Marine Biology Simulation Case Study

Chapter 2

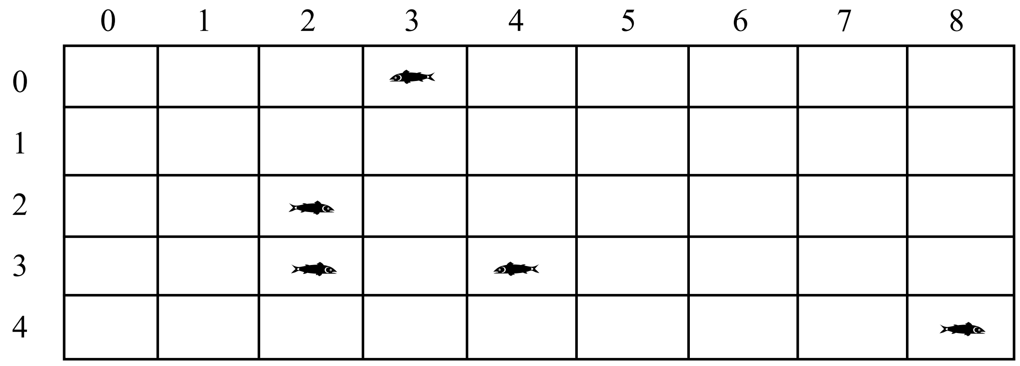
Guided Tour of the Marine Biology Simulation Implementation

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.



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.‡

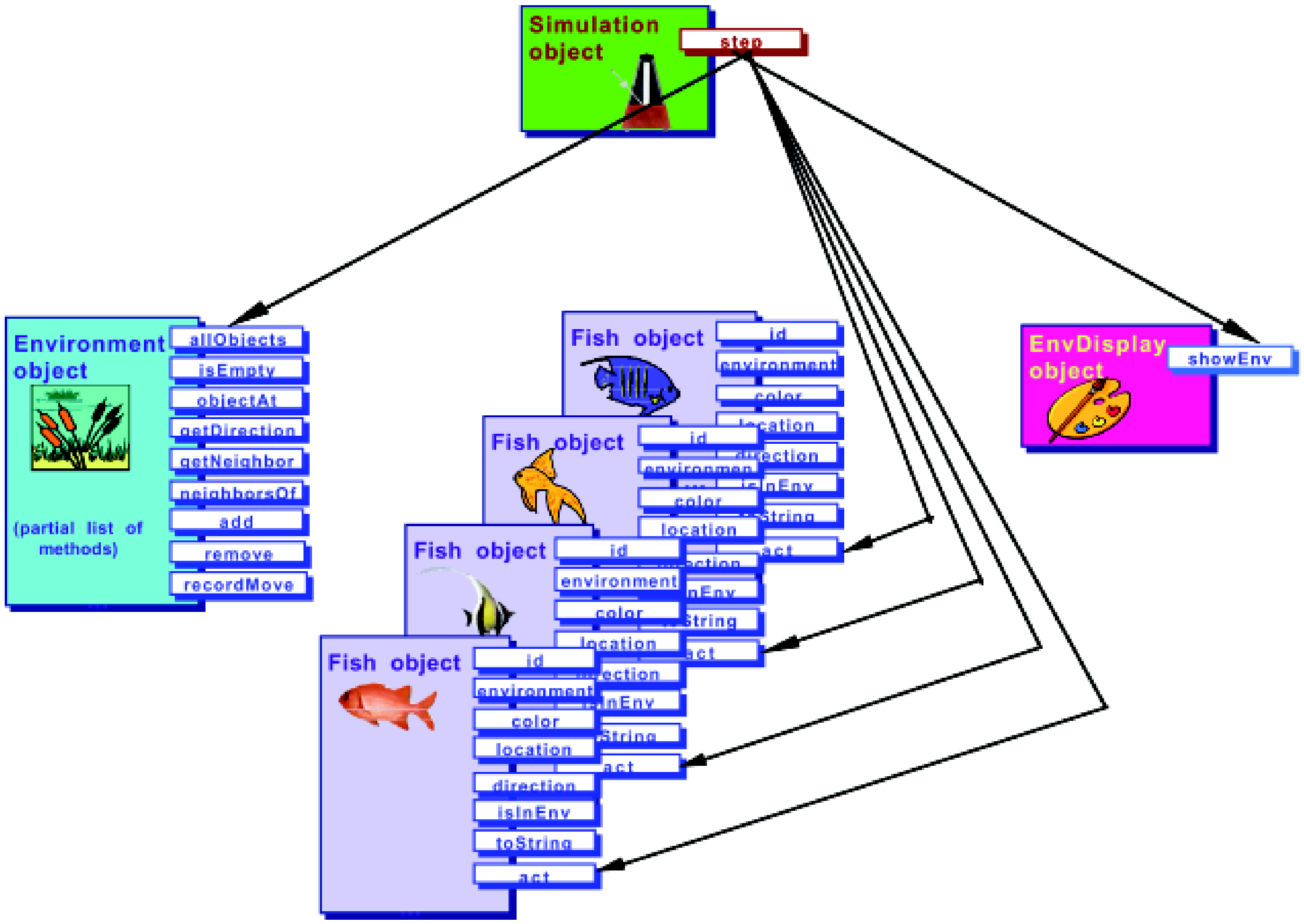
Because the `Simulation` constructor needs the environment and display, the driver (applet, main method, or user interface) that constructs the `Simulation` object must construct the environment and display objects first. The `Simulation` constructor then stores the two parameters so the `step` method 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.



