rerunning my previous tests) to verify that the results were the same and that my modifications had not broken working code.

To help clarify why various fish failed to move, I added a debugging statement to the `nextLocation` method in `SlowFish` to notify me when fish appeared not to move because they were moving too slowly. Finally, I developed a new initial configuration file that contained seven normal fish and seven slow fish. Then I ran the program with the seeded random number generator. My results are below.

	1	2	3	4	5	6	7	8	9	10	Total after 10	Total after 20
Number of slow fish actions	6	5	5	9	11	17	15	16	15	20	119	523
Number of breed attempts	0	0	1	1	2	1	2	1	4	4	16	76
... that were successful	0	0	1	1	2	0	1	1	4	4	14	64
Number of calls to `move`	6	5	4	8	9	17	14	15	11	16	105	459
... that attempted to move beyond cell	2	0	0	1	0	2	2	1	5	4	17	94
... but were blocked	0	0	0	0	0	0	0	0	1	1	2	22
Number of deaths	1	0	0	1	1	2	2	2	5	3	17	96

After ten timesteps, the percentage that had attempted to breed was 13.4% (16 attempts at breeding in 119 calls to the `act` method). Only 14 of the 16 attempts were successful; two fish did not breed because there were no empty neighboring locations. This led to 105 move attempts. Of these, 16.2% attempted to move out of their current cell (17 out of 105), which was a little lower than I expected, given the 1 in 5 chance of trying to move. The number of fish that actually moved was somewhat lower than the number that attempted to move; twice there were fish that were blocked from moving by other fish in neighboring locations. The percentage of slow fish that died in the first ten timesteps was 14.3%, which was also low.

I continued the test up to 20 timesteps. This time the numbers (shown in the table above) were much closer to what I expected.

> ***Analysis Question Set 3:***
>
> 1. In Chapter 3, Pat added breeding and dying behavior by modifying the `Fish` class. Another alternative would have been to create a subclass of `Fish` with breeding and dying behavior, leaving `Fish` unchanged. What are the advantages and disadvantages of the two alternatives? What would be the impact on `DarterFish` and `SlowFish`?
>
> 2. A method is *deterministic* if, given the inputs to it, you can tell exactly what its result will be. A method is *probabilistic* if, given the inputs, there are various probabilities of different results. Of the `nextLocation` methods in `Fish`, `DarterFish`, and `SlowFish`, which are deterministic? Which are probabilistic?

Exercise Set 2:

1. If you have added constructors to the `Fish` class in addition to the three mentioned above, analyze which of these constructors should be added to the `SlowFish` class as well. Add them.

2. Run the marine biology simulation with the `slowAndNormalFish.dat` and `3species.dat` initial configuration files to see how the behavior has changed. (Again, you may find it easier to see the differences between types of movement without breeding and dying behavior.)

3. Redefine the `toString` method in `SlowFish` to clarify that this is a slow fish. This makes it easier to keep track of slow and normal fish in the debugging output. Turn on debugging in the `step` method in `Simulation`, as you did in Chapter 3, and run the simulation again. This will let you observe the changed behavior at a greater level of detail.

4. Choose a different seed for the random number generator and rerun your tests. What effect does this have on the behavior of the simulation?

5. Using Pat's table of test run results, which may or may not match your own results, calculate the following for 20 timesteps.
   - What percentage of slow fish attempted to breed in each timestep? (We're interested in the average over all 20 timesteps, not in the actual percentage for any one timestep.) Does this percentage correspond with the specified 1 in 7 chance of breeding?
   - Consider the number of times the `act` method was called for slow fish over those 20 timesteps, the number of fish that tried to breed, and the number that bred successfully. Given these values, is the number of calls to the `move` method what you would expect? In other words, is `move` being called the correct number of times?
   - How many times was the `nextLocation` method in `SlowFish` called over the 20 timesteps? (Under what conditions is `nextLocation` called?)
   - How many times was the `nextLocation` method in `Fish` called over the 20 timesteps? (Under what conditions is the `Fish nextLocation` method called?)
   - Of the slow fish that did not breed, what percentage moved too slowly to attempt to leave their cells? What percentage attempted to move beyond their own cell (either successfully or unsuccessfully)? What percentages would you expect, given the problem specification? Do the actual results correspond to the expected results?
   - On average, what percentage died? What percentage would you expect, given the problem specification? Does the actual result correspond to the expected result?
   - Compared to the test results for 10 timesteps, are the actual results over 20 timesteps closer to the expected results as Pat claimed?

6. A slow fish moves in each timestep, even when it doesn't move far enough to leave its current cell. As it moves slowly in its own cell, it may change direction. Modify the `SlowFish` class so that even when it doesn't move outside its cell it may still turn right or left (or continue in its current direction).

7. Implement breeding and dying behavior by creating a subclass of the original `Fish` class. (See Question 1 in Analysis Question Set 3 above.)

8. Define a new `CircleFish` subclass of `Fish` that constantly swims in a circle (as much as is possible in a rectangular grid). In each timestep, the circle fish moves to the location forward and to the right, on a diagonal from its current location, if possible. It also changes its direction by turning 90 degrees to the right. If the fish cannot move as described above, it stays in its current location, but still turns 90 degrees to the right. Make the constructor give all circle fish the same color, so they can easily be distinguished from other fish.

9. Refine the `CircleFish` class to make the movement look more like a circle. In the first timestep, a circle fish moves forward one location (if possible), without turning. During the next timestep, it moves to the location forward and to the right (if possible), as described in Exercise 8. If the fish cannot move, it stays in its current location, but turns 90 degrees to the right. After the fish has moved and turned, or just turned without moving, its next movement will be to move forward one location. The fish continually alternates these moves (except when it is unable to move and only turns).

Quick Reference for Specialized Fish Subclasses

This quick reference lists the constructors and methods associated with the specialized fish classes, DarterFish and SlowFish, introduced in this chapter. Public methods are in regular type. *Private and protected methods are in italics.* (Complete class documentation for the marine biology simulation classes can be found in the Documentation folder.)

```
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()
```

Marine Biology Simulation Case Study

Chapter 5

Environment Implementations

About a month after my summer job with the marine biologists ended, they contacted me to ask if I would return to work part time on another task. They wanted the simulation enhanced to support unbounded environments as well as bounded ones. For example, if the biologists were to study fish movement in a specific portion of the Pacific Ocean, then a bounded, two-dimensional environment would not be an appropriate model. In the real world, fish would swim in and out of the area they were studying.

Over the summer, I had not studied how the environment was implemented, so I decided to ask Jamie for an overview of the existing code before I got started. The first part of this chapter is Jamie's explanation of the `Environment` interface and `BoundedEnv` implementation. The second half of the chapter describes an implementation class that I wrote called `UnboundedEnv`.

