The College Board AP® ADVANCED PLACEMENT PROGRAM®

AP® Marine Biology Simulation Case Study

COMPUTER SCIENCE

Teacher's Manual

CS

Advanced Placement Program®
Computer Science

Marine Biology Simulation Case Study Teacher's Manual

The AP® Program wishes to acknowledge and to thank Kathleen Larson of Kingston High School in Kingston, New York.

Permission to Reprint Statement

Contents

Introduction to Case Studies 6

Introduction to the Teacher's Manual 7

Chapter 1 9

Chapter 2 14

Chapter 3 36

Chapter 4 42

Chapter 5 68

Appendix A A1

Appendix B. B1

Appendix C. C1

Appendix D D1

Introduction to Case Studies

Case studies are a teaching tool used in many disciplines. They give students an opportunity to learn from others with more experience and skill. Just as lawyers, physicians, and therapists learn from the prior experience of others in their fields, so can programmers learn from more experienced programmers. Case studies have been a part of the AP Computer Science (APCS) curriculum since the 1994-95 academic year. Some good reasons to use a case study in AP Computer Science include:

- Working with a program of significant length
- Thinking through issues of good program design
- Learning from an expert programmer

A lot can be learned from reading programs written by another person. Difficult topics such as program design and construction of algorithms are often best introduced through studying existing programs, including their design, implementation, and validation. For example, think about when you first studied sorting algorithms. You could start from scratch and devise your own sorting algorithms, or you could learn from the work of others who have devised a selection sort or quicksort. This is also true when learning about object-oriented design, a topic recently introduced to the AP Computer Science curriculum. Through the *AP Marine Biology Simulation Case Study*, the strategies, vocabulary, and techniques of object-oriented design will be emphasized. The case study document, code, and teacher's manual are provided to support learning through the application of these techniques.

Introduction to the Teacher's Manual

The *AP Marine Biology Simulation Case Study* (MBS) provides a large body of code to be studied by all students preparing for the AP Computer Science (APCS) Examinations. This ensures that all students taking the exams will have access to a common, testable body of code. One free-response question and five to ten multiple-choice questions on each of the APCS Examinations will test fundamental computer science concepts in the context of the case study.

You may integrate the case study materials into your course as you see fit. The case study and the accompanying teacher's manual were designed in such a way that you can use these materials throughout the course. You may, in fact, wish to teach many computer science concepts from the APCS curriculum through the case study itself. On the other hand, you have the option of including the case study as a stand-alone unit at some point during the school year. Exactly when to introduce the case study and how much time to devote to it are matters for each teacher's discretion. Keep in mind, however, that your students do need to be prepared to answer the case study questions they will encounter on the examinations.

Be sure to read carefully the "Note about the AP Computer Science Exams" found on page 8 of the case study. Students taking the A exam will be tested on Chapters 1 through 4 only. Students taking the AB exam will need to be familiar with Chapters 1 through 4 and Chapter 5. However, you should feel free to cover Chapter 5 with your A students if you want to do so.

You can begin using the case study very early in your course, even on the first day of school. Students with minimal programming experience will be able to read and understand Chapter 1. If you think your students already know the concepts in Chapter 1, you may decide to begin with Chapter 2. Chapters 3 and 4 are independent of each other and can be taught in either order.

Additional introductory and supplementary materials are available if you need more help getting started. Resources can be found through a variety of sources, such as AP Central™ (apcentral.collegeboard.com) and the AP Computer Science electronic discussion group (EDG).

The code and class documentation provided in Case Study Appendices B, C, and D and the Quick Reference information in Appendices E and F will be provided to students at the time they take the examination. Appendices D and F refer to the AB exam only.

Be sure to look at everything in the `JavaMBS` folder. A wealth of helpful information has been provided for you.

The `index.html` file serves as an annotated table of contents for the files and folders that are part of the distribution, including the `ExecutionInformation` folder. This file is particularly helpful for navigating around the case study and is a very good place to begin.

Note that two versions of the `Fish` class are provided in the `Code` folder. The simpler version is found in the outer `Code` folder along with the other classes and files used by the *AP Marine Biology Simulation Case Study*. In this simpler version, fish move but they don't breed and die. The more complex `Fish` class, found in the `DynamicPopulation` folder within the `Code` folder, includes breeding and dying. You may toggle these files in and out of your project, depending upon whether or not you want fish to breed and die, or you may want to maintain a separate project for each of the two different `Fish` classes. For example, in Chapters 1 and 2 you will probably want to use the simpler `Fish` class so your students can study fish movement without having new fish pop up and current fish disappear. Chapter 3 is about a dynamic fish population, so you will need to use breeding and dying `Fish`. When you teach Chapter 4, with different species of fish, you may want to use the original `Fish` class at first to study fish movement patterns again without the added complexity of breeding and dying. Once the new movement patterns are understood, you can then switch back to the dynamic population.

The `Fish` class provided in Appendix B is the dynamic version used in Chapter 3. This is very important because it is the dynamic version that will be tested on the AP Exam. However, prior to studying Chapter 3, you may want to provide your students with printed copies of the simpler version of the `Fish` class found in the `Code` folder. That way, they can work with the version of `Fish` that contains everything they need for Chapter 2 without the additional `Fish` methods introduced in later chapters.

Before starting, you will probably want to decide how you are going to distribute the code to your students and what compiler or IDE you will be using in the classroom. For example, you might instruct students to download and unzip the entire distribution file from the Web, or you might have them copy the `JavaMBS` folder from a server to their own work area. Alternatively you could keep the entire MBS installation intact, in write-protected folders, but make a copy of the `Code` folder as a working directory and put the appropriate project files, batch files, or makefiles in the working directory with the code. The `ExecutionInformation` folder in the distribution file contains important information on compiling and running the case study with several different environments.

Chapter 1

Experimenting with the Marine Biology Simulation Program

There are no educational prerequisites for beginning this chapter. Your students could work their way through most of Chapter 1 without advance knowledge about loops, conditions, arrays, or classes. All they need to do is follow your instructions for running the case study and make observations about what they see happening. The final section, however, "Sneaking a Peek at Some Code", assumes the student is familiar with constructing objects, invoking methods and `for` loops. You may decide to treat this as an opportunity to teach these topics through the case study. For example, if it's early in the year and you haven't yet taught loops, "Sneaking a Peek at Some Code" could provide the motivation for introducing looping to the class.

Teaching tip: Allow students time to enjoy running the program and to discuss their findings with classmates.

Teaching tip: If you are using the default graphical user interface (GUI) distributed with the case study (i.e., if you haven't replaced it with an interface from another source), you may wish to go through the help file that documents the graphical user interface. You can find the help file (`MBSHelp.html`) in the `JavaMBS/Documentation` folder or from the Help menu in the simulation program. This document may help you answer student questions that come up. You do not need to read or understand all the GUI before starting to teach the case study, though. For example, only the topics in the first two bullets are used in Chapter 1. The next three topics are useful for Chapter 2 (although creating a new environment using the GUI is optional). The last two topics are useful mostly for Chapters 4 and 5. Students do not need to read the help file before Chapter 1. The aspects of the user interface that they need for this chapter should be relatively self-explanatory. They will, however, need instructions for compiling and running the case study in your particular lab setting.

Exercise Set 1 (page 10)

1. Refer to the `ExecutionInformation` folder included with the case study. It contains basic information on how to compile and run the simulation using several different IDEs. This exercise shows students how to get the simulation up and running. They do not need to understand how the simulation or the graphical display work. Encourage your students to experiment with the various GUI menu options.

2. Yes, the *behavior* of each fish is the same both times, although the positions the fish move into will change. This may be hard for students to see because there are many fish moving about. Your students may observe that the fish begin in the same positions each time the simulation is run. They may also remark that they don't see fish going backward. Some students may notice that the fish don't move diagonally either. The main point is that their behavior appears to be consistent although they do not necessarily move to the same positions as they did in a previous run.

Teaching tip: The perspective of the display is that of an overhead camera looking down on the fish in the water. When a fish moves up on the screen, it is moving north. When it moves down, it is moving south, and so on. Students sometimes incorrectly think they are seeing a slice of the body of water.

Analysis Question Set 1 (page 10)

1. The program appears to model the body of water as a two-dimensional grid, something like a graph in coordinate geometry. The grid lines represent boundaries between locations. Students do not need to know any specific syntax from mathematics or Java to answer this question. They can describe the grid as rows and columns or in some other format. If you are using the graphical user interface that comes with the MBS distribution, your students may notice the "tool tips" that pop up in the environment when the mouse pauses for a couple of seconds over a cell in the grid. The tool tips show the cell locations in a row, column format where rows and columns start at 0 and location (0, 0) is in the upper-left corner.

Teaching tip: You might want to investigate whether the body of water is truly bounded. If a fish reaches an edge, can it go off the screen and return or does the boundary act like the edge of a body of water and limit the fish so it can't go beyond the shoreline?

2. Fish can only move within the boundaries of the model and there can never be more than one fish in a location at a given time.

3. The fish may face any of the four major compass directions. The fish's direction does not appear to matter in terms of its initial location, but its direction seems to limit the locations to which the fish may move. This is somewhat difficult to determine when the students are looking at many fish and it provides the motivation to continue the investigation process in the next question.

4. A single fish does not necessarily move in every timestep. Sometimes the fish is prevented from moving because there are no unoccupied locations adjacent to its current location. When it moves, a fish moves exactly one space on the grid. Fish do not always move in the same direction, but the fish don't appear to move backward. Other than not moving backward, there doesn't seem to be a preference for moving straight ahead or turning left or right.

Teaching tip: Some students may argue that the fish do indeed go in the opposite direction but if they look carefully they will see that turning around requires two ninety degree turns, moving to an adjacent location on each turn.

Exercise Set 2 (page 11)

1. You will find a master for the table shown in this problem in Appendix C. You might want to make copies and distribute them to your students before they run the program. They can fill in the table and compare their findings with each other or they can turn them in for a grade. They should fill in the location columns with coordinates in (row, column) order. Direction columns should indicate north, south, east, or west (or an abbreviation). Let students reach their own conclusions and then compare them to the list that follows Exercise Set 2. Answers will vary.

Teaching tip: For some students, identifying rows versus columns and remembering which is which is a stumbling block. The problem is that they are familiar with x and y coordinates, or horizontal locations followed by vertical. Row, column order is vertical first, followed by horizontal. Reinforce verbalizing and writing the correct order from the very beginning of your teaching of the case study.

2. Students may think they have enough data after filling in the chart. Remind them that five timesteps provide very little evidence and are certainly not conclusive.

Analysis Question Set 2 (page 12)

1. Answers will vary but should generally support the conclusions listed on page 11. This should generate classroom discussion.

2. You will find a master for this chart in Appendix C. Repeat filling in the chart and discuss the results. There is stronger evidence now, but it still may not be conclusive.

Analysis Question Set 3 (page 13)

1. The first number represents the row and the second represents the column.

Teaching tip: There are several issues that students may have to grapple with here.

- Row and column values start at 0, not 1.
- The origin is (0, 0) and is located in the upper left hand corner of the grid.
- As you move South in the grid, row values increase.
- As you move East in the grid, column values increase.

Exercise Set 3 (page 13)

1. Yes, everything is as expected.

2. An easy way to do this is to work from an existing file. Open a file, make changes and save the data file under a new name with the `.dat` extension. You could also use an existing file as a model and have students create a new file from scratch.

Note to teachers: Up to this point, there isn't anything that requires any programming knowledge — just the ability to compile and run a program and look at data files in some editor. These exercises can be done as early in the year as you choose.

Sneaking a Peek at Some Code (pages 14–17)

Teaching tip: If students are not already familiar with constructing objects or invoking methods, stop here (or prior to beginning Chapter 2). You could cover these topics by exploring support materials available through AP Central or by using your textbook or other reference materials.

When a fish is constructed, it is passed environment and location parameters. In `SimpleMBSDemo1`, the environment is passed using a variable, `env`. A location, however, is passed by constructing an anonymous `Location` object within the parameter list. Giving a location a variable name isn't necessary because a reference to the location becomes part of the fish's private data.

The image on page 17 is called an "object diagram". It shows interactions among several different objects, which may be instances of the same class or of different classes. This object diagram illustrates the behavior of the driver in `SimpleMBSDemo2`. Public methods are represented with rectangles that extend across the border of the class. When private data and methods are shown, they are entirely within the rectangle representing the class object. Four fish are shown in the diagram but the simulation could be run on a smaller or a larger number of fish.

Object diagrams show which methods are defined for each object and how several objects interact, but they do not show the details of any given method. Don't let your students be concerned about the details at this time. They will learn more about how the methods are implemented in Chapter 2.

Teaching Tip: Make overhead transparencies of this and all other object diagrams (found in Appendix C of this manual) and provide a set of paper copies for your students. They can follow along as you trace the simulation flow. These diagrams can also be very helpful when trying to modify the case study or add methods. They help students keep track of the instance variables, methods, and dependencies.

If students want to experiment with adding more fish, they can do so by directly adding `Fish f4`, `Fish f5`, and so forth to the code in `SimpleMBSDemo1` or `SimpleMBSDemo2`. No other change is necessary.

Note to teachers: If you are familiar with interfaces and have looked ahead to Chapter 2, you may wonder why the two simple demo programs declare the environment to be of type `BoundedEnv` rather than `Environment`. The reason is to avoid having to use and explain the following line of code.

```
  Environment env = new BoundedEnv(NUM_ROWS, NUM_COLS);
```

The problem is that students may not be familiar with seeing a variable declared to be of one type (`Environment`) and then constructed as an instance of another type (`BoundedEnv`). If you (or your students) have looked ahead to Chapter 2, you may have noticed that the `Simulation` constructor takes an `Environment` as a parameter. This means that it can take a `BoundedEnv` object, or an object of any other class that implements the `Environment` interface.

Teaching tip: Steve Andrianoff and David Levine of St. Bonaventure University have written a series of role-playing exercises to help students understand class responsibilities and the interactions between objects. One of these is a Marine Biology Simulation role-playing exercise. You can find links to this and other role-playing examples at AP Central. Role-playing exercises take some practice and the one for the *AP Marine Biology Simulation Case Study* is complex. Starting your students with the exercise labeled "first", will help them understand the role-playing process before they attempt the more involved MBS exercise. Prior to introducing Chapter 2, it would be helpful to have your students participate in these role-playing exercises.

Chapter 2

Guided Tour of the Marine Biology Simulation Implementation

Before you introduce this chapter, your students should be familiar with the basics of how to read classes (including constructing and using objects), basic flow control, and one-dimensional arrays. The case study could, however, be used as an instrument for learning `ArrayList` and inheritance (Chapters 2 through 4) and two-dimensional arrays and interfaces (Chapter 5).

The Big Picture (page 19)

Ask your students to think of other examples of two-dimensional grids with which they are familiar, such as tables, game boards, and geographic maps.

After you have talked about "The Big Picture" and before you go on to "What classes are necessary?" you might want to have students pre-read and discuss the section on black-box test cases, found on pages 42 and 43. This will start your students thinking about the many fish configurations that need to be considered as code is developed.

What do the core classes look like? (page 20)

If you are using an applet, main method, or graphical user interface from another source, you may want to talk about it at this point. If the code is simple enough, you may even want to have your students look at the code. If you are using the graphical user interface that comes with the case study, you can simply point out that the graphical user interface is calling the `Simulation step` method every time the user presses the Step button.

The `Simulation` Class (pages 21–23)

The code for the `Simulation` constructor on page 21 may seem unusual since it calls a method to display the environment. If students question this design decision, point out the last sentence in the second paragraph on page 22 telling us that when a `Simulation` object is constructed, the initial environment configuration will be displayed. This is how we see the fish in their initial positions.

The first paragraph on page 22 introduces the `EnvDisplay` interface. Use of the terms "specifies" and "implements" here is analogous to the use of the terms "declares" and "defines" in C and C++.

The word “interface” has two different (but related) meanings in Java.

- the specification and documentation of the signatures of public constructors and methods in a class (similar to a header file in C or C++)
- the specification of an abstract type (a list of method signatures with no implementations) defined with the `interface` keyword

To reduce confusion, the MBS case study uses the term as described in the second bullet to mean an abstract type defined with the `interface` keyword. It uses phrases like “class documentation” and “list of methods” to refer to public information about a class, such as its method signatures, that would be useful to client code.

The `step` method shown on page 21 may require more explanation than is found on page 22. Conceptually, the method is quite simple. It asks for a list of all the objects in the environment and then loops through the list asking each one to perform its task. The complexity is mostly in the use of types. What is the type of things in the environment? While the environment was written to be quite generic, it does have one basic requirement for all objects that can be stored in it — they must keep track of their location and be able to report the information using a `location()` method. How does one specify “any class that supports a `location()` method”? This is exactly what the `Locatable` interface introduced on page 22 does. It specifies a single method, `location`. Objects of any class that implements this interface may be stored in an environment. That is how the environment keeps track of them. This explains why the `allObjects` method returns an array of `Locatable` objects.