Analysis Question Set 1:

1. At the end of Chapter 1, Pat wondered how the `step` method asks the fish to act when `SimpleMBSDemo2` never passes the fish to the simulation. How does `step` know what fish to ask to act? How did it get enough information from the driver to be able to do this?
2. How does the code in the `Simulation` class explain the differences between `SimpleMBSDemo1` and `SimpleMBSDemo2`?

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.

```
// 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)
```

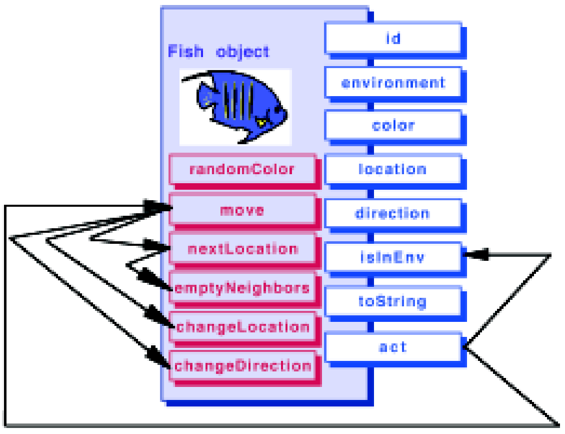
The most important method in the `Fish` class from the client code's perspective is the `act` method. This is the method in which a fish does whatever action is appropriate for a single timestep in the simulation. The code for the `act` method, though, actually does very little. It first checks that 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.



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

Analysis Question Set 5:

Consider the following variable definitions.

```
Location loc1 = new Location(7, 3);
Location loc2 = new Location(2, 6);
Location loc3 = new Location(7, 3);
```

1. What does the expression `loc1 == loc1` evaluate to? What about `loc1.equals(loc1)`?
2. What does the expression `loc1 == loc2` evaluate to? What about `loc1.equals(loc2)`?
3. What does the expression `loc1 == loc3` evaluate to? What about `loc1.equals(loc3)`?

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
```

Analysis Question Set 6:

1. Why does the `Fish` class need an `emptyNeighbors` method? Why doesn't `nextLocation` just call the `neighborsOf` method from the `Environment` class?

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.

Exercise Set 3:

1. Modify your driver program from Exercise Set 2 to find out how many neighboring locations there are around locations (0, 0), (0, 1), and (1, 1).
2. What are those neighbors? Print them out.
3. Based on the dimensions you gave your bounded environment, what other locations have the same number of neighbors?

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.‡

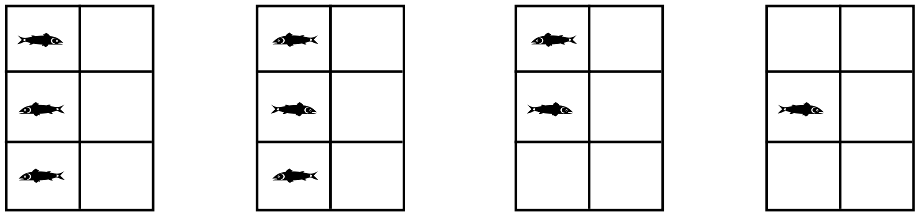
```
protected Location nextLocation()
{
    // Get list of neighboring empty locations.
    ArrayList emptyNbrs = emptyNeighbors();

    // 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

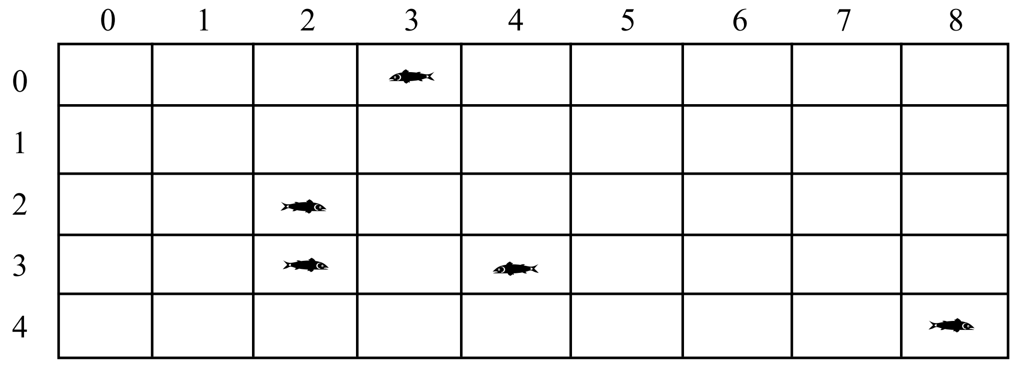
The problem specification defined the following requirements for the simulation.

- 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)
```

<u>Fish Class (implements Locatable)</u>

```
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()
```