Overview of the Existing Design and Code

An `Environment` object models a rectangular grid that contains objects at various grid locations. Although the original program only modeled bounded rectangular environments, the author of the code knew that the marine biologists might want to model unbounded environments in the future.

According to Jamie, the original programmer considered three ways to represent an environment internally.

- The internal data representation for an environment could be a bounded, rectangular data structure, such as a two-dimensional array, corresponding to the grid-like environment being modeled. Objects could be located in the data structure based on their location in the simulated environment. For example, the following two-dimensional grid represents a tic-tac-toe game in progress.

O		O
	X	

- Alternatively, an environment could model objects in a conceptual grid by storing the objects and their locations in a list. For example, the tic-tac-toe game above could be represented as the following list of object, location pairs.

O, (0, 2)	X, (1, 1)	O, (0, 0)

- If the objects being stored in the environment were all identical, the internal representation could just be a list of the locations, without storing the objects at all, as shown on the right in the following diagram. This representation would not work for the tic-tac-toe game described above. If the locations of walls in a small maze were being modeled, however, any of the three representations below might be used.

	Wall	Wall
	Wall	

Wall, (0, 1)	Wall, (0, 2)	Wall, (1, 1)

(0, 1)	(0, 2)	(1, 1)

The original simulation programmer rejected the third model (because the fish being represented in the environment are not identical) and decided to implement the fish environment using the first of these representations — a two-dimensional array corresponding to the grid. Knowing, however, that the marine biologists might want to model other types of environments in the future, the original programmer decided to create `Environment` as an *interface*. The two-dimensional array representation, which the original programmer called the `BoundedEnv` class, is just one possible implementation of the interface. (Chapter 2 introduced the concept of *interfaces*, and a more detailed description is in the next section.)

Analysis Question Set 1:

1. What are the relative merits of the first two internal representations described above for each of the following kinds of simulation?
   - A simulation with a relatively small bounded environment.
   - A simulation with a very large bounded environment and many objects in it.
   - A simulation with a very large bounded environment but very few objects in it.
   - A simulation of an environment whose boundaries are not known or are irregular.

2. What representation would be appropriate if more than one object could occupy each location in the environment?

The `Environment` Interface

An interface looks a little like a class, but it only specifies *what* methods should be implemented without actually implementing them. Because an interface doesn't have any instance variables and doesn't have code for its methods, you can't create objects of an interface directly. Instead, a program that uses an interface needs to also have at least one class that *implements* the interface. In other words, it is necessary to create a class that provides implementations for all the methods that the interface specifies. Any object of the implementation class satisfies the interface and can be used where an object of the interface type is expected. Often a program has several classes that implement an interface, each in a different way. Those classes may have additional methods as well as any instance variables they need for the implementation of their methods.

For the marine biology simulation, the `Environment` interface specifies what methods any environment class needs to provide. The `BoundedEnv` class is one implementation of that interface. The `Simulation` and `Fish` classes could refer directly to objects of the `BoundedEnv` class, but then they would have to be rewritten if the implementation were to change. Instead they always refer generically to the `Environment` interface and stick to the methods specified there. Their parameters and instance variables of type `Environment` actually refer to an object of some class that implements the interface. That object is constructed in the driver (user interface or main function), which is the only place that must explicitly specify the environment's implementation class.

The full interface is shown below, with documentation comments removed. It includes the methods mentioned in Chapter 2 and some additional ones. All interface methods are `public`.

```
public interface Environment
{
  // accessor methods for determining environment dimensions
  int numRows();
  int numCols();

  // accessor methods for navigating around this environment
  boolean isValid(Location loc);
  int numCellSides();
  int numAdjacentNeighbors();
  Direction randomDirection();
  Direction getDirection(Location fromLoc, Location toLoc);
  Location getNeighbor(Location fromLoc, Direction compassDir);
  ArrayList neighborsOf(Location ofLoc);

  // accessor methods that deal with objects in this environment
  int numObjects();
  Locatable[] allObjects();
  boolean isEmpty(Location loc);
  Locatable objectAt(Location loc);
```

```
    // modifier methods
    void add(Locatable obj);
    void remove(Locatable obj);
    void recordMove(Locatable obj, Location oldLoc);
}
```

Many of these methods were described in Chapter 2 but a few are new, including the first two methods, `numRows` and `numCols`. If the environment is bounded, `numRows` and `numCols` return the dimensions of the environment. Otherwise, they return -1, indicating that the number of rows and number of columns are not fixed.

The second set of methods deals with navigating around the environment from cell to cell. The `isValid` method tests whether a cell location is valid; for example, whether it is in the bounds of a bounded environment. The `numCellSides` method defines the shape of the cells in the environment, while the `numAdjacentNeighbors` method specifies how many neighbors there are around each location. For example, a cell in an environment with square cells has four sides and, by default, four adjacent neighbors. A hexagonal cell, on the other hand, would have six sides and, generally, six neighbors. The last four methods in this set are for navigating from a particular cell in the environment. The `randomDirection` method, mentioned in Chapter 2, returns one of the directions that leads from an environment cell to its adjacent neighbors. The `getDirection` method returns the direction to go to get to a particular neighboring location, `getNeighbor` returns the neighboring location in a given direction, and `neighborsOf` returns a list of all the immediate neighboring locations of the given location.

The next set of methods provides access to the objects in an environment or indicates whether a particular cell in the environment is empty. All objects in the environment must be `Locatable`, meaning that they must be instances of classes that implement the `Locatable` interface (in other words, they must provide a `location` method). The `numObjects` method returns the number of `Locatable` objects in the environment, `allObjects` returns those objects in an array, the `isEmpty` method indicates whether a particular location in the grid is empty, and `objectAt` returns the object in a given location if it is not empty. If the location is empty, `objectAt` returns a `null` reference.

Finally, there are three methods that modify the environment: `add`, which adds a new object to the environment; `remove`, which removes an object from the environment; and `recordMove`, which records the fact that an object moved from one location to another. The `recordMove` method is necessary to ensure that the environment and the `Locatable` objects in it are in a consistent state when an object moves.

Analysis Question Set 2:

1. Consider using an environment as a container for objects other than fish, such as band members marching on a field, physics particles moving in a small space, or blocks of colored material in a quilt. What are the requirements for the classes representing the band members, physics particles, or color blocks in order to put objects of those classes into an environment?

2. `Location` is another class that implements an interface, the standard Java `Comparable` interface. Research the `Comparable` interface to discover what methods it includes. When would it be useful to specify that an object or a parameter must be `Comparable`?

3. In Chapter 2, Pat said that `EnvDisplay` is also an interface. What are the benefits of making `EnvDisplay` an interface?