The `step` method could treat the objects referenced by `theFishes` as `Locatable` objects if it only needed to ask each object for its location, but this is not the case. It wants to ask them to `act`, and `Locatable` objects do not necessarily have an `act` method. The `step` method, though, knows more about the objects than the environment does. It knows that this is a marine biology simulation, that the objects are all fish, and that the `Fish` class has an `act` method. Therefore, it *casts* the `Locatable` references in the array as `Fish` references. When you cast a value to another type (by putting the type in parentheses before the value), you are asking the system to create another reference to the object that is of the specified type. In this case, the code is telling the compiler that the elements in `theFishes` array actually refer to `Fish` objects. In order to ask these objects to perform `Fish` behaviors (like `act`), they must be accessed using a `Fish` reference. This cast will only succeed if the object actually is a `Fish` (or a subclass of `Fish`). If the object is not a `Fish`, the code will still compile, but at runtime the program will throw an exception (print an error message and stop).

Teaching tip: Although students have seen exactly three fish in `SimpleMBSDemo1` and `SimpleMBSDemo2`, the diagram depicts four `Fish` objects. Be sure your students do not think in terms of a given example, but rather generalize to any number of `Fish` objects being told to `step`.

Analysis Question Set 1 (page 24)

1. `step` uses an array of `Locatable` objects (fish) that it gets from calling `theEnv.allObjects()`. The `Simulation` constructor is passed the environment as one of its parameters.

2. The `Simulation` class encapsulates the process of moving fish. This means `SimpleMBSDemo2` can construct a `Simulation` object and call its `step` method and expect the program to run correctly without knowing the implementation of the `Simulation` class. So while Pat wondered what was going on when `SimpleMBSDemo2` was run, the fact was that the program did its job by constructing the `Simulation` object and calling method `step`. In particular, `SimpleMBSDemo2` does not need to display the initial configuration of the environment because the `Simulation` constructor does that. Nor does `SimpleMBSDemo2` ask the fish to act or display the environment at the end of the timestep, because the `step` method does both of those things.

Teaching tip: If you have covered interfaces in your class, you can be more specific with your students. `Environment` is, in fact, an interface, not a class. It can be implemented in a number of different ways. The `BoundedEnv` class is an implementation of the `Environment` interface that models a bounded rectangular grid.

The `Environment` Interface (pages 24–25)

Note to teachers: You may notice that the `allObjects` method, described on page 22, returns an array of `Locatable` objects, the `neighborsOf` method returns an `ArrayList`. Your students may ask why not use an array for both, or an ArrayList for both? This is a good time to discuss the advantages and disadvantages of both data structures.

Array

- An array is typed. You must declare the type of element the array will hold. Arrays can hold primitive types or object references.
- You may be able to avoid casting when you use an array. In the case of the `allObjects` method, which returns an array of `Locatable` references, you do not have to cast if only the `location` method needs to be invoked. The `step` method of the `Simulation` class calls the `allObjects` method and does cast each element of the array to a `Fish` reference in order to tell each fish to act.

- An array's size is fixed when it is instantiated. You cannot grow or shrink the physical size of an array. If the array's size needs to be increased or decreased, a new array must be constructed and all the values to keep must be copied over to the new array. Once this is done, the reference to the old array must be assigned to the new array. The old array will then be garbage collected.
- Insertions and deletions in an array require the programmer to code shifting the elements of the array.

`ArrayList`

- An `ArrayList` holds only `Object` references. You must cast each element of an `ArrayList` to the specific type of reference needed in order to invoke methods of the object's specific class. Using an `ArrayList` element's reference will allow the programmer to invoke `Object` methods. *(Note: Versions of Java that include the use of generics will allow the programmer to type an `ArrayList`. This was not available when the case study and teacher's manual were written.)*
- Currently, primitive type variables must be "wrapped" in a corresponding wrapper class object to be added to an `ArrayList`. *(Note: Future versions of Java may include automatic boxing and unboxing of primitives. This was not available when the case study and teacher's manual were written.)*
- An `ArrayList` can shrink and grow as needed. Invoking the `add` and `delete` methods will change the size of an `ArrayList`. The `add` and `delete` methods take care of shifting the elements of the `ArrayList`.

In general, if the size of the data set is known and isn't going to change, an array might be the right choice. When the size of the data set is not known or is volatile (many insertions and deletions), an `ArrayList` may be the right choice.

In the context of this case study, the reasons for choosing an array in one case and an `ArrayList` in the other case is as follows:

- The number of objects in the environment is known. The environment contains `Locatable` objects. Returning a fixed size array of `Locatable` object references is a reasonable design decision.
- The `neighborsOf` method could return as few as two neighboring locations to as many as four or eight neighboring locations with the original environment implementations. Since the size is not known, returning an `ArrayList` is also a reasonable design decision.

If you have not covered constants, you may want to teach the syntax for using constants in client and class code. For example, the code within the `Direction` class can use the NORTH constant by just referring to it directly, but code outside the `Direction` class must specify the class in which NORTH is defined, i.e., `Direction.NORTH`.

Analysis Question Set 2 (page 26)

Teaching tip: Have students do the analysis questions as a paper-and-pencil exercise first. Discuss their answers before doing the implementations in the exercise set that follows.

Note to teachers: As an option, if you have covered the difference between the `equals` method and `==` operator, you may also wish to have students answer the questions in Analysis Question Set 5 at this point.

These exercises focus on the `Location` and `Direction` classes and their methods. Students will be responsible for understanding how to use any of the testable methods in these classes. This includes the `compareTo` method. Depending on when you cover this material, your students may or may not be ready for `compareTo`. If they are not, remember to revisit this topic when you discuss interfaces or searching and sorting.

1. (6, 3), (7, 4), (8, 3), (7, 2)

2. EAST

3. SOUTH, NORTH

4. (6, 3)

5. (5, 3)

6. *Note:* `Direction` and `Location` are immutable classes, meaning that the state of one of their objects never changes once it is constructed.

   The `Location` class has

   - a single constructor with row and column parameters.

Besides `row`, `col`, and `equals`, `Location` has `compareTo`, `hashCode`, and `toString`.

- `compareTo` compares two locations for ordering. (For instance, you could use the `compareTo` method if you wanted to sort fish by their locations. The `compareTo` method uses row-major ordering for its comparisons; two locations in different rows are ordered based on their row number, while two locations in the same row are ordered based on their column number.) This actually isn't discussed until page 87 in Chapter 5, but it may come up in your class at this point. Remember, Chapter 5 material is tested on the AB exam only.
- `hashCode` generates a hash code for this location (necessary because `Location` redefines the `equals` method, but students are not expected to know this technical detail and `hashCode` will not be tested).
- `toString` represents this location as a string and is used to output a location in a readable format. The `toString` method can be used to print a fish's location for testing and debugging.

The `Direction` class has three constructors.

- a default (no arguments) constructor that sets a direction to `North`
- a constructor with a single parameter `degrees`
- a constructor with a single parameter `direction` in `String` form (i.e., "NORTH", "North", or "north")

`Direction` contains the following `public` methods

- `inDegrees` returns the direction value in degrees (where 0 degrees is `North`)
- `equals(Object other)` returns `true` if `other` represents the same direction as this direction and `false` otherwise
- `toLeft` returns a direction that is 90 degrees to the left of this `Direction`
- `toRight` returns a direction that is 90 degrees to the right of this `Direction`
- `toLeft(int degrees)` returns a direction `degrees` to the left of this `Direction`
- `toRight(int degrees)` returns a direction `degrees` to the right of this `Direction`
- `reverse` returns the direction that is the reverse of this `Direction` object
- `toString` that represents this direction as a string
- `randomDirection` returns a random direction in the range of [0,360) degrees

- `roundedDir` rounds this direction to the nearest “cardinal” direction and returns that direction (will not be tested)
- `hashCode` generates a hash code for this `Direction` object (will not be tested)

The `toString` method can be used to report a direction and for testing and debugging purposes.

Exercise Set 1 (page 26)

1. Use `SimpleMBSDemo1` as a template. Make changes as shown below and save the code using a different name. Be sure to change the comments to reflect the exercise. The sample solution is only one possible driver program.

```
/** Chapter 2, Exercise Set 1, Question 1
 *  Write a simple driver program that constructs a BoundedEnv
 *  environment and then tests answers to Analysis Question Set 2.
 **/

public class ExSet1Q1
{
  // Specify number of rows and columns in environment.
  private static final int ENV_ROWS = 20;  // rows in environment
  private static final int ENV_COLS = 20;  // columns in environment

  /** 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);

    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);

    System.out.println("loc1 neighbors:  " + env.neighborsOf(loc1));
    System.out.println("dir1 = " + dir1);
    System.out.println("dir2 = " + dir2 + " dir3 = " + dir3);
    System.out.println("loc3 = " + loc3);
    System.out.println("loc4 = " + loc4);
  }
}
```

2. One way to do this is to add the following lines of code.

```
System.out.println("In degrees, North = " + Direction.NORTH.inDegrees());
System.out.println("In degrees, South = " + Direction.SOUTH.inDegrees());
System.out.println("In degrees, East = " + Direction.EAST.inDegrees());
System.out.println("In degrees, West = " + Direction.WEST.inDegrees());

System.out.println("dir3 in degrees = " + dir3.inDegrees());
```

When the students run this program they will discover that North is 0, East is 90, South is 180, West is 270, and `dir3` is 0.

The `Fish` Class (pages 27–39)

The case study can be used to introduce `ArrayList`. The sections of this chapter that deal with the `nextLocation` and `emptyNeighbors` methods of the `Fish` class introduce some of the `ArrayList` methods. Students are expected to know these methods.

Why do fish have a private environment? Actually they don't. They have a reference to `theEnv`. In `SimpleMBSDemo1` and `SimpleMBSDemo2`, students can see that each fish, when constructed, is passed a reference to the single environment in which they all exist. A fish has to know the particular environment in which it is swimming. Think of all of the fish in a given environment pointing to the same place. There may, in a variation of the case study, be more than one environment (two or more lakes or streams, for example) or more than one simulation running, each with its own fish population.

The first paragraph on page 29 discusses the `initialize` method. If your students are not familiar with the terms "instance" and "instantiate", you may need to explain why "instance variables" are referred to that way. Remember, they are variables that store the state of a given instance of a class object.

The same paragraph states that some textbooks use the term "field". It is important to note that class documentation generated using Sun's standard `javadoc` tool also uses the term "field". The three categories in class documentation are fields, constructors, and methods.

When you teach the `randomColor` method on page 30, discuss with your students the need for the import statements for `java.util.Random` and `java.awt.Color`. This is also a good time to talk about the differences between instance variables, local variables and parameters.

Analysis Question Set 3 (page 31)

1. two
2. 2(2, 6)North, 1(7, 3)South (directions may vary)
3. false
4. true
5. 2(2, 6)North
6. null
7. No. A `Fish` constructor adds the fish to the `Environment`, so there is no reason to add it again. It is critical that the fish and the environment agree on the fish's location at all times. This is why a fish adds itself in its constructor, thus ensuring that the fish and the environment agree on the location as soon as the fish is constructed. (There is more discussion of this and what it means for a fish to be in a *consistent state* on page 33 of the case study narrative.)
8. You wouldn't want client code to be able to modify a fish's state (e.g., change its location, direction, or color) during program execution by calling the `initialize` method directly. For example, if `initialize` were a `public` method, client code could move a fish to an arbitrary location at any time.
9. While it is certainly possible to code `theEnv` as a class variable, it would be inadvisable to do so. There is no guarantee that all fish live in the same environment. Creating `theEnv` as a class variable would force all fish to live in the same environment and would not allow the two simulations with two different environments to be displayed at the same time.

Note to teachers: Although class variables are not in the subset, the decision was made to create `nextAvailableID` variable as a class variable. Since `nextAvailableID` needs to be shared by all of the `Fish` objects, it cannot be an instance variable. An alternate design decision could be to pass the ID number as a parameter and not have the `nextAvailableID` class variable in the `Fish` class. You may want to discuss this design decision with your class. It is better to have the `Fish` class determine the ID number for each `Fish` object, or is it better to have some other class, like an environment class, determine the ID number of each `Locatable` object?

Exercise Set 2 (page 32)

1. One possible solution is given below.

```
/** Chapter 2, Exercise Set 2, Question 1
 *  Write a simple driver program that constructs a BoundedEnv
 *  environment and then tests answers to Analysis Question Set 3.
 **/

public class Ch2ExSet2Q1
{
  // Specify number of rows and columns in environment.
  private static final int ENV_ROWS = 20;    // rows in environment
  private static final int ENV_COLS = 20;    // columns in environment

  /** 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);

    // Set up the information given in Analysis Question Set 3
    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);

    // Question 1
    System.out.println("number of objects:  " + env.numObjects());

    // Question 2
    System.out.println("Objects in the environment are: ");
    Locatable[] fishList = env.allObjects();
    for ( int index = 0; index < fishList.length; index++ )
    {
      System.out.println((Fish)fishList[index]);
    }

    // Question 3
    System.out.println("Is loc1 empty? " + env.isEmpty(loc1));

    // Question 4
    System.out.println("Is loc3 empty " + env.isEmpty(loc3));
    // Question 5
    System.out.println("The object at loc2 is  " +
                         (Fish)env.objectAt(loc2));

    // Question 6  (documentation says should return null)
    System.out.println("The object at loc3 is  " +
                         (Fish)env.objectAt(loc3));
  }
}
```

2. Answers will vary. One possible solution is to modify the code from Question 1.

```
/** Chapter 2, Exercise Set 2, Question 2
 *  Write a simple driver program that constructs a BoundedEnv
 *  environment and then tests answers to Analysis Question Set 3.
 **/

public class Ch2ExSet2Q2
{
  // Specify number of rows and columns in environment.
  private static final int ENV_ROWS = 20;    // rows in environment
  private static final int ENV_COLS = 20;    // columns in environment

  /** 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);

    // Set up the information given in Analysis Question Set 3
    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);

    // Add more fish
    Fish f3 = new Fish(env, new Location(0, 0));
    Fish f4 = new Fish(env, new Location(2, 7));
    Fish f5 = new Fish(env, new Location(7, 7));

    // Question 1
    System.out.println("number of objects:  " + env.numObjects());
    // Question 2
    System.out.println("Objects in the environment are: ");
    Locatable[] fishList = env.allObjects();
    for ( int index = 0; index < fishList.length; index++ )
    {
      System.out.println((Fish)fishList[index]);
    }

    // Now remove some fish and print all objects again
    env.remove(f2);
    env.remove(f4);

    System.out.println("After removing f2 and f4, " +
                       "objects in the environment are:  ") ;
    fishList = env.allObjects();
    for ( int index = 0; index < fishList.length; index++ )
    {
      System.out.println((Fish)fishList[index]);
    }
    System.out.println("Is the location of f1 empty? " +
                        env.isEmpty(f1.location()));
    System.out.println("Is the location of f2 empty? " +
                        env.isEmpty(f2.location()));
  }
}
```

3. Adding the following statement to the above code

   ```
   env.add(new Fish(env, new Location(8, 8)));
   ```

   results in an Illegal Argument Exception that says the location is not a valid empty location. When the call to `env.add` is actually executed, the `Fish` constructor has already added the fish to the environment at location (8, 8), so that location is no longer empty. This behavior is subtle but important to understand. This question exists as an exercise as well as an analysis question (Question 7 on page 31) to help students understand that calling a `Fish` constructor adds a fish to the environment; therefore, there is no need for a method that creates a fish to also make a call to the environment to add that fish. Students may be more likely to remember this behavior if they have run the program and seen it generate an exception.

Simple accessor methods in `Fish` (pages 32–33)

After reading this section, students should look at the actual `Fish.java` file to find the "`implements Locatable`" phrase.

Analysis Question Set 4 (page 33)

1. Look at the `Fish initialize` method. The last step is to add this fish to `theEnv`. The environment receives this fish, which includes its location, and adds the fish to that location if it is not already occupied by a fish. So the fish and the environment agree upon the fish's location and therefore the fish starts out in a consistent state.

2. If a fish (or any object that keeps track of its own location) is removed from the environment (e.g., it dies or is eaten by a shark), then it will still "think" it's at a particular location when it is not. This will put the fish in an inconsistent state. Furthermore, if the `Environment` method `recordMove` does not work as intended, the fish could change to an inconsistent state.

`Fish` movement methods — `act` and its helper methods (pages 33–34)

Your students may wonder why `act` just checks to see if the fish is still in the environment and then calls `move`. Why not get rid of the `act` method and just have a `move` method? The answer is because fish may eventually do more than just move. They could breed, die, eat, blow bubbles, and so on. Creating a "controller" method, like `act`, is a good design decision. It allows the different behaviors of a fish to be factored out into separate methods that can be called by the `act` method. This is especially advantageous when your students begin to create subclasses of the `Fish` class. With each of the different `Fish` behaviors in a separate method, it will be easy to override only the behaviors that need to be redefined.

Analysis Question Set 5 (page 35)

1. true, true

2. false, false

3. false, true

Analysis Question Set 6 (page 36)

1. The `neighborsOf` method returns **all** valid neighboring locations, not just those that are empty. The `emptyNeighbors` code that obtains a fish's empty neighbors from the environment could have been included in `nextLocation` but we want each method to perform one well-defined task. Including the code from `emptyNeighbors` in `nextLocation` would have over-complicated `nextLocation` and made it less readable.

The `emptyNeighbors` method (page 36)

After reading the code for `emptyNeighbors` on page 36, remind your students that we do not know how many empty neighbors there will be. This motivates using an `ArrayList` rather than an array. Note that we need an `import` statement when we use `ArrayList`.

Exercise Set 3 (page 37)

1. (0, 0) has two neighbors; (0, 1) has three neighbors; (1, 1) has four neighbors. One possible code solution is the following.

```
/** Chapter 2, Exercise Set 3, Questions 1 and 2
 *  Write a simple driver program that constructs a BoundedEnv
 *  enviromnment and then tests answers to Analysis Question Set 3.
 **/

import java.util.ArrayList;

public class Ch2ExSet3Q1and2
{
  // Specify number of rows and columns in environment.
  private static final int ENV_ROWS = 20;    // rows in environment
  private static final int ENV_COLS = 20;    // columns in environment

  /** 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);

    // Set up the information given in the problem
    Location loc1 = new Location(0, 0);
    Location loc2 = new Location(0, 1);
    Location loc3 = new Location(1, 1);

    // Answer Exercise Set 3 Question 1
    ArrayList nbrs1 = env.neighborsOf(loc1);
    ArrayList nbrs2 = env.neighborsOf(loc2);
    ArrayList nbrs3 = env.neighborsOf(loc3);
    System.out.println("Location " + loc1 + " has " +
                       nbrs1.size() + " neighbors");
    System.out.println("Location " + loc2 + " has " +
                       nbrs2.size() + " neighbors");
    System.out.println("Location " + loc3 + " has " +
                       nbrs3.size() + " neighbors");

    // Answer Exercise Set 3 Question 2
    System.out.print("The neighbors of location " + loc1 + " are: ");
    for (int index = 0; index < nbrs1.size(); index++)
    {
      System.out.print(nbrs1.get(index) + "  ");
    }
    System.out.println();

    System.out.print("The neighbors of location " + loc2 + " are: ");
    for (int index = 0; index < nbrs2.size(); index++)
    {
      System.out.print(nbrs2.get(index) + "  ");
    }
    System.out.println();

    System.out.print("The neighbors of location " + loc3 + " are: ");
    for (int index = 0; index < nbrs3.size(); index++)
    {
      System.out.print(nbrs3.get(index) + "  ");
    }
    System.out.println();
  }
}
```

2. The neighbors in (row, column) format are:

   for (0, 0): (0, 1) and (1, 0)
   for (0, 1): (0, 2), (1, 1), and (0, 0)
   for (1, 1): (0, 1), (1, 2), (2, 1), and (1, 0)

3. Specific locations will vary, depending upon the size of the bounded environment. However, the four corners will all have two neighbors, all other boundary locations will have three neighbors and all interior locations will have four neighbors.

Analysis Question Set 7 (page 38)

You might suggest that your students label the diagrams a, b, c, d. The answers below follow from thinking about the diagrams as if they were labeled in this way.

1. In each diagram the fish at location (1, 0) has three neighbors.

2.	a. 1	b. 1	c. 2	d. 3
3.	a. none	b. (1, 1)	c. (1, 1), (2, 0)	d. (0, 0), (1, 1), (2, 0)

The `changeLocation` and `changeDirection` methods (page 39)

Technical detail for interested teachers: Consider the code for `changeLocation` and `changeDirection` on page 39. Since `myLoc` is `private`, not `protected`, subclasses cannot change `myLoc` directly. Even if they redefine `changeLocation`, which they can do since it is not `private`, they must use `super.changeLocation` (i.e., the version of `changeLocation` in the `Fish` superclass) to actually modify the location. This ensures that `myLoc` is never changed without also updating the environment.

Analysis Question Set 8 (page 40)

1. For the fish in location (2, 2)

   - `move` will call `nextLocation`
   - `nextLocation` will call `emptyNeighbors`
   - `emptyNeighbors` will first get a list of the four neighbors around location (2, 2), `nbrs`, and then determine and return a list of empty neighbors, (1, 2), (2, 1), and (2, 3)
   - `oppositeDir` will then be set to the location behind the fish, (2, 1), and that location will be removed from the list of empty neighbors
   - if there are no empty neighbors from which to select a new location, the fish's current location is returned, but in this case there are still two empty neighbors left.
   - otherwise, an index is randomly selected and the location of the empty neighbor at that index is returned
   - the fish in location (2, 2) might move to (1, 2) or (2, 3), depending on the randomly chosen index.

2. A new instance variable, `myAge`, would have to be initialized in the `initialize` method and incremented (probably in the `act` method) according to some rule, such as “upon each timestep the fish age is incremented by 1”. In order to access the age, you will need an accessor method `age`. You might also want to modify the `toString` method to show the age.

Exercise Set 4 (page 40)

Remember, you can check the `Documentation` folder for help with constructors and methods available in each class.

1. The code for changing the color is a simple method as shown below.

```
/** Changes fish's color to the parameter newColor
**/
public void changeColor(Color newColor)
{
  myColor = newColor;
}
```

2. The code below shows one way to construct fish using the third `Fish` constructor.

```
  // Chapter 2, Exercise Set 4, Question 2
  // Demonstrates initializing direction and color
  // If you want to initialize all fish to the same
  // direction and color, use variables as shown with f1,
  // or individualize the fish as with f2 and f3
  Direction dir = new Direction("east");
  Color col = new Color(200, 50, 50);
  Fish f1 = new Fish(env, new Location(2, 2), dir, col);
  Fish f2 = new Fish(env, new Location(2, 3),
                   new Direction("west"),  new Color(100, 150, 50));
  Fish f3 = new Fish(env, new Location(5, 8),
                   new Direction("south"),  new Color(50, 50, 100));
```

The modification shown below implements the suggested changes made in this question. Your students may be much more creative with this exercise.

```
  for ( int i = 0; i < NUM_STEPS; i++ )
  {
    if (i % 2 == 0)     //even numbered step
    {
      f1.changeColor(Color.red);
    }
```

```
      else
      {
        f1.changeColor(Color.yellow);
      }
      sim.step();
    }
```

The `Color` class is not part of the AP Java Subset so you may not want to spend much time on it, but a fun follow-up exercise is to use the `changeColor` method to make fish become lighter (or darker or change color) with age. To do this, have students research the `Color` class to find methods that lighten or darken a color. Then call the `changeColor` method from the `act` method, passing it a lightened or darkened color `(color().lighter(), myColor.lighter(), or color().darker())`.

Test Plan (pages 41–43)

Consider the bulleted list of test cases on page 42. Some of your students may comment that the fifth test case, "A file with a fish in every location …", covers the sixth test case, "A fish with no empty locations around it …". Your students would have a good point but the "fish in every location" is a special case, a boundary case. The "fish with no empty neighbors" is a common, more general case that fits in the sequence of cases that follow it. Just because another case subsumes a given case is not necessarily a reason to leave it out. You might not develop a separate test run for "fish with no empty neighbors". (Here we are distinguishing between test cases and test runs.) One frequently comes up with test runs that test several cases simultaneously. The "fish in every location" test case describes one test run. But, as you can see from the paragraph and diagram at the top of page 43, a single test run may be used to describe several test cases. The "fish with no empty neighbors" case could be part of the "fish in every location" test run or you could create a separate file to test "fish with no empty neighbors". Be careful though, because if one of the fish in the neighboring locations is able to move before the "fish with no empty neighbors", that fish may then have an open place to which it can move.

Using the `fish.dat` file as the basis of a test run covers several of the test cases. Not only does it test for a fish with one valid empty neighboring location (the fish in the lower right corner) but also the fish at (0, 3) and (2, 2) will have two valid empty neighboring locations and the fish at (3, 2) will have three after the fish at (2, 2) moves.

Note to teachers: Although your students may not discover this yet, fish move in row-major (top-down, left-to-right) order. If it comes up, you might want to have students do Questions 3 and 5 on page 45 at this point.

Analysis Question Set 9 (page 43)

1. There might be a fish whose location is outside the bounded environment or the environment dimensions might be invalid. The test cases involving invalid input and multiple fish in the same location can be tested with appropriate initial configuration files.

2. Students should list one test case for each of the three conditions. Answers will vary. Several of the black-box test cases cover one or more of the named conditions:

   - a file with a single fish in an environment bigger than 1 x 2 covers the cases of multiple neighbors and an empty neighbor
   - a file with a fish in every location (multiple neighbors, a non-empty neighbor)
   - a file with a fish with no empty locations around it (multiple neighbors, a non-empty neighbor)
   - a fish with a single adjacent empty location in front or to the side (multiple neighbors, an empty neighbor, and a non-empty neighbor)
   - a fish with a single adjacent empty location behind it (multiple neighbors, an empty neighbor, and a non-empty neighbor)
   - a fish with two adjacent empty locations in front or to the side (multiple neighbors, an empty neighbor, and a non-empty neighbor)
   - a fish with three adjacent empty locations in front and to the side (multiple neighbors, an empty neighbor)

3. The environment could have exactly one row and two columns or one column and two rows. In each case there are exactly two locations, one occupied by the fish and the other empty.

Exercise Set 5 (page 45)

It is a good idea to copy the data files you plan to use into the same folder as the class files. If you copy the data into the `Code` folder or keep them in the `DataFiles` folder, when you click on `File/Open environment file`, you will need to click on the drop down menu to navigate to the folder where the data files are found.

1. Remind students that they are always starting with the same data file, say `fish.dat`, so the fish will always begin in the same configuration. If they use the same seed number, the color and movement will be the same each time the program is run. If they change seed numbers, the color and movement will be different from one run to another.

2. The text output tells you the ID, location, and direction of each fish at each step.

3. Below is a table showing a sample run using the `fish.dat` file. (You will find a blank master for this table in Appendix C.) This run tested the cases in which a fish faces a boundary and the more general cases of a fish selecting between one, two, or three available locations.

Your students should observe each step carefully, noticing the test cases for each fish at each step. They should make up a table like the one below and chart each fish's location and direction. They might also note the locations on each step that represent special test cases. They need to look closely as each step is executed because it's hard to reconstruct the way the grid looked from the text output after the five steps have been completed, except by rerunning with the same initial configuration and the same seed.

Step 0 represents the fish in their initial positions from the `fish.dat` file.

Step	Fish 1	Fish 2	Fish 3	Fish 4	Fish 5	Fish 6
0	(5, 5) North	(3, 3) South	(10, 10) East	(6, 8) West	(2, 10) North	(1, 1) South
1	(5, 6) East	(4, 3) South	(9, 10) North	(5, 8) North	(2, 9) West	(2, 1) South
2	(4, 6) North	(4, 4) East	(9, 11) East	(4, 8) North	(3, 9) South	(3, 1) South
3	(4, 7) East	(5, 4) South	(8, 11) North	(4, 9) East	(3, 8) West	(3, 2) East
4	(5, 7) South	(5, 5) East	(7, 11) North	(5, 9) South	(2, 8) North	(3, 3) East
5	(5, 8) East	(6, 5) South	(7, 10) West	(5, 10) East	(2, 9) East	(3, 4) East

4. The point is to save the configuration of the environment to a file, so that you can start up again at that point or so that you could run regression tests later with the same input file and same initial seed, and verify that the state after 5 (or 10) steps is the same as before. This can be done from the file menu in the GUI that is distributed with the case study.

5. The order is top-down, left-to-right (row major) order. As each fish moves, it can change the available locations for a fish that has not yet had its turn to move. Note, however, that the class documentation for the `Fish` class does not specify this particular ordering. Consider possible fish movement if the order was column major instead.

6. Answers will vary. Here are some examples based on `fish.dat`. Each represents a separate data file to be saved with an appropriate name.

   - `invalidenv.dat`:
     bounded 12
   - `emptyenv.dat`:
     bounded 12 12
   - An example of a file with a single fish is `onefish.dat`, provided in the data files.
   - `twofishsameloc.dat`:
     bounded 12 12
     Fish 5 5 North
     Fish 3 3 South
     Fish 5 5 East
   - An example of a file with a fish in every location is `fullenv.dat`, also provided in the data files.

You can use `fullenv.dat` for a fish with no empty locations around it. You could also add a fish that is surrounded by other fish to `fish.dat`, but be careful. Remember that one of the surrounding fish might move before the fish in the middle (top-down, left-to-right), leaving the fish that was surrounded with a place to move into. You might challenge your students to come up with a situation in `fish.dat` that meets this criterion.

7. Test cases required for `move`:

   - A fish with no available locations where it can move should remain in its current location.
   - A fish with at least one available location where it can move should move to that location.

Test cases required for `nextLocation`:

- A fish for which there are no empty neighbors or where the only empty neighbor is the one behind it should return its current location.
- A fish for which there is at least one valid available location, in front, to the left, or to the right, should return one of those locations.

Test cases required for `emptyNeighbors`:

- A fish with no valid adjacent neighbors should return an empty `ArrayList`.
- A fish with four or fewer adjacent locations, all containing valid fish, should return an empty `ArrayList`.
- A fish with at least one empty adjacent location, should return a list of empty neighbors (possibly including the location behind the fish).
- A fish with four empty adjacent locations should return a list of all four locations (including the location behind the fish).

Analysis Question Set 10 (page 47)

These questions are open-ended and should lead to good classroom discussion.

1. The last scenario corresponds to the design chosen by the original programmer.

2. Answers will vary. This question provides a rich opportunity to discuss design decisions and changes in the implementation and parameter lists of various methods as they are shifted from one class to another.

3. Answers will vary. Students should describe (and justify) their reasoning. It is often surprising how alternative designs can sound perfectly reasonable with the right justification. You could have students try to represent the design with CRC (Class-Responsibilities-Collaborators) cards to help decide which alternatives represent reasonable design choices. Another way to discuss designs is to look at the *cohesion* of the various modules (Do they make sense? Do they hold together conceptually?) and the *data coupling* (Are you passing lots of values all around, or is the encapsulation good enough that you don't have to pass lots of data to many different objects?). You might develop (or have your students develop) a role-play script for each scenario to demonstrate its feasibility.

4. Answers will vary. Students should describe (and justify) their reasoning.

5. Answers will vary. Students should describe (and justify) their reasoning. Encourage discussion.

Chapter 3

Creating a Dynamic Population

Design and Implementation (pages 52–53)

Teaching tip: If you covered Chapter 4 before covering Chapter 3, you may wish to have students discuss the reasons for making the `breed` and `die` methods `protected` rather than `private` at this point. As you cover the `breed` method, your students should notice the similarity between the first few lines of `breed` and the first few lines of the `move` method in the `SlowFish` class.

Note to teachers: If you are covering the chapters in order, you may want to read ahead in Chapter 4 for an explanation of `protected` versus `private`, even though you do not need to go into it with your students at this time.

Note to teachers: The `Fish` modifications necessary for Chapter 3 are described and contained in the `FishModsForChap2.txt` file found in the `Code` folder. The `Fish` class (recall that this is the version of the `Fish` class that will be tested on the APCS Exam), complete with modifications, is found in the `DynamicPopulation` folder inside the `Code` folder.

Analysis Question Set 1 (page 56)

1. No, the `Simulation` object asks the environment for a list of all its objects (`allObjects`), so it has a reference to each fish also. That reference still exists after the environment has removed the fish. This is why the `act` method in `Fish` calls `isInEnv`. See the Analysis Question Set 4 in Chapter 2 for related questions. (A class that implements `EnvDisplay` may also ask the environment for a list of all the objects in order to display them, but it doesn't need to keep track of its list while other classes are manipulating the environment. It can ask for a new list and immediately display the objects each time the `showEnv` method is called.)

2. The change would be advantageous if behavior were added to the `act` method that depended on whether or not the fish had moved. There are no disadvantages other than, in the current context, the `act` method would end up ignoring the return value of a method call (pedagogical disadvantage rather than a design disadvantage).

Testing (pages 58–60)

Teaching tip: The test results documented in the case study were generated using the `fish.dat` configuration file and a seed of 17.

Analysis Question Set 2 (page 60)

1. Answers will vary. Students may say they would create separate methods because each has its own task. Some may say they would not create separate methods because it's easier to think of these tasks as part of `act` or `move`. You can expect a variety of responses.