4. The `step` method in the `Simulation` class gets a list of `Locatable` objects from the `Environment allObjects` method and then casts each of them to a `Fish`. It does this because the `Locatable` interface does not have an `act` method, but `Fish` does. Consider an alternative design that includes an `Actable` interface, with a single `act` method. In this design, the `Fish` class would implement both `Locatable` and `Actable`. The `step` method in `Simulation` would cast the `Locatable` objects to `Actable` before invoking the `act` method. What are the advantages and disadvantages of the two designs?

5. Another alternative design, in addition to the one in Exercise 4, would be to add the `act` method to the `Locatable` interface. What are the advantages and disadvantages of this design, compared to the other two?

The `BoundedEnv` Class (and `SquareEnvironment`)

The first thing Jamie pointed out about the `BoundedEnv` class is that it `extends` (is a subclass of) another class called `SquareEnvironment`. `SquareEnvironment` is an *abstract* class implementing the `Environment` interface, meaning that it does not implement all of the methods specified by `Environment`. Just as one can't create an instance of an interface, one can't create an instance of an abstract class either, since both have unimplemented methods.

The `SquareEnvironment` abstract class implements only those `Environment` methods needed to define an environment of square cells whose neighbors are to the north, south, east, and west. These methods are `numCellSides`, `numAdjacentNeighbors`, `randomDirection`, `getDirection`, `getNeighbor`, and `neighborsOf`. The only "navigational" method that `SquareEnvironment` does not implement is the `isValid` method, which tests the validity of a location, since that depends in part on other factors, such as whether the environment is bounded. `SquareEnvironment` implements none of the other `Environment` methods, so those must be implemented by subclasses such as `BoundedEnv`. Even though it is abstract, `SquareEnvironment` does have two constructors. The first constructor creates an environment in which there are four adjacent neighbors around each cell, each sharing one of the cell's sides. The second constructor provides a way to create an environment with either four adjacent neighbors around each cell or eight (the four that share sides with the cell and the four on the diagonals, which share only a single point with the cell.)

Jamie told me that I would be able to use `SquareEnvironment` for my unbounded environment without any modifications, so I could treat it as a black box class. I decided to do this, studying the class documentation for `SquareEnvironment`, but not its implementation.

Instance variables, constructor, and the simplest accessor methods — `numRows`, `numCols`, `numObjects`, **and** `isValid`:
Unlike the `Environment` interface, the `BoundedEnv` class has instance variables and a constructor. `BoundedEnv` implements the conceptual two-dimensional grid of the environment as a two-dimensional array. It also keeps a separate count of the number of objects in the grid. The instance variables for `BoundedEnv` appear below.

```
private Locatable[][] theGrid;   // grid representing the environment
private int objectCount;         // # of objects in current environment
```

The two sets of square brackets in the declaration for `theGrid` indicate that this is a two-dimensional array. To refer to a specific element (or, more precisely, to a `Locatable` object at a specific location) in the 2-D array, we use two indices that indicate the row and column within the array, respectively. For example, the expression `theGrid[r][c]` refers to the `Locatable` object in row `r` and column `c` of the array. Both rows and columns start at 0. Thus, the object in the first row and first column is referred to internally as `theGrid[0][0]`; the object in the third row and the seventh column is referred to as `theGrid[2][6]`. To construct a two-dimensional array, we specify the number of rows and number of columns in square brackets, as in the constructor below. (The constructor, like those in `DarterFish` and `SlowFish`, calls `super` to initialize any inherited attributes in `SquareEnvironment`. In this case the call isn't required, because the default constructor would be called automatically, but it makes the behavior of the constructor clearer.)

```java
public BoundedEnv(int rows, int cols)
{
    super();

    theGrid = new Object[rows][cols];
    objectCount = 0;
}
```

Three of the next four accessor methods in `BoundedEnv` are fairly straightforward. The `numRows` and `numCols` methods use standard Java constructs for determining the number of rows and columns in a two-dimensional array. The number of rows is written `theGrid.length`, while the number of columns is `theGrid[0].length` (in other words, the length of the first row). This only works if there is a first row; otherwise, `theGrid[0]` is undefined. Fortunately, the precondition on the `BoundedEnv` constructor states that the number of rows must be greater than zero. The `numObjects` method is even simpler than `numRows` and `numCols`; it merely returns the `objectCount`.

```java
public int numRows()
{
    return theGrid.length;
}

public int numCols()
{
    return theGrid[0].length;
}

public int numObjects()
{
    return objectCount;
}
```

The `isValid` method is slightly more complicated. First of all, methods that check the validity of a parameter should be able to accept a full range of both valid and invalid values. A null reference is one kind of invalid location; a location that is out of bounds is another. Valid locations are in the bounds of the environment, which are the same as the bounds of the internal 2-D array.

```
public boolean isValid(Location loc)
{
    if ( loc == null )
        return false;

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

Accessor methods that deal with a single location — `objectAt` and `isEmpty`:
The `objectAt` method, shown below, receives a location as a parameter, `loc`, and returns either the object at that location or a null reference if the location is not valid or if no object exists at location `loc`. First `objectAt` checks that the location is in the bounds of the environment, using the `isValid` method described above. If the location is valid, `objectAt` returns the object at the grid location indicated by `loc`'s row and column attributes.

```
public Object objectAt(Location loc)
{
    if ( ! isValid(loc) )
        return null;

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

The `isEmpty` method is similar to `objectAt`; it takes a location as a parameter and returns `true` or `false` depending on whether the location is empty or contains an object. Like `objectAt`, `isEmpty` first checks whether the location parameter is in the bounds of the environment. An out-of-bounds location causes `isEmpty` to return `false` because, although it does not contain an object, it is not a valid empty location in the environment. If the location is valid, `isEmpty` calls `objectAt`; if `objectAt` returns `null`, then there is no object at the location and `isEmpty` returns `true`. Otherwise, it returns `false`.

```
public boolean isEmpty(Location loc)
{
    return isValid(loc) && objectAt(loc) == null;
}
```

Searching through the grid — `allObjects` and `toString`:
The `allObjects` method, shown below, creates a list of all the objects in the environment by stepping through the entire grid. Since it knows how many objects it should find in the environment, it can store them in an array (`theObjects`), which it initializes to be the correct size. It also creates a second local variable (`tempObjectCount`) to keep track of how many objects have been copied over to the array so far; this serves as an index into `theObjects` as it is filled.

The `allObjects` method uses two nested `for` loops. This is a common pattern for traversing a two-dimensional, indexed data structure. These loops check all the locations for objects to be added to the array. The outer loop steps through all the rows in the grid. For each row, the inner loop steps through all the column positions. (This is called *row-major order*.) Inside the inner loop is the code that must be executed for each cell in the grid. First, `allObjects` retrieves the contents of the location indicated by the row and column. If there is an object at that location, it is added to the array of objects to be returned.

```
public Object[] 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;
}
```

The `toString` method creates a single string representing all the objects in the environment. It could have the same nested `for` loops as `allObjects`, but instead it calls that method, letting it do the work of traversing the two-dimensional data structure. The `toString` method then steps through the array created by `allObjects` to generate the string.

```
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;
}
```

Analysis Question Set 4:

1. Would an array be a good choice of data representation for the list created by `allObjects` if it did *not* know the number of objects in the environment right from the start? Why or why not?