2. You might easily put the test for empty neighbors in the `act` method. If the test was false, your code could bypass both attempting to breed and attempting to move. However, the code is more readable and true to the simulation as it is written. That is, a fish first attempts to breed. If the fish does not breed, then it attempts to move. Students may argue that the test for a fish dying should have been placed in the `die` method for consistency. There are arguments on either side of this design decision.

3. The main advantage is readability. Sometimes adding a boolean variable makes reading more straightforward. On the other hand, unnecessary variables may increase complexity and actually cause code to be less readable. Again, this is a design decision that reflects the programmer's (or design team's) bias. Point out that in-line commenting such as the comment following `if (! breed())` in `act` (see page 57) often greatly improves readability.

Teaching tip: Teachers who have covered Chapter 4 before Chapter 3 may wish to ask students to discuss the relative merits of modifying the original `Fish` class to add breeding and dying behavior versus creating a subclass with this behavior.

Exercise Set 1 (page 61)

Teaching tip: Since two versions of the `Fish.java` file are used in the case study (and you may add more variations, as in the change color exercise), you need to be careful that you are using the correct version and do not accidentally overwrite a version that you want to keep. One way to do this is to save a copy of each file using a descriptive name such as `FishOrig.java` for the Chapter 2 version and `FishBreed.java` for the Chapter 3 version. These files cannot be compiled because the names do not match the class. Be sure to save the appropriate version as `Fish.java` and compile.

1. Following the instructions and using `fish.dat`, step through the simulation. Watch to see which fish breed, which die, and which "family" of descendants gradually populates the environment.

2. The results from Chapter 2 should be quite different from the saved results in Chapter 3. Fish did not breed and die in Chapter 2. The Chapter 3 files will probably show that the `Fish` population is changing. The changing population and the fact that random numbers are also being generated for breeding and dying should not only change the sequence of random numbers generated for fish movement, but also the possibilities of where a fish can move.

   Note to teachers: The `MBSGUI` allows the user to save an experiment configuration and restart the simulation from the saved configuration at a later time using the same seed.

3. Rather than make a permanent change, just comment out the color parameter as shown below. The code becomes a call to the three-parameter fish constructor.

```
Fish child = new Fish(environment(), loc,
                      environment().randomDirection()/*, color()*/);
```

It is now very difficult to keep track of the original fish and their offspring. In fact, it's hard to tell which fish have bred. You can still identify when a fish dies, except in the event that another fish breeds into the same location and the new fish is the same color as the fish that died. You would have to look at the debug output to see if the ID of the fish in that location changed.

Teaching tip: The `Color` class is not part of the AP subset so you may not want to spend any more time on it, but if you do, there are several interesting variations on this exercise. One variation is to override/redefine the `randomColor` method to generate shades of red or blue or green. Pick a random number for one of the three colors and set the other two colors to 0. Another variation is to construct children that are lighter or darker than their parents, or lighter half the time and darker half the time. See the comment about Question 2 in Chapter 2, Exercise Set 4, for more information about lighter and darker colors.

4. Be careful about where you place the new parameters in `initialize`. They have to be consistent with each constructor.

```
/** Constructs a fish at the specified location in a given environment.
 *  The Fish is assigned a random direction, random color, a constant
 *  probability of breeding, and a constant probability of dying.
 *  (Precondition: parameters are non-null; <code>loc</code> is valid
 *  for <code>env</code>.)
 *  @param env  environment in which fish will live
 *  @param loc  location of the new fish in <code>env</code>
 **/
public Fish(Environment env, Location loc)
{
  initialize(env, loc, env.randomDirection(),
               randomColor(), 1.0/7.0, 1.0/5.0);
}
```

5. Try experimenting with the probabilities of breeding and dying to create relatively stable populations. You may want to specify a color for each fish so that it's easier to see the populations.

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

When the probabilities of breeding and dying are both 0.0, the behavior should be the same as before breeding and dying were added.

Teaching tip: Teachers who have covered Chapter 4 before covering Chapter 3 may wish to ask students whether Pat should have implemented `public` or `protected` accessor and modifier methods for the `probOfBreeding` and `probOfDying` instance variables. (See the related analysis question in Chapter 4.) Another topic for discussion is whether a subclass might want to change these probabilities during a fish's lifetime or whether it is enough to be able to set the probabilities using the constructor created here. For example, if students wanted to create a subclass where a fish's probability of dying increases as it ages (or when it enters certain "polluted" locations in the environment), it would be important to have a modifier method to change the probability. On the other hand, one might not want to create a `public` modifier method that indicates that it is okay for client code to change a fish's probability of dying.

6. One possible solution is shown below.

   Add the following instance variables to the class.

   ```
   private int myTimesBred;
             // the number of times a fish bred in its lifetime
   ```

   Add the following statement to the `initialize` method.

   ```
   myTimesBred = 0;
   ```

   Modify the `breed` method to increment `myTimesBred` each time the fish breeds.

   ```
   myTimesBred++:
   ```

   Modify the `die` method.

   ```
   protected void die()
   {
     Debug.turnOn();  // if debug is not already turned on
     Debug.println(toString() + " about to die.");
     Debug.println(toString() + " bred " + myTimesBred + " times.");
     Debug.restoreState();
     environment().remove(this);
   }
   ```

   Answers to the questions about the maximum and minimum times a fish bred will vary. Have your students observe these values from the `Debug` output.

7. Add a `private` instance variable `myAge`. In the `initialize` method, initialize `myAge` to 0, so each fish will start out at age 0. At the beginning of the `act` method increment `myAge`, add `myAge` to the debugging output in the `die` method and be sure that debugging is turned on. In one sample run, the oldest age at which a fish died was 17, and the youngest was 1. In general, the ages at which fish died were as expected, averaging around age 5 for the sample.

Teaching tip: If you use the "create an environment" option in the GUI, and begin with only a few fish (say three), it's easy to watch the fish movements and to follow the debugging output.

8. Here is one way to solve the problem. Your students may find others. Initialize `probOfBreeding` to 1.0/3.0 and `probOfDying` to 1.0/10.0. At the end of `act`, add the following code.

   ```
   probOfDying += 0.1;
   ```

   At the beginning of `breed`, add the following code.

   ```
   if (myAge < 3)
     return false;
   ```

Note to teachers: As an added challenge problem, you may ask your students to determine the overall maximum and minimum times any fish bred. Since a single run may produce many fish (a single run can produce hundreds, with fish frequently breeding and dying), it is difficult to keep track of visually. This would involve changes to both the `Simulation` and `Fish` classes. One solution is to use class variables in `Fish` (recall that `nextAvailableID` is an example of a class variable) for the extreme values with the minimum initialized to `Integer.MAX_VALUE` and the maximum initialized to 0. Have each fish update the max or min, if necessary, when it dies. `Simulation` would have to ask the `Fish` class for the extreme values at the end of the simulation.

Chapter 4

Specialized Fish

This chapter introduces inheritance and dynamic binding to the case study. `DarterFish` introduces the necessity for defining new constructor(s), the use of `super` in subclass constructors, and the redefinition of "inherited" methods. `SlowFish` introduces an additional instance variable and the use of `super.method` in a redefined method. Although the chapter introduces the use of `super` in constructors and redefined methods, it does not cover these topics thoroughly. `FastFish` (an activity added in Appendix D of this teacher's manual) introduces an additional method and the use of a `protected` accessor method to allow a subclass to modify `private` inherited data.

Design Issues (pages 63–65)

Note to teachers: The concepts of inheritance, polymorphism, and dynamic binding sometimes cause confusion. This note will try to explain them.

In Java, one inherits from another class by using the keyword `extends`. We note that `DarterFish extends Fish`. Because a `DarterFish` "is-a" `Fish`, a `DarterFish` object can be used anywhere that a `Fish` object can be used. In particular, if we declare:

```
  Fish fsh;
```

then both

```
  fsh = new Fish(env, loc);
```

and

```
  fsh = new DarterFish(env, loc);
```

are valid. This idea of being able to store different types in the same variable is one example of polymorphism. In general, polymorphism refers to being able to deal with multiple types. The `"+"` operator in Java (and most programming languages) is polymorphic — it can be used on integers, doubles, and strings.

A primary goal of inheritance is to allow code reuse. By extending a class, one is able to create a derived class whose objects can use all of the `public` or `protected` methods defined in the base class. One need only write the code that makes the behavior of the derived class different than the behavior of the base class. One way to do this is to write new methods that are not in the base class. Another way is to redefine a method that is already in the base class. An example of this is the `move` method defined in `Fish` and then redefined in `DarterFish`.

This gives rise to an interesting question. If `fsh` is defined as above, which `move` method should be used when you call the method `fsh.move()` ? In Java, the answer depends on what is actually referenced by `fsh`. If `fsh` references a `Fish` object, then the `Fish move` should be used. If `fsh` references a `DarterFish` object, then the `DarterFish move` should be used. Because the value in `fsh` might be passed as a parameter or might differ depending on input to the program, it is often not possible to decide whether `fsh` will refer to a `Fish` or a `DarterFish` when the program is compiled. Thus it is often not possible to decide which `move` method to use at compile time. The decision is therefore made at run time.

The case study narrative uses the term "dynamic binding" for this process of deciding which method to use at run time, based on the contents of `fsh`. Unfortunately, there is no single standard terminology for the process of dynamic binding. Various textbooks use the terms "late binding", "polymorphic dispatch", and "dynamic dispatch on type" to describe this process. The various terms are mentioned here only to make it easier to make the connection to whichever term your textbook uses.

The dynamic binding of redefined methods to an object can be difficult to describe and difficult to illustrate. Students may understand what is happening better if they act it out. Refer to the role-playing activities designed by David Levine and Steve Andrianoff, mentioned earlier in this teacher's manual.

Additional information on inheritance, dynamic binding, polymorphism and the `protected` keyword can be found in a variety of resources such as textbooks, Web sites, and the AP Computer Science electronic discussion group (EDG).

Darter Fish (pages 65–67)

If you teach Chapter 4 before Chapter 3, you will need to explain that `generateChild` is a method introduced in Chapter 3 to create new fish when fish breed. Your students will probably understand this easily when they think about different species of fish breeding.

Note to teachers: There is an alternative to the `Fish` class and its subclasses each having their own `generateChild` method — this alternative is called reflection. Using this technique, the code in `Fish breed` would essentially say, "Whatever type of fish I am, create another just the same." The subclasses would not need to redefine anything. This technique is, however, outside the scope of the AP Java subset.

Analysis Question Set 1 (page 68)

1. Pat's first draft doesn't account for jumping over a fish to move two locations forward. The new code in `nextLocation` gets the neighboring location in front of the fish and then the location in front of that, but returns the location two places in front, if it is empty, without checking to see if the location directly in front is also empty.

2. Answers will vary. You could introduce a probability of turning 90 degrees left or right. You could also have the fish turn left or right on each *n*th timestep (e.g., each tenth timestep). If the fish is a breeding fish, it might rotate 90 degrees each time, or every *n*th time, it breeds. Or the fish might change orientation when it reaches a certain age. There are many possibilities.

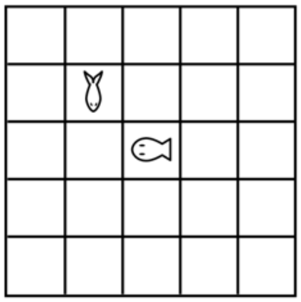
Testing the `DarterFish` Class (pages 68–70)

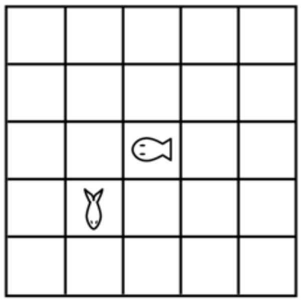
You may want to use the `Fish` class from Chapter 2 for testing so that you are working with a fixed population of fish and do not have breeding and dying. You could maintain two separate projects, one using the original `Fish` class and the other using the breeding and dying `Fish` class, and test the darter fish with the appropriate project. A third alternative, if you think your students are up for it, is to add a new menu item to the graphical user interface to turn breeding/dying fish on and off. See the Help menu in the simulation program (or `MBSHelp.html` in the `Documentation` folder) for details.

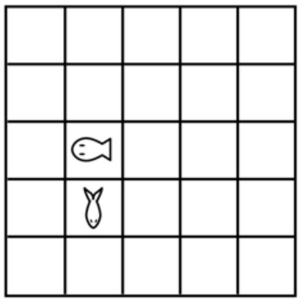
The test results shown were generated using the `darter.dat` configuration file and a seed of 17. You could have students run similar tests using different data files, such as `darterAndNormalFish.dat`, or with a different seed. If you are using only `DarterFish`, you may want to use random colors for this test so that it is easier to distinguish individual fish.

Exercise Set 1 (page 71)

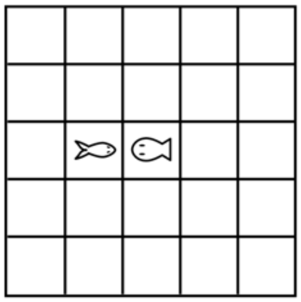
1. Here are some sample illustrations. The label under each picture sequence describes the validity of the darter fish move.

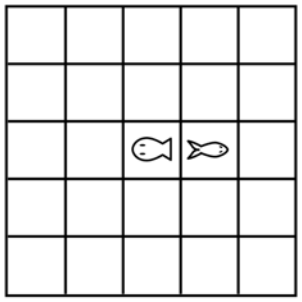

Initial Position

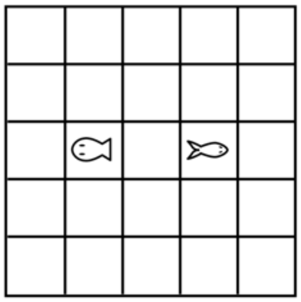

Darter Moves First


Fish Moves Second

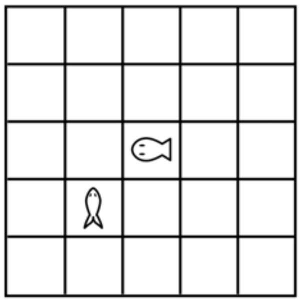
VALID MOVE

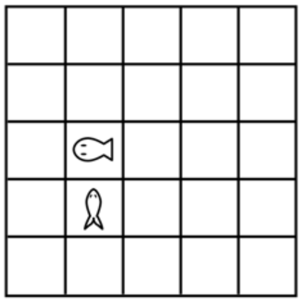

Initial Position

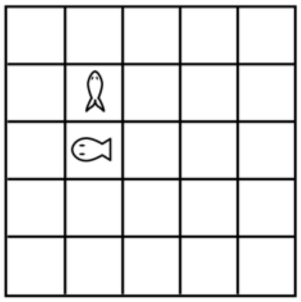

Darter Moves First


Fish Moves Second

INVALID MOVE


Initial Position


Fish Moves First


Darter Moves Second

INVALID MOVE

2. If you want to distinguish more easily between `DarterFish` and `Fish`, open the `MBSGUI.java` file and read the comments carefully (or the section on customizing the graphical user interface in `MBSHelp.html` in the `Documentation` folder). Follow the directions for uncommenting the code for Chapter 4, darter fish and slow fish. It's much easier to see how the behavior differs.

3. As in previous questions of this type, students should save the configuration file and analyze and discuss the results. `Fish` should move in the same way, although the results are likely to be different due to randomization. `DarterFish` will not move in the same way as `Fish` because their rules for moving are different.

4. If you cover Chapter 4 before Chapter 3, you probably will not have added any new constructors, but if you taught Chapter 3 first, your students may have a constructor that includes parameters for the probability of breeding and probability of dying for each individual fish. `DarterFish` could also include such a constructor. If you do not want your students to add the complexity of breeding and dying at this time, but you do want them to add the constructor, set the probabilities to 0.0 for the time being.

5. Students are being asked to redefine the `toString` method in the `DarterFish` class. One simple example is given below. Encourage your students to find other ways to make the darter fish information stand out, especially when there is more than one species in the environment.

```
/** Returns a string representing key information about this fish.
 * @return  a string indicating the fish's ID and location
 **/
public String toString()
{
  return "DarterFish " + super.toString();
          // you could use "Darter" or even "D"
}
```

Note to teachers: An alternate way to get the name `DarterFish` printed in output would be to add it to every print or debug statement that calls `toString`. In that case the call to `toString` would use the inherited `toString` from the superclass. However, finding and modifying all these statements is more tedious and error-prone than redefining `toString`.

Note to teachers: If your students have studied the `Object getClass` and `Class getName` methods, you could have them modify `Fish toString` to include the class name, together with the fish's ID and location instead of redefining `toString` in `DarterFish`. (These methods are not in the AP Java Subset.) If you do this, there is no need to redefine the `toString` method in any of the `Fish` subclasses (see Exercise 3 in Exercise Set 2 and Exercise 2 in Exercise Set 3 below).

6. The simulation will behave in the same way, but results will be different when you start with different seed numbers.

7. The code below is one example of a `TurningDarter` subclass. The use of the `super` keyword in the constructors can be seen in the `DarterFish` class and was explained on page 67 of the narrative.

You may want your students to include a `toString` method for this class as they did with the `DarterFish` class. They will need to create a data file for turning darters. An easy way to do this is copy the `darter.dat` file into a new text file, change all the `DarterFish` to `TurningDarter` and save the file as `turningDarter.dat`. Add your new `turningDarter.java` (or whatever you named the class) file, which is the new subclass file, to your project and run the `MBSGUI` specifying `turningDarter.dat`.

```
// Chapter 4, Exercise Set 1, Question 7  TurningDarter Class
// The turningDarter has all the characteristics of a darter fish except
// that it has a 0.1 chance of turning right or left before it tries to
// move forward.

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

public class TurningDarter extends DarterFish
{
  // Instance Variables: Encapsulated data for EACH TurningDarter fish
  private double probOfTurning;
                 // defines likelihood of turning in each timestep
  private double probOfRight;
                 //defines likelihood of turning right
```

```
// constructors

/** Constructs a turning darter fish at the specified location in a
 *  given environment.   This turning darter is colored magenta.
 *  (Precondition: parameters are non-null)
 *  @param env    environment in which fish will live
 *  @param loc    location of the new fish in <code>env</code>
 **/
public TurningDarter(Environment env, Location loc)
{
  // Construct and initialize the attributes inherited from Fish.
  super(env, loc, env.randomDirection(), Color.magenta);
  probOfTurning = 0.1;
  probOfRight = 0.5;
}

/** Constructs a turning darter fish at the specified location with
 *  the direction in a given environment.
 *  This turning darter is colored magenta.
 *  (Precondition: parameters are non-null)
 *  @param env    environment in which fish will live
 *  @param loc    location of the new fish in <code>env</code>
 *  @param dir    direction the new fish is facing
 **/
public TurningDarter(Environment env, Location loc, Direction dir)
{
  // Construct and initialize the attributes inherited from Fish.
  super(env, loc, dir, Color.magenta);
  probOfTurning = 0.1;
  probOfRight = 0.5;
}

/** Constructs a turning darter fish of the specified color at the
 *  specified location and direction.
 *  (Precondition: parameters are non-null)
 *  @param env    environment in which fish will live
 *  @param loc    location of the new fish in <code>env</code>
 *  @param dir    direction the new fish is facing
 *  @param col    color of the new fish
 **/
public TurningDarter(Environment env, Location loc, Direction dir,
                     Color col)
{
  // Construct and initialize the attributes inherited from Fish.
  super(env, loc, dir, col);
  probOfTurning = 0.1;
  probOfRight = 0.5;
}
```

Note to teachers: Note that `probOfTurning` and `probOfRight` are set to the same values in three different constructors. Separating out these two assignments into an `initialize` method called by each constructor makes it less likely that an error would be introduced if one of these values were modified. This would be a better solution, but the solution on the previous pages is more typical of what students will produce. This comment applies to other solutions in this section as well.

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


  /** Moves this fish in its environment.
   *  A turning darter fish first sees if it should turn left or right
   *  and then darts forward (as specified in nextLocation) if possible,
   *  or reverses direction (without moving) if it cannot move forward.
   **/
  protected void move()
  {
    // If this is the 0.1 chance of turning, turn left or right
    Random randNumGen = RandNumGenerator.getInstance();
    if ( randNumGen.nextDouble() < probOfTurning )
    {
      turnRightOrLeft(direction());
      Debug.println(toString() + " Turned and now facing " +
                     direction());
    }
    super.move();
  }
```

Note to teachers: Calling `super.move()` is much better than reproducing all of the code for moving as a `DarterFish`. Furthermore, it has the advantage that if the `DarterFish move` method is later modified the `TurningDarter` will automatically continue to move like a `DarterFish` after (perhaps) turning.

```
  /** Turns this fish right or left
   **/
  protected void turnRightOrLeft(Direction dir)
  {
    Random randNumGen = RandNumGenerator.getInstance();
    if (randNumGen.nextDouble() < probOfRight)
    {
      changeDirection(dir.toRight());
    }
    else
    {
      changeDirection(dir.toLeft());
    }
  }
}
```

Slow Fish (page 72)

Implementation of the `SlowFish` Class (page 72)

Teachers who have covered Chapter 3: Your students should notice the similarity between the first few lines of `SlowFish nextLocation` and the first few lines of `Fish breed`.

Analysis Question Set 2 (page 74)

1. Advantages

   - The `move` method deals with changing both location and direction. Redefining `nextLocation` rather than `move` clarifies that the only difference in behavior is how (or, actually, how often) a slow fish changes location.
   - `nextLocation` already has the responsibility to return the old location if the fish cannot move, so the test for whether a slow fish should move or not easily fits into this design.
   - There is no need to reproduce and modify the code in `move`.

Disadvantages

- `nextLocation` is always called. If the test had been in `move`, the code would bypass the call to `nextLocation` whenever a slow fish did not move.
- If slow fish were asked to randomly change direction when they do not move beyond their own cell this design would not work well, because in that case the way slow fish choose their next location and the way they choose their next direction would both be different. Thus, either `nextLocation` would have to do something that is an inappropriate side effect of its main purpose (changing a direction has nothing to do with computing a next location) or the design would have to be changed.

2. The subclass would not have a way to access the value of `probOfMoving`. Students can test this for themselves using `TurningDarter` (or `TurningSlowFish` if you assign Question 6 on page 77). The lack of accessor methods for the `probOfBreeding` and `probOfDying` instance variables will be a problem if you want any of the new species of fish to be able to breed or die in a different way that requires using these values.

3. `protected`. These accessor methods should be used only by subclasses of the superclass.

Testing the `SlowFish` Class (pages 74–75)

These test results were generated using the `slowFish.dat` configuration file and a seed of 17.

Analysis Question Set 3 (page 75)

1. Advantages

   - If `BreedingAndDyingFish` is a subclass of `Fish`, when you add other classes, you could have them extend either `Fish` or `BreedingAndDyingFish`. It would be easy to change the class being extended so you could test your new class as having either a static or a dynamic population.

   Disadvantages

   - You would need to decide whether `DarterFish` is a subclass of `Fish` or of `BreedingAndDyingFish` or else create two subclasses, one for each. There is this difficulty when two quite different types of behavior are added or modified using inheritance.

2. `DarterFish` are deterministic while `Fish` and `SlowFish` are probabilistic.

Exercise Set 2 (pages 76–77)

Note to teachers: You may wish to have students go through the optional `FastFish` section in Appendix D (or implement `FastFish` as an assignment) before doing the last three exercises in this Exercise Set. If you want to assign `FastFish`, just give your students the Problem Specification section.

1. Constructors with probabilities of breeding and dying should be added to slow fish. Your students may have other suggestions if they have added other constructors.

2. Each species' behavior is consistent with the movement rules we expect.

3. One possible solution is given below.

```
/** Returns a string representing key information about this slow fish.
 *  @return a string indicating the fish's ID and location
 **/
public String toString()
{
  return  "SlowFish " + super.toString();
}
```

4. The initial configuration is the same. Runs are consistent when the same seed is used and differ when different seeds are used.

5. This exercise was done using the data from the narrative. In the 20 timesteps, 14.5 percent of the time the slow fish attempted to breed (not always successfully), while 18.4 percent of the time the slow fish died. Expected results were 14.3 percent and 20 percent respectively. Twelve fish did not breed because there were no empty neighboring locations, resulting in 459 calls to the `move` method when fish did not breed. Every call to `move` resulted in a call to the `nextLocation` method in `SlowFish`. Of these, only 20.5 percent attempted to move beyond their own cell; 79.5 percent moved too slowly. Expected results were 20 percent and 80 percent. These were much closer to the expected performance than the figures after 10 moves. Your students' results may vary.

6. The question is not completely specified. It does not tell you how, in the case that a slow fish doesn't move outside its cell, you should determine the probability that it will turn in place and, if it turns, the probability that it will turn to the left or to the right.

One example of a solution is given below. This `move` method should be added to `SlowFish`.

You will need to add two instance variables, for example, `probOfTurning` and `probOfTurningRight` to the constructor. The changes will be similar to those made in the instance variables and constructors for turning darter fish.

```
protected void move()
{
  Location current = location();

  super.move();

  if (current.equals(location()))   // Have we moved?
  {
    Random randNumGen = RandNumGenerator.getInstance();

    // If this is the 0.1 chance of turning, turn left or right
    if (randNumGen.nextDouble() < probOfTurning)
    {
      turnRightOrLeft(direction());
      Debug.println(" Turned and now facing " + direction());
      return location();
    }
  }
}

/** Turns this fish right or left
 **/
protected void turnRightOrLeft(Direction dir)
{
  Random randNumGen = RandNumGenerator.getInstance();
  if (randNumGen.nextDouble() < probOfRight)
  {
    changeDirection(dir.toRight());
  }
  else
  {
    changeDirection(dir.toLeft());
  }
}
```

You might also have your students create a `TurningSlowFish` class and a data file for normal, slow, and turning slow fish. In having your students build another subclass, you preserve the original class and provide practice for your students in determining what should be included in the subclass and what should be left out. The `move` method given above can be added to the new class and will work just as if it were added to `SlowFish`. Code for the entire class is given on pages 52-55.

It is tempting to do the turning in `nextLocation` rather than in `move`. This is not a good design, because it makes `nextLocation` have a side effect (turning the fish) that is not related to its primary purpose (choosing and returning a next location). Deciding what is good and bad design is a matter of taste and experience, but most computer scientists would consider it bad design to have the `nextLocation` method perform the turn.

```
// Chapter 4, Exercise Set 2, Question 6
// A turning slow fish may turn right or left even if it doesn't move
// outside its own cell.
import java.awt.Color;
import java.util.ArrayList;
import java.util.Random;

/**
 *  Marine Biology Simulation:
 *  The TurningSlowFish class represents a fish in the MBS 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 behavior is the same as for objects of the Fish
 *  class.  If it doesn't move beyond its cell, the fish may still
 *  turn left or right.
 *
 *  TurningSlowFish objects inherit instance variables and much of
 *  their behavior from the SlowFish class.
 *
 **/

public class TurningSlowFish extends SlowFish
{
  // Instance Variables: Encapsulated data for EACH slow fish
  private double probOfMoving;
          // defines likelihood in each timestep
  private double probOfTurning;
          // defines likelihood of turning in place
  private double probOfRight;
          // defines likelihood of turning right

  // constructors

  /** Constructs a turning slow fish at the specified location in a
   *  given environment.   This slow fish is colored green.
   *  (Precondition: parameters are non-null; <code>loc</code> is
   *  valid for <code>env</code>.)
   *  @param env    environment in which fish will live
   *  @param loc    location of the new fish in <code>env</code>
   **/
```

```
public TurningSlowFish(Environment env, Location loc)
{
  // Construct and initialize the attributes inherited from SlowFish.
  super(env, loc, env.randomDirection(), Color.green);

  // Define the likelihood that a turning slow fish will move in any
  // given timestep.  This is the same value for all slow fish.
  probOfMoving = 1.0/5.0;  // 1 in 5 chance in each timestep
  probOfTurning = 0.1;     // 1 in 10 chance of turning in place
  probOfRight = 0.5;       // chance of turning right is one-half
}

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

  // Define the likelihood that a turning slow fish will move in any
  // given timestep.  This is the same value for all slow fish.
  probOfMoving = 1.0/5.0;  // 1 in 5 chance in each timestep
  probOfTurning = 0.1;     // 1 in 10 chance of turning in place
  probOfRight = 0.5;       // chance of turning right is one-half
}

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

```
    // Define the likelihood that a turning slow fish will move in any
    // given timestep.  This is the same value for all slow fish.
    probOfMoving = 1.0/5.0;  // 1 in 5 chance in each timestep
    probOfTurning = 0.1;     // 1 in 10 chance of turning in place
    probOfRight = 0.5;       // chance of turning right is one-half
  }

  // redefined methods

  /** Returns a string representing key information about this
   *  turning slow fish.
   *  @return a string indicating the fish's ID and location
   **/
  public String toString()
  {
    return  "TurningSlowFish " + super.toString();
  }

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

  protected void move()
  {
    Location currentLoc = location();

    super.move();

    if (currentLoc.equals(location()))  // Have we not moved?
    {
      Random randNumGen = RandNumGenerator.getInstance();

      // If this is the probOfTurning chance of turning,
      //  turn left or right.
      if (randNumGen.nextDouble() < probOfTurning)
      {
        turnRightOrLeft(direction());
        Debug.println(" Turned and now facing " + direction());
        return location();
      }
    }
  }
  /** Turns this fish right or left
   **/
  protected void turnRightOrLeft(Direction dir)
  {
    Random randNumGen = RandNumGenerator.getInstance();
    if (randNumGen.nextDouble() < probOfRight)
    {
      changeDirection(dir.toRight());
    }
    else
    {
      changeDirection(dir.toLeft());
    }
  }
}
```

7. Copy the code from the dynamic `Fish` class that includes breeding and dying into a new file and save it using a name for the breeding and dying subclass, such as `BDFish.java`. Be sure to include `extends Fish` in the class declaration and be careful to change references to `Fish` to read `BDFish`. Compare this code to the original `Fish` class. Change the constructors in `BDFish` in the same way the constructors were changed for `SlowFish` or `DarterFish`. Use as much of the code from the superclass `Fish` as possible. Remove from your new `BDFish` class all code that duplicates code that is already in `Fish`. Keep all code that was modified or added for Chapter 3 (instance variables, modified `act` and `move`, `breed`, `die`, `generateChild`, and `toString`). Make sure `generateChild` constructs a new `BDFish`.

   Very important: Remember to run this using the `Fish` class from Chapter 2, not the dynamic version of `Fish`.

   The following code is an example of the `BDFish` subclass. You will also need to create a data file that contains `BDFish`. One way to do this is to copy `fish.dat` to a new text file and then add `"BD"` in front of each `"Fish"` in the file. Save the file as `BDFish.dat` or some other appropriate name.

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

```
/**
 *  Marine Biology Simulation:
 *  The BDFish class represents a breeding and dying fish subclass in
 *  the Marine Biology Simulation.
 *  Each BDfish has a unique ID, which remains constant throughout its
 *  life.  A BDfish also maintains information about its location and
 *  direction in the environment.
 **/

public class BDFish extends Fish
{
  private double probOfBreeding;  // defines likelihood in each timestep
  private double probOfDying;  // defines likelihood in each timestep

  // constructors and related helper methods

  /** Constructs a BDFish at specified location in a given environment.
   *  The BDFish is assigned a random direction and random color.
   *  (Precondition: parameters are non-null.)
   *  @param env    environment in which fish will live
   *  @param loc    location of the new fish in <code>env</code>
   **/
  public BDFish(Environment env, Location loc)
  {
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, env.randomDirection());
    // Define the likelihood that a BDFish will breed or die
    // myAge initialization provided, to be used if desired
    probOfBreeding = 1.0/7.0;
    probOfDying = 1.0/5.0;
  }

  /** Constructs a BDFish at the specified location and direction in a
   *  given environment.  The BDFish is assigned a random color.
   *  (Precondition: parameters are non-null.)
   *  @param env    environment in which fish will live
   *  @param loc    location of the new fish in <code>env</code>
   *  @param dir    direction the new fish is facing
   **/
  public BDFish(Environment env, Location loc, Direction dir)
  {
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, dir);
    // Define the likelihood that a BDFish will breed or die
    // myAge initialization provided, to be used if desired
    probOfBreeding = 1.0/7.0;
    probOfDying = 1.0/5.0;
  }
```

```
/** Constructs a BDFish of the specified color at the specified
 *  location and direction.
 *  (Precondition: parameters are non-null.)
 *  @param env    environment in which fish will live
 *  @param loc    location of the new fish in <code>env</code>
 *  @param dir    direction the new fish is facing
 *  @param col    color of the new fish
 **/
public BDFish(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 BDFish will breed or die
  // myAge initialization provided, to be used if desired
  probOfBreeding = 1.0/7.0;
  probOfDying = 1.0/5.0;
}

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


// modifier method

//  (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();
  // uncomment the following line of code to increase probability
  // of dying as fish ages
  //probOfDying += 0.1;
}
```

```
  // helper methods

  /** Attempts to breed into neighboring locations.
   *  @return    <code>true</code> if fish successfully breeds;
   *             <code>false</code> 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(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;
  }

  /** 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.
    BDFish child = new BDFish(environment(), loc,
                              environment().randomDirection(), color());
    Debug.println("  Newly created: " + child.toString());
  }
  /** Removes this fish from the environment.
   **/
  protected void die()
  {
    Debug.println(toString() + " about to die ");
    environment().remove(this);
  }
}
```