2. Read the documentation for the accessor methods. Do the `allObjects` or `toString` methods guarantee the order in which they return the environment objects?

Modifying an environment — `add`, `remove`, and `recordMove`:
The `add` method adds an object, `obj`, to a specified location, `loc`.

```
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++;
}
```

First, `add` verifies the precondition from the documentation comment (not shown) that `loc` is a valid, empty location. A precondition is a condition that must be met when the method is called in order for it to work as expected. Technically, making sure a precondition is met is the responsibility of the calling method, but the original simulation programmer chose to check the preconditions in `add`, `remove`, and `recordMove`. If `add` didn't check this condition, the new object would simply replace the existing object and the program would continue as if no error had occurred. Rather than allow the program to run with missing data, `add` throws an exception to notify the user of the error when the precondition is not met. `IllegalArgumentException` is a standard Java exception class whose objects represent run-time program errors. These are serious errors that the program cannot handle and is not meant to handle, so throwing the run-time exception causes the program to halt. Not all anomalous conditions are errors that should cause the program to stop running, though. For example, a fish at the edge of the environment might pass out-of-bounds locations to `isEmpty` when it is trying to find its empty neighbors. In this case, it is perfectly valid to have an invalid location as a parameter.

If the location is empty, as it should be, `add` inserts the object into the grid at the row and column indicated by the location. The `add` method then increments the count of how many objects are in the environment.

The `remove` method reverses the task accomplished by the `add` method. The precondition for `remove` is that the object to be removed must be in the environment, but the check in the code is actually more explicit — the object must be in the environment at the correct location or else the method throws an exception. Whereas the body of `add` puts the new object in the grid and increments `objectCount`; the body of `remove` puts a null reference in the grid and decrements `objectCount`.

```java
public void remove(Locatable obj)
{
    Location loc = obj.location();
    if ( objectAt(loc) != obj )
        throw new IllegalArgumentException("Cannot remove " +
            obj + "; not there");

    theGrid[loc.row()][loc.col()] = null;
    objectCount--;
}
```

Finally, there's the `recordMove` method, which should be called by any `Locatable` object when it moves. The purpose of `recordMove` is to ensure that the environment and the object are in a consistent state; in other words, that they agree on where the object is. The method takes two parameters: the moving object, `obj`, and its old location. Its precondition is slightly more complicated than the others: the object's new location (`obj.location()`) must be a valid location in the environment, and there shouldn't be any *other* object there. The object itself should either be in that environment location already or the environment should move it there. In either case, at the end of the method the object and the environment should be in a consistent state.

If the object did not in fact move (the old location is the same as the new location), then recordMove doesn't have to do anything. Otherwise, recordMove double-checks that the object is still at the old location and that there isn't any other object at the new location. If everything is okay, the method moves the object by putting it (or, actually, a reference to it) into the grid at the row and column specified by the new location (obj.location()), and by putting a null reference into the old location. The code, with comments removed, appears below.

```java
public void recordMove(Locatable obj, Location oldLoc)
{
    Location newLoc = obj.location();
    if ( newLoc.equals(oldLoc) )
        return;

    Locatable foundObject = objectAt(oldLoc);
    if ( ! (foundObject == obj && isEmpty(newLoc)) )
        throw new IllegalArgumentException("Precondition violation " +
                        "moving " + obj + " from " + oldLoc);

    theGrid[newLoc.row()][newLoc.col()] = obj;
    theGrid[oldLoc.row()][oldLoc.col()] = null;
}
```

Analysis Question Set 5:

1. What would happen if the parameter passed to add were a null reference? Would an exception be thrown? Would the null reference be added to the environment? Would this invalidate the objectCount instance variable?

2. Why does the add method call the isEmpty method to check its precondition, while remove calls objectAt?

3. Consider implementing a new removeFrom method in BoundedEnv that would take a location as its parameter. The removeFrom method would remove whatever object was at the specified location from the environment. If the specified location were empty, removeFrom would just return without generating an error. How would the efficiency of this method compare to that of the existing remove method? In other words, how many steps would each method have to execute in order to remove an object?

```java
public void removeFrom(Location loc)
{
    // implementation goes here
}
```

Test Plan for the BoundedEnv Class

The test plan for the `BoundedEnv` class developed by the original programmer included black box and code-based test cases. The "black box" in this case, though, is the `BoundedEnv` class rather than the whole application. Thus, the black box test cases were developed by looking at the method descriptions, preconditions, and postconditions in the class documentation, and the code-based test cases were developed by looking at the method implementations.

An environment implementation is difficult to test in isolation, because the environment needs objects stored in it. Those objects could be anything that is `Locatable` — fish, band members, physics particles, or color blocks. The original programmer tested `BoundedEnv` in the context of the marine biology simulation program.

Black Box Test Cases

The class documentation for `BoundedEnv` yields black box test cases and informal expected results like those in the following incomplete list.

- The `numRows` and `numCols` methods for a `BoundedEnv` object should return `r` and `c`, where `r` and `c` are the first and second parameters passed to the constructor, respectively.
- The `isValid` method should return `false` when passed a location with a negative number for either the row or column value, when passed a location whose row value is greater than or equal to `numRows`, or when passed a location whose column value is greater than or equal to `numCols`. One clever example that tests all of these cases is a fish in location (0, 0) of a 1 x 1 bounded environment that asks for all its neighbors [(-1, 0), (0, 1), (1, 0) and (0, -1)]. The `isValid` method returns `false` for all four of these neighbors.
- The `isValid` method should return `true` if both the row and column values are greater than or equal to zero and less than `numRows` or `numCols`, respectively. One clever example that tests all of these cases is a fish in location (1, 1) of a 3 x 3 bounded environment that asks for all its neighbors [(0, 1), (1, 2), (2, 1) and (1, 0)]. The `isValid` method returns `true` for all four of these neighbors.
- The `numObjects` method should return the number of objects that have been added to the environment minus the number of objects that have been removed. If no objects have been added to the environment, `numObjects` should return zero. This can be tested with a number of different initial configuration files, including one that specifies environment dimensions but no fish.
- The `allObjects` method should return a list of the objects that have been added to the environment and not yet removed. If no objects have been added to the environment or if all the added objects have been removed, `allObjects` should return an empty list. An initial configuration file that specifies no fish, combined with sample runs of fish breeding and dying, will test this method thoroughly.

- The `objectAt` method returns objects at various locations. It returns `null` for an invalid location or a valid, empty location. The `allObjects` method tests `objectAt` for all valid locations (and also tests the `numRows` and `numCols` methods). Only the case of an invalid location remains untested.
- The `add` method adds the new object if the location is empty. Any run of the simulation program with at least one fish in it tests the proper use of the `add` method. The black box test case from Chapter 2 that specifies two fish in the same location tests the violation of the precondition.

These are just a few of the test cases for the `BoundedEnv` class. Running the marine biology simulation program with the test cases described in Chapter 2 addresses many of the test cases for `BoundedEnv`. For example, calling the `Simulation step` method on both an empty environment and on a non-empty environment will test the `allObjects` method and, based on our knowledge of the implementation of `allObjects`, will also test the `numRows` and `numCols` methods and some of the test cases for `objectAt`. Similarly, when a fish in location (0, 0) asks for all its neighbors, this tests that `isValid` works correctly for a negative row, a negative column, and, assuming that the environment is bigger than 1 x 1, two valid locations. When the fish asks if its two neighboring locations are empty, the simplest test cases for `isEmpty` and `objectAt` are also tested.

An Unbounded Environment: Problem Specification

Once Jamie explained the details of the `BoundedEnv` implementation to me and I had taken time to look it over more carefully, I was ready to think about the problem of implementing an unbounded environment.

The two-dimensional array representation used in the `BoundedEnv` class is appropriate for a simulation of a relatively small, roughly rectangular environment or even a large rectangular environment as long as it has many fish in it. If, however, the simulation is very large and has relatively few fish, then the two-dimensional array wastes space. The 2-D array representation is also inappropriate if the environment is so big that it is essentially unbounded (such as the ocean) or if its shape is very irregular.

As I thought about the problem of implementing an unbounded environment, I realized that I needed more information from the marine biologists. Did they want to only keep track of the fish within a bounded area, such as a fixed section of the Pacific Ocean, but allow fish to swim in and out of the area? If we weren't keeping track of the fish outside the area, the simulation could occasionally create new fish at the boundaries to represent fish swimming in and delete fish from the environment whenever they swam out. Or, did the biologists want to keep track of all the fish in the environment, regardless of where they swam, without imposing any boundaries on the environment? This might be more useful to the biologists if they were studying a certain population, such as a school of whales, as they moved throughout the ocean.

I talked to the biologists to clarify this point, and discovered that they wanted to track fish wherever they went in the environment, not just within certain boundaries. They also verified that they still wanted to be able to simulate fish movement in a bounded environment when that was more appropriate. I made the following list of requirements and verified it with the biologists before starting my design.

- The environment should continue to be modeled as a conceptual grid of cells, with fish moving from cell to cell in the grid.
- The program should support a bounded, rectangular environment for simulations where that is appropriate.
- The program should also support an unbounded environment for simulations where that is more appropriate.
- Regardless of the type of environment, the simulation should keep track of all the fish from the time they are born until they die.

Design and Implementation of the Unbounded Environment

I decided to review the environment design alternatives that Jamie had described to me. They included the two-dimensional representation that the original programmer used, a list of the objects and their locations, and a list of just locations. The original marine biology simulation programmer had decided that the third design was not appropriate because the fish in the environment are not all identical. The second design, though, seemed like it would be a good choice for an unbounded environment because it had no explicit boundaries.

	X	X
	X	

X, (0, 1)	X, (0, 2)	X, (1, 1)

(0, 1)	(0, 2)	(1, 1)

The `UnboundedEnv` Class

Instance variables, constructors, and the simplest accessor methods — `numRows`, `numCols`, `isValid`, and `numObjects`:
Now I was ready to implement the `UnboundedEnv` class. I started by copying `BoundedEnv.java`. Because the internal data representation for the new class would be different, it was obvious that the instance variables would change. Instead of a two-dimensional array containing fish and empty cells (where the fish's location in the array corresponds to its location in the conceptual grid of the environment), I needed a list of all the fish and their locations. Actually, because a fish keeps track of its own location, as do all `Locatable` objects, all I needed was a list of `Locatable` objects.

The next question was how to represent the list. I could have used a one-dimensional array, but that didn't seem a good choice since the number of fish in the environment changes every time a fish is born or dies. Arrays cannot change size after they are constructed. Instead I decided to use an `ArrayList`. This is a dynamic data structure that can grow and shrink as the simulation runs. (I had already seen `ArrayList` objects, because `Environment` and `Fish` use them for their lists of neighboring locations.) So, the key instance variable in the `UnboundedEnv` class is an `ArrayList` of `Locatable` objects called `objectList`. In fact, it is the only instance variable in this class. I realized I didn't need the `objectCount` instance variable because the expression `objectList.size()` provides the same information.

```
private ArrayList objectList; // list of Locatable objects in environment
```

The constructor for `UnboundedEnv` is a default constructor because it doesn't make sense to specify the number of rows and number of columns for an unbounded environment. The constructor simply calls `super()` to initialize any inherited attributes and then creates an empty `ArrayList`. It does not initialize `objectCount` because that instance variable doesn't exist.

The `numRows`, `numCols`, `isValid`, and `numObjects` methods are just as simple. The `numRows` and `numCols` methods return -1, an invalid dimension indicating that the environment is unbounded. The `isValid` method which tests locations always returns `true` (unless the location is `null`) because all locations are valid in the unbounded environment, including ones with negative indices. The `numObjects` method returns the size of `objectList`.

**Searching through the environment — `isEmpty, objectAt, allObjects,`
and `toString`:**
In the bounded environment, with its two-dimensional array implementation, the
`isEmpty` and `objectAt` methods do not need to search through the internal data
structure to find the appropriate location to check. They can go directly to the specified
row and column in the 2-D array. With the list implementation of the unbounded
environment, however, they must search through the list for the object with the
appropriate location.

In `BoundedEnv`, `isEmpty` returns `true` if the given location is in bounds and is empty. For `UnboundedEnv`, it is enough to just check whether the given location is empty, since all locations are "in bounds." That left the question of how to determine whether a location is empty. The `BoundedEnv` programmer chose to use the more general `objectAt` method to implement `isEmpty`, and I chose to do the same. My first draft of the `objectAt` method looped through the list, comparing the location of each `Locatable` object with the parameter, `loc`. When it found an object with a matching location (using the `equals` method in the `Location` class), it returned the object. Here's my first version of the code for `isEmpty` and `objectAt`.

```
public boolean isEmpty(Location loc)
{
    return (objectAt(loc) == null);
}

public Locatable objectAt(Location loc)      // first draft!
{
    // 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 it.
            return obj;
        }
    }

    // No such object found.
    return null;
}
```

The `allObjects` method seemed at first to be trivial; I could just return the `objectList` instance variable. Then I realized that `objectList` is an `ArrayList`, but `allObjects` needs to return the objects in a one-dimensional array. As a result, `allObjects` creates a new array of the appropriate size, and, in a loop, copies objects from the `ArrayList` to the array. The code appears below.

```
public Locatable[] allObjects()
{
    Locatable[] ObjectArray = new Locatable[objectList.size()];

    for ( int index = 0; index < objectList.size(); index++ )
    {
        ObjectArray[index] = (Locatable) objectList.get(index);
    }

    return ObjectArray;
}
```

The order of the objects in the list returned by the `allObjects` method is the same as their order in the internal `ArrayList`, but might not be the same as the order returned by the `allObjects` method in the `BoundedEnv` class. This does not violate the `Environment` interface, since it only states that the `allObjects` method "Returns all the objects in this environment." It does not specify the order in which they are returned.