8. The following code is an example of a `CircleFish` class.

Note to teachers: This exercise is interesting but it isn't completely specified. It isn't clear whether a circle fish moves forward and then to the right (and therefore both locations must be empty in order for the circle fish to make the diagonal move) or if only the location on the diagonal, forward and to the right, must be empty. The author's intention was the second interpretation. A fish can move diagonally, provided the location on the diagonal is empty, even if the location directly in front of the fish is not empty. Your students may miss this assumption because it is not stated in the question, so they may write an additional (unnecessary) check for an empty location in front and as well as on the diagonal when only the diagonal needs to be checked. You may decide to allow your students to interpret the question either way and, if they ask you to clarify the problem, initiate a discussion about problem specifications. The code that follows presents one way to solve the problem as the author intended.

```
/** Chapter 4, Exercise Set 2, Question 8
 *  Create a CircleFish class of fish that, within each step,
 *  continually attempts to move forward and right so that it makes
 *  a diagonal move and goes in a circular direction.
 *  If the fish cannot make the diagonal move, it stays in its current
 *  location but still turns 90 degrees to the right.
 **/
import java.awt.Color;

public class CircleFish extends Fish
{
  // constructors

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

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

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

// redefined methods

/** Returns a string representing information about this Circlefish.
 *  @return a string indicating the fish's ID and location
 **/
public String toString()
{
  return  "CircleFish " + super.toString();
}
```

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

/** Moves this fish in its environment.
 *  A CircleFish moves in a circular pattern (as specified
 *  in nextLocation) if possible,
 *  or remains in its current location and turns right.
 **/
protected void move()
{
  Location oldLoc = location();
  super.move();
  if (oldLoc.equals(location()))
  {
    // Otherwise, turn right.
    ChangeDirection(direction().toRight());
  }
}

/** Finds this fish's next location.
 *  A CircleFish moves to the right diagonally if it can, otherwise
 *  it remains in its current location and turns right.
 *  A CircleFish can only move to empty locations diagonally forward
 *  and to the right.
 *  If the CircleFish cannot move diagonally forward,
 *   <code>nextLocation</code>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 inFrontAndRight = env.getNeighbor(oneInFront,
                                   direction().toRight());
    // Test for empty location diagonally up and right
    if ( env.isEmpty(inFrontAndRight) )
    {
      Debug.println("  Location diagonally up and  right is empty? " +
                         env.isEmpty(inFrontAndRight));
      return inFrontAndRight;
    }
    // Only get here if there isn't a valid location to move to
    Debug.println("  Circle is blocked.");
    return location();
  }
}
```

9. One way to solve this problem is shown on the next few pages. A `private` boolean instance variable, `justTurned`, is added to `CircleFish` and used to toggle between moving forward or diagonally. Students can note that a subclass may contain additional instance variables. Modifications are shown in boldface.

Note that we use the variable `justTurned` in `nextLocation` to decide what sort of move to make, but that we update its value in `move`. It is possible to update `justTurned` in `nextLocation` instead, but this is not as good a design decision. The way that we have implemented this, `nextLocation` is a pure function, with no side effects. It is responsible for computing the next location and returning it, nothing more. If one were to call it twice in a row without calling `move` in between, one would get the same answer both times. However, if `justTurned` is updated in `nextLocation` instead of in `move`, then this would no longer be true.

The `move` method is responsible for updating the location and the direction of the `CircleFish`. It is therefore the natural place to update the `justTurned` variable, which records whether the `CircleFish` has moved forward or turned most recently.

```
// CircleFish.java

import java.awt.Color;

/**
 *  CircleFish objects inherit instance variables and much of their
 *  behavior from the Fish class.
 *
 *  @author Chris Nevison
 *  @author Alyce Brady
 *  @version 25 March 2003
 **/

public class CircleFish extends Fish
{
  private boolean justTurned;

  public CircleFish(Environment env, Location loc)
  {
    // Construct and initialize the aspects inherited from Fish.
    super(env, loc);
    justTurned = true;
  }

  /** Constructs a CircleFish of that is orange at the specified
   *  location and direction.
   *  @param env    environment in which fish will live
   *  @param loc    location of the new fish in <code>env</code>
   *  @param dir    direction the new fish is facing
   **/
  public CircleFish(Environment env, Location loc, Direction dir)
  {
    // Construct and initialize the aspects inherited from Fish.
    super(env, loc, dir, Color.orange);
    justTurned = true;
  }

  /** Constructs a CircleFish of the specified color at the specified
   *  location and direction.
   *  @param env    environment in which fish will live
   *  @param loc    location of the new fish in <code>env</code>
   *  @param dir    direction the new fish is facing
   *  @param col    color of the new fish
   **/
  public CircleFish(Environment env, Location loc, Direction dir,
                    Color col)
  {
    // Construct and initialize the aspects inherited from Fish.
    super(env, loc, dir, col);
    justTurned = true;
  }

  // redefined methods

  /** Returns a string representing information about this Circlefish.
   *  @return a string indicating the fish's ID and location
   **/
  public String toString()
  {
    return  "CircleFish " + super.toString();
  }

  /** Creates a new CircleFish, with the same color as this CircleFish
   *  @param loc    location of the new fish
   *  @return    the new fish
   **/
  protected void generateChild(Location loc)
  {
    new CircleFish(environment(), loc,
               environment().randomDirection(), color());
  }

  /** Moves this fish in its environment.
   *  A CircleFish alternates moves one location forward, and one
   *  location on the diagonal to its right, turning to the right
   *   in the latter case.
   *  If it does change location, the CircleFish turn right,
   *  thereby avoiding being trapped at a side or corner.
   **/
  protected void move()
  {
    // Find a location to move to.
    Debug.print(toString() + " attempting to move.  ");

    Location oldLoc = location();
    Location nextLoc = nextLocation();

    if (nextLoc.equals(oldLoc))
    {
      changeDirection(direction().toRight());
      justTurned = true;
      Debug.println("  Now facing " + direction());
    }
    else
    {
      // Move.
      changeLocation(nextLoc);
      if (!justTurned)
      {
        changeDirection(direction().toRight());
        justTurned = true;
        Debug.println("  Now facing " + direction());
      }
      else
      {
        justTurned = false;
      }
    }
  }

  /** Finds this fish's next location.
   *  A CircleFish alternates moves one location forward, and one
   *  location on the diagonal to its right.
   *  If the CircleFish cannot move forward, then
   *  <code>nextLocation</code> returns the fish's current location.
   *  If the fish moves forward, the boolean justTurned is set to false,
   *  otherwise, it is set to true.
   *  @return    the next location for this fish
   **/
  protected Location nextLocation()
  {
    Environment env = environment();
    Location forward = env.getNeighbor(location(), direction());
    Location rightDiag =
                env.getNeighbor(forward, direction().toRight());
    if (justTurned && env.isEmpty(forward))
    {
      return forward;
    }
    else if (!justTurned && env.isEmpty(rightDiag))
    {
      return rightDiag;
    }
    else
    {
      return location();
    }
  }
}
```

Chapter 5

Environment Implementations

This chapter covers interfaces, two-dimensional arrays and the `ArrayList` class very thoroughly. Some `ArrayList` methods were already mentioned at least briefly in Chapter 2: `get`, `size`, `add(Object)` and `remove(Object)`. The last, `remove(Object)`, is not in the AP Java subset but is used in the case study. This chapter covers these methods and the `remove(int)` and `set` methods.

Analysis Question Set 1 (page 80)

1. Answers will vary. Invite students to contribute their ideas and encourage the class to discuss the merits offered. It is important to note the following:

    - A large and sparsely populated two-dimensional array is wasteful of memory. An environment with irregular boundaries is difficult to represent as a two-dimensional array.
    - If data is stored in an unordered list, less memory may be used. Inserting a new object is trivial, but checking for neighboring objects and displaying the environment will be more complex.
    - If the list is sorted, insertions will be more complex but the display will probably be more straightforward.

    In general, think of the merits in terms of memory used and time required to insert an object, remove an object, search for an object, and display the environment.

2. The list representation would allow you to keep track of more than one object in a location in the environment. You could use a two-dimensional array where each element contains a set of `Locatable` objects found at that location.

Analysis Question Set 2 (page 83)

1. Each class must implement the `location` method from the `Locatable` interface because the parameter to the `add` method is a `Locatable` object. The classes must also explicitly state `"implements Locatable"` in the class declaration or be a subclass of a class that does.

2. The `Comparable` interface contains only one method, `compareTo`. We want to compare two locations when we are checking for consistency between a fish's location and the environment location. In general, it is useful to implement the `Comparable` interface for items that can be ordered, so you can make less-than or greater-than comparisons.

3. It would make it easy to change the display from GUI to text or to a different form of GUI or text, or even to a class that stores information in a file.

4. Answers may vary. The current implementation of `Simulation` casts the objects in the `allObjects` array to `Fish`. If you wanted to use the `Simulation` class for a marching band, you would either need to edit this cast or have `BandMember` inherit from `Fish`. If `Simulation` were to cast the objects in the `allObjects` array to `Actable`, it could be used with different classes as long as they implement the `Actable` interface. A disadvantage is the added complexity of managing another interface.

5. Adding `act` to the `Locatable` interface would mean that `act` could be implemented in different ways for each class that implements `Locatable`. This adds flexibility to `Locatable` objects. On the other hand, every `Locatable` object would be required to implement an `act` method, whether `act` is appropriate for that object or not. In terms of design, `act` is not necessarily related to something that is `Locatable`.

Analysis Question Set 3 (page 87)

1. The checking is different. The `objectAt` method would return `null` if the location is not valid, so then `isEmpty` would return `true`. But the specification for `isEmpty` says that it should return `false` if the location is not valid.

Analysis Question Set 4 (page 88)

1. An array would not be a good representation because it might not be memory efficient. An array cannot be easily resized. Even when it is sparsely populated, it must be allocated with enough capacity to hold the maximum number of objects for the case when the environment is completely full. A better choice is an `ArrayList` because it resizes as needed.

2. The methods do not guarantee the order in which they return the environment objects. In fact, `toString` specifically says "(not necessarily in any particular order)". For example, by switching the outer and inner loops in `allObjects`, you could change the order in which objects are returned to column-major order.

Analysis Question Set 5 (page 90)

1. In the first line of `add`, the call to `obj.location()` would cause a `NullPointerException` exception to be thrown. Program execution would terminate immediately.

2. Method `isEmpty` only reports whether there is any object at the location. For `remove`, though, we need to verify that there is an object at the location and that it is the one we are trying to remove.

Note to teachers: Consider assigning Question 3 as a paper-and-pencil exercise. This could provide a good practice exercise for the free-response part of the AP Exam. Alternatively, you could have students actually code and test this exercise.

3. AB students should be able to express the efficiencies using Big-Oh notation. Both methods look at only a single cell in the internal two-dimensional array, so both are O(1).

```
public void removeFrom(Location loc)
{
  if ( !isEmpty(loc) )
    theGrid[loc.row()][loc.col()] = null;
}
```

Note to teachers: Another suggested topic for discussion with your class is using `null` references, as `BoundedEnv` does, versus using objects of an `EmptyCell` class to represent empty cells in the environment.

Analysis Question Set 6 (page 95)

1. No, it would not be appropriate. Conceptually, it is not true to say that an unbounded environment “is-a” bounded environment.

2. In a very large environment, it is possible, although highly unlikely, that there might actually be `Integer.MAX_VALUE` rows or columns. Using a negative value will not cause this ambiguity.

Searching through the environment (pages 95–97)

If you have covered the `ArrayList toArray` method, you could have students re-implement `allObjects` as shown below. The `ArrayList toArray` method is not required by the AP Java subset.

```
public Locatable[] allObjects()
{
  Locatable[] theList = new Locatable[objectList.size()];
  return (Locatable[]) objectList.toArray(theList);
}
```

The local variable is necessary so that we can pass an array of the right type to the `toArray` method.

Analysis Question Set 7 (page 97)

1. Any location with the same row and column should be considered a match, regardless of whether it is exactly the same `Location` object. See the description of the `move` method in Chapter 2.

2. The simulation asks the fish to act in the order that they were returned in the `allObjects` array. A differing processing order means that fish will use different random numbers to decide where to breed, move, and die. Also, the neighbors of a fish depend on the processing order, since a fish may become a neighbor (or leave the neighborhood) of another fish that hasn't yet acted.

Analysis Question Set 8 (page 101)

1. The `indexOf` method is not intended for use by clients of the class. It is a helper method for `objectAt`. The use of `protected` allows `indexOf` to be used in subclasses of `UnboundedEnv` as well as in `UnboundedEnv` itself.

2. Pat's first draft of `remove` calls `objectAt`, which loops through the `ArrayList` to check the precondition that the object is in the environment. Then it calls `objectList.remove(obj)`, so the list has to be traversed a second time to find the object again and remove it.

3. The first draft does 3*n comparisons (two calls to `objectAt` and one to `isEmpty`), whereas the final version does 2*n comparisons (two comparisons in each iteration of the single loop). They are, however, both O(n) in the long run.

4. 1st draft of `remove`: O(n)
   2nd draft of `remove`: O(n)
   1st draft of `recordMove`: O(n)
   2nd draft of `recordMove`: O(n)

If your class has covered the `indexOf` method in `ArrayList`, you may wish to ask students to consider possible alternative method implementations. For example, would it make sense to implement the `objectAt` method using `ArrayList indexOf` ? (No, because `ArrayList indexOf` needs the entire object to match against, whereas `objectAt` knows only the location.) Would it make sense to implement the `remove` method using `ArrayList indexOf` ? (Yes, see the code sample below.) Would it make sense to implement the `recordMove` method using `ArrayList indexOf` ? (No, it would have the same problems as Pat's first draft.)

```
public void remove(Locatable obj)
{
  // Find the index of the object to remove.
  int index = objectList.indexOf(obj);

  if ( index == -1 )
    throw new IllegalArgumentException("cannot remove "
                                       + obj + "; not there ");

  // Remove the object.
  objectList.remove(index);
}
```

The code above shows what the `remove` method would look like.

Analysis Question Set 9 (page 103)

1. When a fish moves out of the display, its direction is changed to the direction it just moved, which is away from the display. Fish do not move backward, so on the second timestep a fish could not move back into the display area.

2. The fish will probably be processed in a different order because of the behavior of the two different `allObjects` methods. Refer back to Analysis Question Set 7, Number 2. Since fish on the boundaries of the viewable area can move outside of this area, fish in the middle will have more room to move.

3. This is a very open-ended question. A simple approach is to create a driver that puts some kind of `Locatable` objects into the environment and tests the methods of the `UnboundedEnv` class.

4. This question foreshadows Exercise Set 1, Question 3, on page 104. The two-dimensional array could have many empty locations, so it's not a very efficient representation. An `ArrayList` would only store actual `Fish` objects. (Recall that the location is included as part of each fish.) You would want to preserve code that keeps track of boundaries (`isValid` and calls to `isValid`) and otherwise use code from the `ArrayList` implementation.

5. For the bounded environment, `objectAt` is O(1), because no searching is required. For the unbounded environment, using an `ArrayList`, the performance is O(n), where n is the number of fish.

Note to teachers: If you are covering this chapter with A-level students, they do not need to describe performance in Big-Oh terms but should be able to describe the difference between checking for an object at a specific location in a two-dimensional array and searching for an object in an unordered list.

6. In the bounded representation, `allObjects` performance does not depend on the number of fish but is proportional to O(`numRows() * numCols()`), while in the unbounded representation it is O(n), where n is the number of fish.

7. The answer is the same as for Question 5 because `isInEnv` calls `objectAt`.

Teaching tip: Here is an excellent opportunity to discuss trade-offs when choosing how an object will be represented. Will some methods be called more frequently than others? Will breeding and dying make a difference?

8. Again, this question is open-ended and a source of considerable classroom discussion. If you cover the case study before you get to topics such as binary search trees, hash tables, and maps, you may wish to bring these questions up later in the course at a time when they are appropriate. (See the code in Exercise Set 1.)

This table gives the average case Big-Oh running times for a sorted list, BST, and a map that is implemented as a hash table using the Java `HashMap` class.

	`objectAt`	`allObjects`	`add`	`remove`
Sorted List	O(log n)	O(n)	O(n)	O(n)
BST	O(log n)	O(n)	O(log n)	O(log n)
`HashMap`	O(1)	O(n)	O(1)	O(1)

Exercise Set 1 (page 104)

1. The test cases work as expected. That is, valid data files run without error and fish move as expected to valid locations. Invalid data files continue to generate an error. The major difference is that fish can move in and out of the observed environment. Without encountering a barrier or another fish, `DarterFish` can move away forever.

2. Fish lined up in row 0 and column 0 no longer are constrained by a boundary. They can move to negative positions.

3. The following are examples of implementations.

`VLBoundedEnv`

The code that follows implements a very large bounded environment as an extension of `UnboundedEnv`.