The `allObjects` method had at first seemed trivial, but the `toString` method really was. I did not have to make any modifications to it from the `BoundedEnv` version.

Analysis Question Set 7:

1. Why did Pat's first draft of the `objectAt` method use the `equals` method in `Location` rather than the `==` operator?

2. The lists returned by the `BoundedEnv` and `UnboundedEnv` versions of the `allObjects` method might have their objects in different orders. How could this difference lead to different behavior in the marine biology simulation, even if the initial configuration of fish and the random number seed were the same?

Modifying an environment — `add`, `remove`, and `recordMove`:
The `add` method in `UnboundedEnv` is very similar to that in `BoundedEnv`. The only difference is that it uses the `ArrayList add` method to add the object to the list rather than putting it in the appropriate spot in the two-dimensional grid.

Next I turned to the `remove` method. The first thing it needs to do is to determine where the object to be removed is located in the list of `Locatable` objects. This is similar to what happens in `objectAt`, except that `objectAt` is passed a location and `remove` is passed the actual object, from which it can get the location. In fact, my first draft of `remove` called `objectAt` to check the precondition that the object is in the environment, just as the `remove` method of the `BoundedEnv` class does. Then it called the same `ArrayList remove` method that was used in the `nextLocation` method of `Fish`, passing it the object to remove.

```
public void remove(Locatable obj)              // first draft!
{
    // 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.
    objectList.remove(obj);
}
```

After thinking about the performance of my draft `remove` method, I decided to modify it. Rather than have it step through the list twice, once to check that the object was there and once to remove it, I decided to step through the list once. I would find the index of the object in the list while checking the precondition, and then use that index to remove the object directly, without doing a second search. Then I decided that rather than writing two versions of the code to step through the list, one in `objectAt` looking for an object and one in `remove` looking for an index, I would write it once in a helper method and call the helper method from both `objectAt` and `remove`. Since it is easy and efficient to get an object from an `ArrayList` given its index, the new method, `indexOf`, returns the index of the `Locatable` object for a given location, or -1 if no such object exists. Its code is based on my first draft of `objectAt`. The new `objectAt` method merely returns the object at that index, while `remove` calls the `ArrayList` `remove` method that takes an index as a parameter. The code for `indexOf` and the modified `objectAt` and `remove` methods, with some comments removed, appears below.

```
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;
}

public Locatable objectAt(Location loc)        // final draft!
{
    int index = indexOf(loc);
    if ( index == -1 )
        return null;

    return (Locatable) objectList.get(index);
}

public void remove(Locatable obj)              // final draft!
{
    int index = indexOf(obj.location());
    if ( index == -1 )
        throw new IllegalArgumentException("Cannot remove " +
            obj + "; not there");

    objectList.remove(index);
}
```

Finally I moved on to the `recordMove` method. If this method didn't check its precondition and postcondition, it wouldn't have to do anything at all! The unbounded environment does not keep track of an object's location in any way, it just makes use of the fact that all the objects in it are `Locatable` and therefore keep track of their own locations. If an object modifies its own location, then it has moved, and there's nothing in the environment that needs to be updated as a result.

So, the only thing I needed to do in `recordMove` was to check the precondition and postcondition. Just as a precondition is a condition that must be true before the method executes, a postcondition is one that must be true after the method executes. The precondition and postcondition for a method are sometimes called its *contract*, since they state that if the precondition is met when the method begins executing, then the method will make sure that the postcondition is met when it is done. Checking the precondition and postcondition for `recordMove` isn't as trivial as it sounds. The precondition states that "`obj.location()` is a valid location and there is no other object there." The first part of the condition does not actually need to be checked, since all locations are valid in an unbounded environment. To check the second half, though, requires verifying that the object itself is in the list and that there is no other object in the list at that location. That means checking all the objects in the list. The postcondition for `recordMove` states that "`obj` is at the appropriate location (`obj.location()`), and either `oldLoc` is equal to `obj.location()` (there was no movement) or `oldLoc` is empty." Again, the first part of this condition does not require any checking; if the object is in the list at all, it must be at the appropriate location. The second part of the condition requires looking through the list to see if there is an object whose location is the same as `oldLoc`. If there is, it should be the object that "moved" (or, rather, didn't move).

My first draft of `recordMove`, shown below, was simple but, unfortunately, incomplete and inefficient.

```
public void recordMove(Locatable obj, Location oldLoc)
                                        // incomplete draft!
{
    // Simplest case: There was no movement.
    Location newLoc = obj.location();
    if ( newLoc.equals(oldLoc) )
         return;

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

This version of recordMove checks that the object is in the list and that oldLoc is either equal to the object's location (no movement) or empty. It does not, however, verify that there aren't two objects at the new location, which could happen if an object moved to a location without first checking that it was empty. After some thought, I implemented the version of recordMove shown below. This version makes a single pass through the list, looking for all objects in either the object's new location or its old location. At the end of the pass, it checks that there was exactly one object at the new location. It also checks that the old location is either the same as the new location (in which case we know that there is exactly one object there) or that it is empty.

```
public void recordMove(Locatable obj, Location oldLoc)
                                           // final draft!
{
    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);
    }
}
```

Displaying an Unbounded Environment

How does one display an unbounded environment, or even an extremely large bounded environment? I decided to ask Jamie about this. It turns out that the graphical display class I was using to display the fish in the environment always uses the same size display window and tries to fit the entire environment in that window. It adjusts the size of the cells in the environment depending on how many cells there are in the environment. For example, a 10 x 10 environment would have much smaller cells than a 5 x 5 environment. If the environment has so many cells that they would be too tiny to see, however, the display class abandons its attempt to display all the cells and only shows a portion of the environment. In other words, cell size gets smaller and smaller as the environment gets larger and larger, until the cell size hits a predefined minimum. According to Jamie, if `numRows` and `numCols` return unreasonable values, either negative numbers or unacceptably large positive numbers, the display defaults to the minimum cell size and shows whatever portion of the environment fits in the window with location (0, 0) in the upper left-hand corner.

Choosing an Appropriate Environment Representation

One question I still had was how the program would know whether a bounded or unbounded environment was more appropriate? I talked with the marine biologists and they said that they would specify in the initial configuration files whether an environment should be bounded or unbounded. They would specify a bounded environment by putting the word "bounded" in the first line with the environment's dimensions, as they were already doing. They would specify an unbounded environment by putting the word "unbounded" in the first line, without any dimensions. Because it is the responsibility of the program driver (main method, applet, or graphical user interface) to read in the initial configuration file and construct the environment, the driver code would need to change. I was using a graphical user interface that the original programmer had written and was not familiar with that code; Jamie offered to update it for me.

Testing the UnboundedEnv Class

The black box test cases for the `UnboundedEnv` class are the same as for the `BoundedEnv` class, although the expected results are often different. For example, a fish in location (0, 0) in a bounded environment should have two valid neighboring locations, while the same fish in an unbounded environment will have four. The list below describes the black box behavior of the `UnboundedEnv` class relative to the behavior of the `BoundedEnv` class.

- The `numRows` and `numCols` methods for an `UnboundedEnv` object always return -1.
- The `isValid` method should always return `true`.
- The `numObjects` and `allObjects` methods should return the same values as they do for a bounded environment.
- The `isEmpty` and `objectAt` methods should work just as they do for a bounded environment, except that all non-null locations are valid.
- The `toString` method should work just as it does for a bounded environment.
- The `add`, `remove`, and `recordMove` methods should work just as they do for a bounded environment.

The first thing I did to test the unbounded environment was to run it with the same tests that I had used to test my previous modifications to the simulation program. For all cases where the fish had been somewhere in the middle of the bounded environment I expected to see similar types of results, although not exactly the same. The differences in behavior that I was most interested in, though, were with fish along the edges or boundaries of the bounded environment.

To test the changed behavior for an unbounded environment I created a new initial configuration file (`boundaryFish.dat`) with fish lined up along row 0 and down column 0 (what had been the top row and the left-most column of the bounded environment). I also turned on debugging for just the `Fish move` method. I expected about half of the fish to move out of the display area in the first timestep, and for some of them to move back into the display area in later timesteps. I found that fish did move out of the display area as I had expected. To my surprise, at least until I thought more about it, not a single fish moved back into the display area during the second timestep, although some did in the third and fourth timesteps.

Analysis Question Set 9:

1. When Pat was testing the `UnboundedEnv` class, why didn't any fish that moved out of the display area in the first timestep move back into it in the second timestep?

2. Why might the behavior of a fish that had been in the middle of a bounded environment be different in an unbounded environment?

3. What kind of test program would you write if you wanted to test the `UnboundedEnv` class on its own before using it in the marine biology simulation program?

4. How would you implement a very large bounded environment class? Would you represent the environment internally using a two-dimensional array, an `ArrayList`, or some other data structure? If you were to use an `ArrayList`, what methods could you use from `UnboundedEnv`? What methods could you use from `BoundedEnv`?

5. What is the performance of the `objectAt` method if the environment is a `BoundedEnv` object? What if it is an `UnboundedEnv` object?

6. What is the performance of the `allObjects` method if the environment is a `BoundedEnv` object? What if it is an `UnboundedEnv` object?

7. What is the performance of the `Fish isInEnv` method if the environment is a `BoundedEnv` object? What if it is an `UnboundedEnv` object?

8. Consider representing `objectList` in other ways. What would be the effect on the performance of methods like `objectAt`, `allObjects`, `add`, and `remove` if the list were kept in sorted order? If the list were represented as a binary search tree? If it were represented as a hash map?

Exercise Set 1:

1. Run the marine biology simulation with data files from the `UnboundedEnvDataFiles` folder inside the `DataFiles` folder. Run the same tests you ran in Chapters 1– 4. What differences do you observe?

2. Run the simulation with the `boundaryFish.dat` file from the `UnboundedEnvDataFiles` folder. What behavior do you observe?

3. Implement a class to represent a very large bounded environment that contains relatively few fish in one of two ways.

    *   Define a `VLBoundedEnv` class that extends `UnboundedEnv`. You will need to add a constructor and modify all methods that depend on the bounds. [Alternative: Because a very large bounded environment is not a specialized type of unbounded environment, a cleaner design would be to have an abstract `ListEnv` class that implements only the methods that support the `ArrayList` representation, regardless of whether the environment is bounded or unbounded. Then have both `UnboundedEnv` and `VLBoundedEnv` extend `ListEnv`.]
    *   Define an `SMBoundedEnv` class that uses a sparse matrix. This version should extend `SquareEnvironment` and should use an array of linked lists to represent the grid of `Locatable` objects. Each element in the array should be a linked list of the objects that occur in that row of the grid, in increasing column order. (AB only)

    [Note: If you are using the graphical user interface distributed with the case study, edit the MBSGUI class and add VLBoundedEnv or SMBoundedEnv to the list of classes that can represent bounded environments. Now when you read in a file specifying a bounded environment, you will be given a choice as to which representation you would like to use.]

4. Define a `SLUnboundedEnv` class that uses a sorted list. Make sure that your `objectAt` method takes advantage of the sorted nature of the list. (AB only) *[Note: If you are using the graphical user interface distributed with the case study, edit the MBSGUI class and add SLUnboundedEnv to the list of classes that can represent unbounded environments.]*

5. Define a `BSTUnboundedEnv` class that uses a binary search tree. (AB only) *[Note: If you are using the graphical user interface distributed with the case study, edit the MBSGUI class and add BSTUnboundedEnv to the list of classes that can represent unbounded environments.]*

6. Define a `HMUnboundedEnv` class that uses a hash map. (AB only) *[Note: If you are using the graphical user interface distributed with the case study, edit the MBSGUI class and add HMUnboundedEnv to the list of classes that can represent unbounded environments.]*

Quick Reference for Core Classes and Interfaces

This quick reference lists the constructors and methods associated with the core classes described in this chapter. Public methods are in regular type. *Private and protected methods are in italics.* (Complete class documentation for the marine biology simulation classes can be found in the Documentation folder).

```
Environment Interface

public int numRows()
public int numCols()

public boolean isValid(Location dir)
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) (Black Box)

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

This quick reference lists the public constants, constructors, and methods associated with the utility classes described in this chapter. The marine biology simulation program also uses subsets of the following standard Java classes: `java.util.ArrayList`, `java.util.Random`, and `java.awt.Color`. (Complete class documentation for the marine biology simulation classes can be found in the `Documentation` folder).