```
public class VLBoundedEnv extends UnboundedEnv
{

  private int numberOfRows;
  private int numberOfCols;

  public VLBoundedEnv(int rows, int cols)
  {
    // Construct and initialize the inherited attributes
    super();
    numberOfRows = rows;
    numberOfCols = cols;
  }

  public int numRows()
  {
    return numberOfRows;
  }

  public int numCols()
  {
    return numberOfCols;
  }
  public boolean isValid(Location loc)
  {
    if ( loc == null )
      return false;

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

The `ListEnv` class

The `ListEnv` class, an abstract class, will include the implementation of all the methods from the `UnboundedEnv` class except `numRows`, `numCols`, and `isValid`, which will be made `abstract`. These three methods will be implemented in the subclasses `UnboundedEnv` and `VLBoundedEnv`. The only change needed to the `VLBoundedEnv` class shown above is in the `extends` clause so that the class `extends ListEnv` instead of `UnboundedEnv`.

Note to teachers: You can copy `UnboundedEnv` and save it as `ListEnv`, being careful to change each `UnboundedEnv` in the code to `ListEnv`, adding the `abstract` keyword and removing the method bodies for `numRows`, `numCols`, and `isValid`.

```
import java.util.ArrayList;

public abstract class ListEnv extends SquareEnvironment
{
  private ArrayList objectList;

  public ListEnv()
  {
    objectList = new ArrayList();
  }

  public abstract int numRows();

  public abstract int numCols();

  public abstract boolean isValid(Location loc);

  // The rest of the methods copied from UnboundedEnv

}
```

`UnboundedEnv` as a subclass of `ListEnv`

```
public class UnboundedEnv extends ListEnv
{

  public UnboundedEnv()
  {
     super();
  }


  public int numRows()
  {
     return -1;
  }

  public int numCols()
  {
     return -1;
  }

  public boolean isValid(Location loc)
  {
     return loc != null;
  }

}
```

`SMBoundedEnv` **class**

Note to teachers: In this code, you will see similarities with the `BoundedEnv` and `VLBoundedEnv` classes. Consider how this code would look if you started with `BoundedEnv`. The code that follows shows modifications in boldface. Many of the comments have been removed to save space.

```
import java.util.LinkedList;
import java.util.Iterator:

public class SMBoundedEnv extends SquareEnvironment
{
  private LinkedList[] rows;
  private int numberOfCols;

  public SMBoundedEnv(int nRows, int nCols)
  {
    super();

    numberOfCols = nCols;
    rows = new LinkedList[nRows];
    for ( int index = 0; index < rows.length; index++ )
      rows[index] = new LinkedList();
  }


  public int numRows()
  {
    return rows.length;
  }

  public int numCols()
  {
    return numberOfCols;
  }

  public int numObjects()
  {
    int objectCount = 0;

    for ( int index = 0; index < rows.length; index++ )
      objectCount += rows[index].size();

    return objectCount;
  }
```

```
public Locatable[] allObjects()
{
  Locatable[] theObjects = new Locatable[numObjects()];
  int tempObjectCount = 0;

  // Step through all the rows.
  for ( int index = 0; index < rows.length; index++ )
  {
    // Step through all the objects in this row.
    Iterator itr = rows[index].iterator();
    while ( itr.hasNext() )
    {
      // Put the next object in the array.
      theObjects[tempObjectCount] = (Locatable)itr.next();
      tempObjectCount++;
    }
  }
  return theObjects;
}

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

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

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

public Locatable objectAt(Location loc)
{
  if ( ! isValid(loc) )
    return null;
  // Step through all the objects in the row for location loc
  Iterator itr = rows[loc.row()].iterator();
  while ( itr.hasNext() )
  {
    // Is this object at the location we're looking for?
    Locatable obj = (Locatable) itr.next();
    if ( obj.location().equals(loc) )
    {
      // Found the object -- return it.
      return obj;
    }
  }
  return null;
}

// modifier methods

public void add(Locatable obj)
{
  Location loc = obj.location();
  if ( ! isEmpty(loc) )
    throw new IllegalArgumentException("Location " + loc +
                                   " is not a valid empty location");
  // Add object to the environment.
  rows[loc.row()].add(obj);
}

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 environment.
  rows[loc.row()].remove(obj);
}
public void recordMove(Locatable obj, Location oldLoc)
{
  // Simplest case: There was no movement.
  Location newLoc = obj.location();
  if ( newLoc.equals(oldLoc) )
    return;

  // Verify that there wasn't already an object at new location.
  Iterator itr = rows[newLoc.row()].iterator();
  while ( itr.hasNext() )
  {
    Locatable current = (Locatable) itr.next();
    if ( current.location().equals(newLoc) && current != obj )
    {
      throw new IllegalArgumentException("There's already " +
                "an object (" + current + ") at location " + newLoc);
    }
  }

  // If object stayed within same row, no change made;
  // otherwise remove object and add it to new row.
  if ( newLoc.row() != oldLoc.row() )
  {
    // Remove object from old row and add it to new row.
    rows[oldLoc.row()].remove(obj);
    add(obj);
  }
}

}
```

4. **`SLUnboundedEnv class`** implementation

The class `SLUnboundedEnv` is an implementation of an unbounded environment that uses a sorted `ArrayList` to store the fish.

The advantage of using a sorted list is that it enables you to use binary search to find an object in the list, taking O(log n) time for n fish, instead of O(n) time. In addition, a binary search can be used to find the location in which to add or remove an element. However, note that the `add` operation is still O(n) overall since in an `ArrayList` the elements above the index where an element is added must all be moved up one position. The `remove` operation is similarly O(n) because of the work required to shuffle down the elements once an object has been removed.

The time for the `recordMove` method is also O(n). This method starts with a linear loop to verify the preconditions. This code is copied from the ordinary unsorted list traversal in `UnboundedEnv`. It doesn't use binary search because the fish that is moving has updated its internal location but it is still in its old position within the sorted list, making the list unsorted. Until that fish is properly moved, methods that depend on binary search such as `isEmpty` and `objectAt` can return erroneous results. If the fish has moved up in the list, a search for the correct index is carried out by moving down the list from the old index, sliding elements up until the correct index is reached, much like the inner loop in insertion sort. If the fish moves down, the same procedure is used, but moving up the list instead of down.

The class `SLUnboundedEnv` is an implementation of an unbounded environment that uses a sorted `ArrayList` to store the fish. It again uses the helper method `indexOf` to find the index of an object in this `ArrayList`. In this case, however, when an object is not found, `indexOf` returns the index, `k`, where that object would be placed if it were added to the list.

The following methods for `SLUnboundedEnv` have identical implementations as for `UnboundedEnv`.

- the constructor
- `numRows`
- `numCols`
- `isValid`
- `numObjects`
- `allObjects` (but the order of objects is row major order for this version)
- `isEmpty`
- `toString`

The method `objectAt` is changed to verify that the desired object is actually at the index returned by `indexOf`.

The `add` and `remove` methods both use `indexOf` to find the index from which to add or remove the object.

The method `recordMove` starts with the same body as the version from the original `UnboundedEnv` but has additional code to move the object to the correct place within the sorted list using a shuffle operation similar to that of insertion sort.

```java
import java.util.ArrayList;

public class SLUnboundedEnv extends SquareEnvironment
{
  private ArrayList objectList;

  public SLUnboundedEnv()
  {
    // Construct and initialize inherited attributes.
    super();

    objectList = new ArrayList();
  }

  // accessor methods

  public int numRows()
  {
    return -1;
  }

  public int numCols()
  {
    return -1;
  }


  public boolean isValid(Location loc)
  {
    return loc != null;
  }


  public int numObjects()
  {
    return objectList.size();
  }


  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;
  }
```

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


public Locatable objectAt(Location loc)
{
  int index = indexOf(loc);
  if ( index >= objectList.size())
    return null;

  Locatable objAtIndex = (Locatable)objectList.get(index);
  if (!objAtIndex.location().equals(loc) )
    return null;

  return objAtIndex;
}


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

public void add(Locatable obj)
{
  // Check precondition.  Location should be empty.
  Location loc = obj.location();
  if ( ! isEmpty(loc) ) // loc is not empty
  {
    throw new IllegalArgumentException("Location " + loc +
                                " is not a valid empty location");
  }
  else
  {   // add object to list in correct position
    objectList.add(indexOf(loc), obj);
  }
}


public void remove(Locatable obj)
{
  // Find the index of the object to remove.
  int index = indexOf(obj.location());
  if (index >= objectList.size() || objectList.get(index) != obj)
  {
    throw new IllegalArgumentException("Cannot remove " +
                                        obj + "; not there");
  }
  else
  {
    // Remove the object.
    objectList.remove(index);
  }
}


public void recordMove(Locatable obj, Location oldLoc)
{
  // We cannot use binary search to find the existing object
  // (or object at the new location) since the object
  // being moved is in the list, but at the "wrong" place
  // for its current location.  We have to find the object
  // the "hard" way with an ordinary linear search
  int objectsAtOldLoc = 0;
  int objectsAtNewLoc = 0;
  int foundIndex = -1;

  // 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++;
    if (thisObj == obj)
      foundIndex = index;
  }

  // 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);
  }
  int index = foundIndex;
  if ( newLoc.compareTo(oldLoc) < 0 )
  {
    // move to earlier position in list, sliding elements up
    while (index > 0 &&
  newLoc.compareTo(((Locatable)objectList.get(index-1)).location())<0)
    {
      objectList.set(index, objectList.get(index - 1));
      index--;
    }
    objectList.set(index, obj);
  }
  else if ( newLoc.compareTo(oldLoc) > 0 )
  {
    // move to later position in list, sliding elements down
    while (index < objectList.size() - 1 &&
newLoc.compareTo(((Locatable)objectList.get(index+1)).location())>0)
    {
      objectList.set(index, objectList.get(index + 1));
      index++;
    }
    objectList.set(index, obj);
  }
}

// internal helper method

/** Get the index of the object at the specified location or
 *  or the index where an object at that location should be added
 *  to the list.
 *  Uses binary seach, so time is O(log K) for K fish,
 *  Returns the location of the object if found, or where it
 *  should be, if not found
 **/
protected int indexOf(Location loc)
{
  int low = 0;
  int high = objectList.size() - 1;

  while ( low <= high )
  {
    int mid = (low + high) / 2;
    Location midLoc = ((Locatable)objectList.get(mid)).location();
    if ( loc.equals(midLoc) )
    {
      Debug.println("indexof " + loc + " is " + mid);
      return mid;
    }
    else if ( loc.compareTo(midLoc) < 0 )
      high = mid - 1;
    else // loc > midLoc
      low = mid + 1;
  }
  Debug.println("indexof " + loc + " is " + low);
  return low;
}

}
```

5. **`BSTUnboundedEnv class`** implementation

This problem asks that the unbounded environment be implemented using a binary search tree (BST). This could be done directly as shown in this solution or indirectly by using the `TreeMap` implementation of a map. (The Java library class `TreeMap` is implemented with a balanced binary search tree.) Exercise 6 shows a solution using the `HashMap` implementation of a map. The code for a `TreeMap` would be virtually identical to the code for Exercise 6 that uses a `HashMap`. Here we directly implement a BST within the environment.

We use the implementation class `TreeNode` to implement the nodes of the BST and we store the `Locatable` objects directly in the nodes. We need a `private` data field, `root`, for the root node of the tree. We also need a `private` integer data field, `numObj`, to store the number of objects, since we are not using a structure with a size method.

The following methods for `BSTUnboundedEnv` have identical implementations as for `UnboundedEnv`.

- the constructor
- `numRows`
- `numCols`
- `isValid`
- `isEmpty`
- `toString`

The method `numObjects` simply returns the value of the instance variable `numObj`, which is updated each time an object is added or removed.

The other methods all use recursive helper functions. The helper method for `allObjects` does an in-order traversal of the BST.

The helper method for `add` uses the location of the object to search for the node where the object should be added, throwing an exception if there is already an object at that location. Otherwise, the helper method creates and adds a node at the appropriate position in the tree and returns the tree to the `add` method.

The helper method for `remove` has a twist. It carries the location as well as the object as parameters through the recursive calls. By using this technique, this helper method can also be used in the `recordMove` method, where the object already has a changed location, but `oldLoc` contains the location that is used to locate the object in the BST. This is the most complicated of the helper methods since after locating the node containing the object to be removed, it must carry out the removal. For clarity of code, the actual removal of the target node has been factored out in method `removeTargetNode` which is called from the `removeHelper` method when the object is found.

The method `recordMove` uses the helper method `removeHelper` to remove the object from its previous position. This works because although this particular object has changed its internal representation of its location, the other nodes in the BST have not and the old location is passed as a parameter that is used for the search. The search proceeds normally until the node with the object to be removed is reached. Since the first check at each node is to see if the object reference is the same as the object to be removed, the fact that the object being removed has a changed location will not affect the search. If the objects are not the same, then their locations are compared. If the object is removed successfully, then the `add` method is used to place it back into the BST at the correct position.

The `add`, `remove`, and `recordMove` methods all have expected O(log n) time (where n is the number of objects) for a balanced tree. Since this implementation does not balance the tree, the worst case of O(n) time could occur.

```
public class BSTUnboundedEnv extends SquareEnvironment
{
  private TreeNode root; // root of the BST
  private int numObj; // stores the number of objects

  // constructors
  public BSTUnboundedEnv()
  {
    // Construct and initialize inherited attributes.
    super();

    root = null;
    numObj = 0;
  }

  // accessor methods
```

```
public int numRows()
{
  return -1;
}

public int numCols()
{
  return -1;
}

public boolean isValid(Location loc)
{
  return loc != null;
}

public int numObjects()
{
  return numObj;
}


public Locatable[] allObjects()
{
  Locatable[] objectArray = new Locatable[numObjects()];

  // Put all the environment objects in the list.
  // Uses inorder traversal with allObjectsHelper method.
  allObjectsHelper(root, objectArray, 0);
  return objectArray;
}


private int allObjectsHelper(TreeNode node,
                             Locatable[] objectArray, int indexIn)
{
  if ( node != null )
  {
    int index = allObjectsHelper(node.getLeft(),
                                 objectArray, indexIn);

    objectArray[index] = (Locatable)node.getValue();
    index++;

    return allObjectsHelper(node.getRight(), objectArray, index);
  }
  else
  {
    return indexIn;
  }
}
```

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


public Locatable objectAt(Location loc)
{
  return objectAtHelper(root, loc);
}


private Locatable objectAtHelper(TreeNode node, Location loc)
{
  if ( node == null )
    return null;
  else
  {
    int cmp =
           loc.compareTo(((Locatable)node.getValue()).location());
    if (cmp == 0)
      return (Locatable)node.getValue();
    else if ( cmp < 0 )
      return objectAtHelper(node.getLeft(), loc);
    else // cmp > 0
      return objectAtHelper(node.getRight(), loc);
  }
}

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

public void add(Locatable obj)
{
  if ( !isEmpty(obj.location()) )
    throw new IllegalArgumentException("Location " + obj.location() +
                                    " is not a valid empty location");
  root = addHelper(root, obj);
}

private TreeNode addHelper(TreeNode node, Locatable obj)
{
  if ( node == null )
  {
    numObj++;
    return new TreeNode(obj);
  }

  Location loc = obj.location();
  if ( loc.compareTo(((Locatable)node.getValue()).location()) < 0 )
    node.setLeft(addHelper(node.getLeft(), obj));
  else //loc.compareTo(((Locatable)node.getValue()).location()) > 0
    node.setRight(addHelper(node.getRight(), obj));

  return node;
}

public void remove(Locatable obj)
{
  root = removeHelper(root, obj, obj.location());
}

private TreeNode removeHelper(TreeNode node,
                              Locatable obj,  Location loc)
{
  if ( node == null )
  {
    throw new IllegalArgumentException("Cannot remove " +
                                        obj + "; not there");
  }
  else if ( node.getValue().equals(obj) ) // found it
  {
    numObj--;
    return removeTargetNode(node);
  }
  else if ( ((Locatable)node.getValue()).location().equals(loc) )
  {
    throw new IllegalArgumentException("Cannot remove " + obj +
                                "; different object at its location");
  }
  else if (loc.compareTo(((Locatable)node.getValue()).location()) < 0)
  {
    node.setLeft(removeHelper(node.getLeft(), obj, loc));
    return node;
  }
  else //loc.compareTo(((Locatable)node.getValue()).location()) > 0
  {
    node.setRight(removeHelper(node.getRight(), obj, loc));
    return node;
  }
}
private TreeNode removeTargetNode(TreeNode target)
{
  if ( target.getRight() == null )
  {
    return target.getLeft();
  }
  else if ( target.getLeft() == null )
  {
    return target.getRight();
  }
  else if ( target.getRight().getLeft() == null )
  {
    target.setValue(target.getRight().getValue());
    target.setRight(target.getRight().getRight());
    return target;
  }
  else // right child has left child
  {
    TreeNode parent = target.getRight();
    while ( parent.getLeft().getLeft() != null )
      parent = parent.getLeft();

    target.setValue(parent.getLeft().getValue());
    parent.setLeft(parent.getLeft().getRight());
    return target;
  }
}


public void recordMove(Locatable obj, Location oldLoc)
{
  Location newLoc = obj.location();
  int oldCount = numObj;

  root = removeHelper(root, obj, oldLoc);

  if ( oldCount == numObj || !isEmpty(newLoc) )
  {
    throw new
           IllegalArgumentException("Precondition violation moving "
                                    + obj + " from " + oldLoc);
  }

  root = addHelper(root, obj);
}

}
```