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(Object other)
Direction toRight()
Direction toRight(int degrees)
Direction toLeft()
Direction toLeft(int degrees)
Direction reverse()
String toString()
static Direction randomDirection()
```

The following are not tested:
FULL_CIRCLE
```
int hashCode()
Direction roundedDir(int numDirections,
          Direction startingDir)
```

`EnvDisplay` Interface

```
void showEnv()
```

`Locatable` Interface

```
Location location()
```

`Location` Class

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

The following is not tested:
```
int hashCode()
```

`RandNumGenerator` Class

```
static Random getInstance()
```

Java Library Utility Classes

```
java.util.ArrayList Class (Partial)

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

Marine Biology Simulation
Case Study

Appendix A

Testable Classes and Concepts

The AP Computer Science Course Description and AP Computer Science Java Subset list the topics, concepts, and Java language constructs that are part of the AP Computer Science A and AB curricula and may be tested on the AP Exams. This appendix lists additional classes and terms from the marine biology simulation case study that may be tested on the exams. It also lists classes and terms that appear in the case study code, class documentation, or narrative but will not be tested. Items labeled (AB only) may be tested on the AB exam but will not be tested on the A exam.

Classes and Terminology

Chapter	May appear in test questions	Will NOT be tested
1		`SimpleMBSDemo1` `SimpleMBSDemo2` `MBSGUI` **terms:** *driver*
2	`Simulation` (implementation) `Environment` (documentation) `Fish` (implementation) `Debug` (documentation) `Direction` (documentation) `EnvDisplay` (documentation) `Locatable` (documentation) `Location` (documentation) `RandNumGenerator` (documentation) `ArrayList: remove(Object)` method (and other methods in the AP subset) `Color:` constructor and constants in Quick Reference **terms:** *boundary tests, client code, dynamic binding, field, instance variable, override, pseudo-code, redefine, timestep*	`Fish: nextAvailableID` class variable `Direction: FULL_CIRCLE` constant; `hashCode` and `roundedDir` methods `Location: hashCode` method visibility rules of `protected` keyword implementation of black-box classes **terms:** *black box, black-box and code-based tests, class variable, consistent and inconsistent state, pseudo-random, seed*

Chapter	May appear in test questions	Will NOT be tested
3	`Fish` (modified implementation) **terms:** *pseudo-code*	

4	`DarterFish` (implementation)
	`SlowFish` (implementation)
	redefinition of `public` and
	   `protected` methods
	terms: *dynamic binding,
	  override, redefine*
		terms: *regression testing,
		  probabilistic vs.
		  deterministic methods*

5	`Environment` (complete
	  interface)
	`BoundedEnv` (implementation)
	  (AB only)
	`UnboundedEnv` (implementation)
	  (AB only)
		`SquareEnvironment`
		terms: *row-major order,
		  contract*

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 `Environment`, which is black box, can be found in Appendix C.

Simulation.java

```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)

```java
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());
    }
```

```java
    /** 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;
    }
```

```java
    /** 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());
    }
```

```java
    /** 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

```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);
    }
```

```java
   /** 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());
       }
   }
```

```java
    /** 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

```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
    }
```

```java
    /** 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 marine biology simulation utility classes covered in Chapters $1 - 4$ (`Debug`, `Direction`, `EnvDisplay`, `Locatable`, `Location`, and `RandNumGenerator`).

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 dir)`
> 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 neighbor of a location in the specified direction (whether valid or invalid).

`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.

`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(java.lang.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(java.lang.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

Constants: EAST, NORTH, NORTHEAST, NORTHWEST, SOUTH, SOUTHEAST, SOUTHWEST, WEST, FULL_CIRCLE (FULL_CIRCLE will not be tested on the AP Exam)

public **Direction**()
     Constructs a default `Direction` object facing North.

public **Direction**(int degrees)
     Constructs a `Direction` object — initial compass direction in degrees.

public **Direction**(java.lang.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**(java.lang.Object other)
     Indicates whether some other `Direction` object is "equal to" this one.

public int **hashCode**()
     Generates a hash code for this direction (will not be tested on the AP Exam).

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 java.lang.String **toString**()
     Returns a string indicating the direction.

public Direction **roundedDir**(int numDirections, Direction startingDir)
     Rounds this direction to the nearest "cardinal" direction (will not be tested on the AP Exam).

public static Direction **randomDirection**()
     Returns a random direction.

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

```
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(java.lang.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 (will not be tested on the AP Exam).

```
public int compareTo(java.lang.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 java.lang.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.

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++;
}
```

```java
    /** 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

```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;
    }
```

```java
/** 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);
    }
```

```java
/** 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 (black box)

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**()
> Generates a random direction.

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 E

Quick Reference for A Test

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)
```

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(Object other)
Direction toRight()
Direction toRight(int degrees)
Direction toLeft()
Direction toLeft(int degrees)
Direction reverse()
String toString()
static Direction randomDirection()
```

The following are not tested:
```
FULL_CIRCLE
int hashCode()
Direction roundedDir(int numDirections,
          Direction startingDir)
```

EnvDisplay Interface

```
void showEnv()
```

Locatable Interface

```
Location location()
```

Location Class

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

The following is not tested:
```
int hashCode()
```

RandNumGenerator Class

```
static Random getInstance()
```

Java Library Utility Classes

java.util.ArrayList Class (Partial)

```
boolean add(Object o)
void add(int index, Object o)
Object get(int index)
Object remove(int index)
boolean remove(Object o)
Object set(int index, Object o)
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 F

Quick Reference for AB Test

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) (Black Box)

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(Object other)
Direction toRight()
Direction toRight(int degrees)
Direction toLeft()
Direction toLeft(int degrees)
Direction reverse()
String toString()
Static Direction randomDirection()
```

The following are not tested:
```
FULL_CIRCLE
int hashCode()
Direction roundedDir(int numDirections,
          Direction startingDir)
```

EnvDisplay Interface

```
void showEnv()
```

Locatable Interface

```
Location location()
```

Location Class

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

The following is not tested:
```
int hashCode()
```

RandNumGenerator Class

```
static Random getInstance()
```

Java Library Utility Classes

java.util.`ArrayList` Class (Partial)

```
boolean add(Object o)
void add(int index, Object o)
Object get(int index)
Object remove(int index)
boolean remove(Object o)
Object set(int index, Object o)
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.

Simulation.java

Fish.java

DarterFish.java

SlowFish.java

BoundedEnv.java

UnboundedEnv.java

2001-02 AP Computer Science Development Committee and Chief Reader

Mark Weiss, Florida International University, Miami, *Chair*
Robert (Scot) Drysdale, Dartmouth College, Hanover, New Hampshire
Judith Hromcik, Arlington High School, Texas
Joe Kmoch, Washington High School, Milwaukee, Wisconsin
Richard Kick, Hinsdale Central High School, Illinois
Andrea Lawrence, Spelman College, Atlanta, Georgia
Julie Zelenski, Stanford University, California
Chief Reader: **Christopher Nevison,** Colgate University, Hamilton, New York
ETS Consultants: **Frances Hunt, Dennis Ommert**

8/2002