6. The hash map code bears similarities to the sparse matrix and unbounded environment code. The code below lists only the modified methods with modifications shown in boldface.

```
import java.util.HashMap;

public class HMUnboundedEnv extends SquareEnvironment
{
  private HashMap objectMap;

  public HMUnboundedEnv()
  {
    // Construct and initialize inherited attributes.
    super();

    objectMap = new HashMap();
  }


  public int numObjects()
  {
    return objectMap.size();
  }

  public Locatable[] allObjects()
  {
    Locatable[] objectArray = new Locatable[objectMap.size()];

    // Put all the environment objects in the list.
    Iterator keyIterator = objectMap.keySet().iterator();
    for ( int index = 0; keyIterator.hasNext(); index++ )
    {
      Location loc = (Location) keyIterator.next();
      objectArray[index] = (Locatable) objectMap.get(loc);
    }
    return objectArray;
  }
```

Note to teachers: The above `allObjects` code is correct and in line with the AP Java subset (testable subset of the Java language). Two alternative solutions are on the following pages.

```
// Alternate code 1
  public Locatable[] allObjects()
  {
    Locatable[] objectArray = new Locatable[objectMap.size()];

    //  values not in subset
    Iterator objectIterator = objectMap.values().iterator();
    for ( int index = 0; objectIterator.hasNext(); index++ )
    {
      objectArray[index] = (Locatable) objectIterator.next());
    }

    return objectArray;
  }


// Alternate Code 2
  public Locatable[] allObjects()
  {
    Locatable[] objectArray = new Locatable[objectMap.size()];

    index = 0;
    //  values not in subset
    Iterator objectIterator = objectMap.values().iterator();

    while ( objectIterator.hasNext() )
    {
      objectArray[index] = (Locatable) objectIterator.next();
      index++;
    }
  }
```

Note to teachers: The following is the rest of the `HashMap` code.

```
  public boolean isEmpty(Location loc)
  {
    return (objectAt(loc) == null);
    // OR return ! objectMap.containsKey(loc);
  }

  public Locatable objectAt(Location loc)
  {
    return (Locatable) objectMap.get(loc);
  }
  // modifier methods

  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.
    objectMap.put(loc, obj);
  }

  public void remove(Locatable obj)
  {
    // Find the index of the object to remove.
    Location loc = obj.location();
    if ( ! objectMap.containsKey(loc) )
      throw new IllegalArgumentException("Cannot remove " +
                                         obj + "; not there");

    // Remove the object.
    objectMap.remove(loc);
  }

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

    // Otherwise, the object should still be mapped to the old
    // location, and nothing should be mapped (yet) to the new
    // location.
    Locatable foundObject = objectAt(oldLoc);
    if ( ! (foundObject == obj && isEmpty(newLoc)) )
      throw new
            IllegalArgumentException("Precondition violation moving"
                                  + obj + " from " + oldLoc);
    // Associate the object with the proper location.
    objectMap.remove(oldLoc);
    objectMap.put(newLoc, obj);
  }
}
```

Appendix A

Sample Multiple-Choice Questions

1. Assume that `fsh` has been defined and initialized as a `Fish` object in a client class that contains the following code segment.

```
int leftCounter = 0;

for (int k = 0; k < 100; k++)
{
  Direction dir = fsh.direction();
  Direction dirLeft = dir.toLeft();

  fsh.act();

  if ( /* condition */ )
    leftCounter++;
}
```

Which of the following could be used to replace `/* condition */` so that the variable `leftCounter` accurately stores the number of times that `fsh` turned to the left?

(A) `dir.equals(dirLeft)`
(B) `dir.equals(fsh.direction())`
(C) `dirLeft.equals(dir)`
(D) `dirLeft.equals(fsh.direction())`
(E) `(fsh.direction()).equals(new Direction(dir))`

2. Consider the following three statements.

I. `SlowFish` "is-a" `Fish`
II. `SlowFish` "is-a" `Locatable`
III. `Location` "is-a" `Locatable`

Which of the above statements is (are) true?

(A) I only
(B) II only
(C) III only
(D) I and II only
(E) I, II, and III

3. Three students attempt to create an `AnchoredFish` class. An `AnchoredFish` breeds and dies like a regular fish but never moves. All three students started with the following code.

```
public class AnchoredFish extends Fish
{
  // constructors not shown

  protected void generateChild(Location loc)
  {  /* implementation not shown */  }
}
```

Each of them chose to override a single method from the `Fish` class. However, each chose a different method.

I.
```
public void act()
{
  /* does nothing */
}
```
II.
```
protected void move()
{
  /* does nothing */
}
```
III.
```
protected Location nextLocation()
{
  return location();
}
```

Which of these attempts successfully create(s) an `AnchoredFish` class that breeds and dies like a regular fish but never moves?

(A) I only
(B) II only
(C) I and II only
(D) II and III only
(E) I, II, and III

(AB only)

4. If the 2-D array `theGrid` in the `BoundedEnv` class was replaced by a linked list of `Locatable` objects, which of the following method's running time would improve if the environment was sparsely populated?

   (A) `isValid`
   (B) `numObjects`
   (C) `allObjects`
   (D) `objectAt`
   (E) `add`

(AB only)

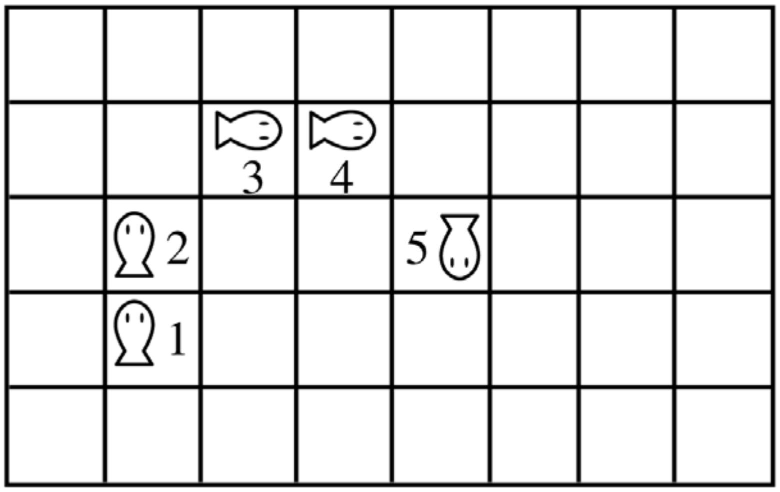
5. Let n be the number of fish in the environment. Assume that `UnboundedEnv` is implemented efficiently. What would be the average-case running time of the method `isEmpty` in `UnboundedEnv` if the environment was represented as a `HashMap` ?

   (A) $O(1)$
   (B) $O(\log n)$
   (C) $O(n)$
   (D) $O(n^2)$
   (E) The answer depends on the number of rows and columns occupied by fish.

Sample Free-Response Question (A Exam)

This question involves reasoning about the code from the *AP Marine Biology Simulation Case Study*. A copy of the code is provided as part of this exam.

Consider defining a new type of fish called `CircleFish` that swims in circles. A `CircleFish` alternates the way that it moves. It first goes one cell forward, if possible. Next it moves to the cell diagonally to the right if that cell is empty, turning right at the same time. (This is illustrated by moving from position 2 to position 3 in the diagram below.) If either type of move is blocked because the target cell is not empty, then the `CircleFish` stays in place, but turns right. Whenever a `CircleFish` turns, it should next attempt to move forward. The moves of a single `CircleFish` object are shown in the diagram below.



A `CircleFish` class can be defined by inheriting behavior from the `Fish` class and adding or overriding methods as appropriate. Since a `CircleFish` alternates its pattern of movement, it will need a state variable to keep track of which movement is next.

(a) Write a partial class declaration for `CircleFish` that includes the heading for the class, any instance variables that must be declared in the `CircleFish` class, and a two-parameter constructor that takes the environment and the initial location of the `CircleFish` as parameters. Other constructors and methods need not be shown.

(b) One way to implement `CircleFish` is to override the `nextLocation` method to return the next location of the `CircleFish`, which may be the cell immediately forward, the cell diagonally to the right, or the same cell that the `CircleFish` currently occupies, according to the rules given on the previous pages.

Write the `CircleFish` method `nextLocation`. If the state of the fish is such that this move should be forward, then `nextLocation` should return the location forward from this fish unless it is not empty, in which case the current location should be returned. If the state of the `CircleFish` is such that this move should be diagonally to the right, then `nextLocation` should return the location diagonally to the right, unless that location is not empty, in which case the current location should be returned.

In writing `nextLocation` you may use any accessible `CircleFish` methods and any other public classes and methods from the MBS case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `nextLocation` below.

```
protected Location nextLocation()
```

(c) Write the `CircleFish move` method. Method `move` should change the location and direction of the fish as needed, according to the rules of movement described at the beginning of the question. In addition, the state of the fish must be updated.

In writing `move` you may use any accessible `CircleFish` methods including `nextLocation` and any other public classes and methods from the MBS case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit. Assume that `nextLocation` works as specified, regardless of what you wrote in part (b).

Complete method `move` below.

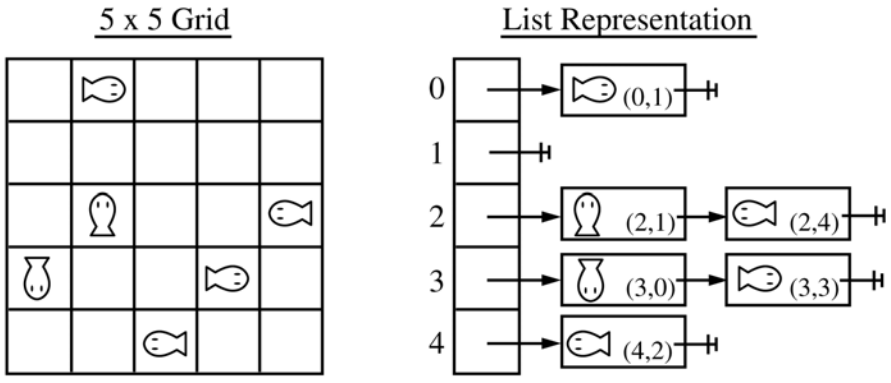
```
protected void move()
```

Sample Free-Response Question (AB Exam)

This question involves reasoning about the code from the *AP Marine Biology Simulation Case Study*. A copy of the code is provided as part of this exam.

In the original version of the case study, the `BoundedEnv` class uses a two-dimensional array of `Locatable` objects, `theGrid`, to represent the region in which the simulation takes place. Consider an alternate representation where the fish in each row are stored in a singly linked list. The implementation of `theGrid` becomes a one-dimensional array where each entry is a reference to the first node in the linked list for that row, or `null` if the row is empty. Each list node contains a fish and a reference to the node containing the next fish in that row. The linked list is ordered by column index of the location of the fish, from smallest to largest.

In the example below, a 5 x 5 grid is diagrammed on the left and the list representation of that grid is shown on the right. In this representation, `theGrid[2]` is a reference to the first node of a list containing two fish: a fish facing north at location (2,1) and a fish facing west at location (2,4). The element `theGrid[1]` is `null`, indicating that there are no fish in that row.



Each list of fish will be implemented using the `ListNode` implementation class provided in the Quick Reference.

Consider the following changes (shown in bold) to the `private` instance variables of the `BoundedEnv` class.

```
// Instance Variables: Encapsulated data for each
// BoundedEnv object
   private ListNode[] theGrid;    // array representing the
                                  // environment
   private int objectCount;       // # of objects in current
                                  // environment

   private int myNumCols;         // # of columns in the grid
```

(a) Modify the `BoundedEnv` method `allObjects` to use the revised data structure. In writing `allObjects`, you may use any other `BoundedEnv` methods or the public methods of any other class used in this case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit. Assume all methods work as specified.

Complete method `allObjects` below.

```
/** 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.

  // insert code here




  // end of inserted code

  return theObjects;
}
```

(b) Modify the `BoundedEnv` method `add` to use the revised data structure. The new `Locatable` object should be inserted into the correct row's linked list, maintaining the order of the list sorted by the object's location's column index.

In writing `add`, you may use any other `BoundedEnv` methods or the public methods of any other class used in this case study. Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit. Assume that all methods work as specified.

Complete method `add` below.

```
/** Adds a new object to this environment at the
 *  location it specifies.
 *  (Precondition: <code>obj.location()</code> 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.

  // insert code here
```

Appendix B

Solutions for Sample Test Questions

Multiple-Choice

1. (D) `dirLeft.equals(fsh.direction())`

   Rationale:
   Each time through the loop, the variable `dir` holds the direction of the fish before it acts, and `dirLeft` holds the direction facing left of the fish's direction before the call to `act`. After the call to `act`, the current direction of the fish can only be obtained through the accessor method `direction`.

2. (D) I and II only

   Rationale:
   The `Fish` class implements the `Locatable` interface and the `SlowFish` class extends the `Fish` class. The `Location` class is a separate class encapsulating a location in the grid. It does not fall into the hierarchy of the `Fish` class and its subclasses.

3. (D) II and III only

   Rationale:
   Overriding the `move` method to do nothing will cause the fish to stay in the same location. Overriding the `nextLocation` method to return the current location will also keep the fish in the same location. In each of these cases, the fish will still breed and die as inherited from the `Fish` class. Overriding the act method to do nothing will keep the fish in the same location, but will not allow the fish to breed and die as specified.

4. (C) `allObjects`

   Rationale:
   The performance of the `isValid` and `numObjects` methods would be about the same as before. The performance of the `allObjects` method improves because the loop will only traverse locations that hold fish rather than traversing the entire grid as in the original implementation. The performance of the `objectAt` and `add` methods is worse because the appropriate location must be found within the linked list.

5. (A) $O(1)$

   Rationale:
   Method `isEmpty` will hash based on the location to determine if a fish is present at that location. The question states that `UnboundedEnv` is implemented efficiently. Lookup in a hash table is $O(1)$.

Sample Free-Response Solution (A Exam)

(a)

```
public class CircleFish extends Fish
{
  private boolean justTurned;

  public CircleFish(Environment env, Location loc)
  {
    super(env, loc);
    justTurned = true;
  }

  // other constructors and methods not shown
}
```

(b)

```
protected Location nextLocation()
{
  Environment env = environment();
  Location forward = env.getNeighbor(location(), direction());
  Location rightDiag = env.getNeighbor(forward,
                                       direction().toRight());
  if (justTurned && env.isEmpty(forward))
  {
    return forward;
  }
  else if (!justTurned && env.isEmpty(rightDiag))
  {
    return rightDiag;
  }
  else
  {
    return location();
  }
}
```

(c)

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

  if (nextLoc.equals(oldLoc))
  {
    changeDirection(direction().toRight());
    justTurned = true;
  }
  else
  {
    changeLocation(nextLoc);

    if (!justTurned)
    {
      changeDirection(direction().toRight());
      justTurned = true;
    }
    else
    {
      justTurned = false;
    }
  }
}
```

Sample Free-Response Solution (AB Exam)

(a)

```
/** 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.

  // insert code here

  ListNode curNode;
  for (int r = 0; r < numRows(); r++)
  {
    curNode = theGrid[r];
    while (curNode != null)
    {
      theObjects[tempObjectCount] = (Locatable)curNode.getValue();
      curNode = curNode.getNext();
      tempObjectCount++;
    }
  }

  // end of inserted code here

  return theObjects;
}
```

(b)

```
/** Adds a new object to this environment at the location
 *  it specifies.
 *  (Precondition: <code>obj.location()</code> is a valid empty
 *   location.)
 *  @param obj the new object to be added
 *  @throws   IllegalArgumentException if 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.

  // insert code here

  if (theGrid[loc.row()] == null ||
      ((Locatable)theGrid[loc.row()].getValue()).location().col() >
                                                            loc.col())
  {
    theGrid[loc.row()] = new ListNode(obj, theGrid[loc.row()]);
  }
  else
  {
    ListNode curNode = theGrid[loc.row()];
    while (curNode.getNext() != null &&
         ((Locatable)curNode.getNext().getValue()).location().col() <
                                                           loc.col())
    {
      curNode = curNode.getNext();
    }
    ListNode temp = new ListNode(obj, curNode.getNext());
    curNode.setNext(temp);
  }

  objectCount++;
}
```

Appendix C

Chapter 1

Exercise Set 2: Problem 1 (page 11)

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

Chapter 1

Analysis Question Set 2: Problem 2 (page 12)

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						
21						
22						
23						
24						
25						

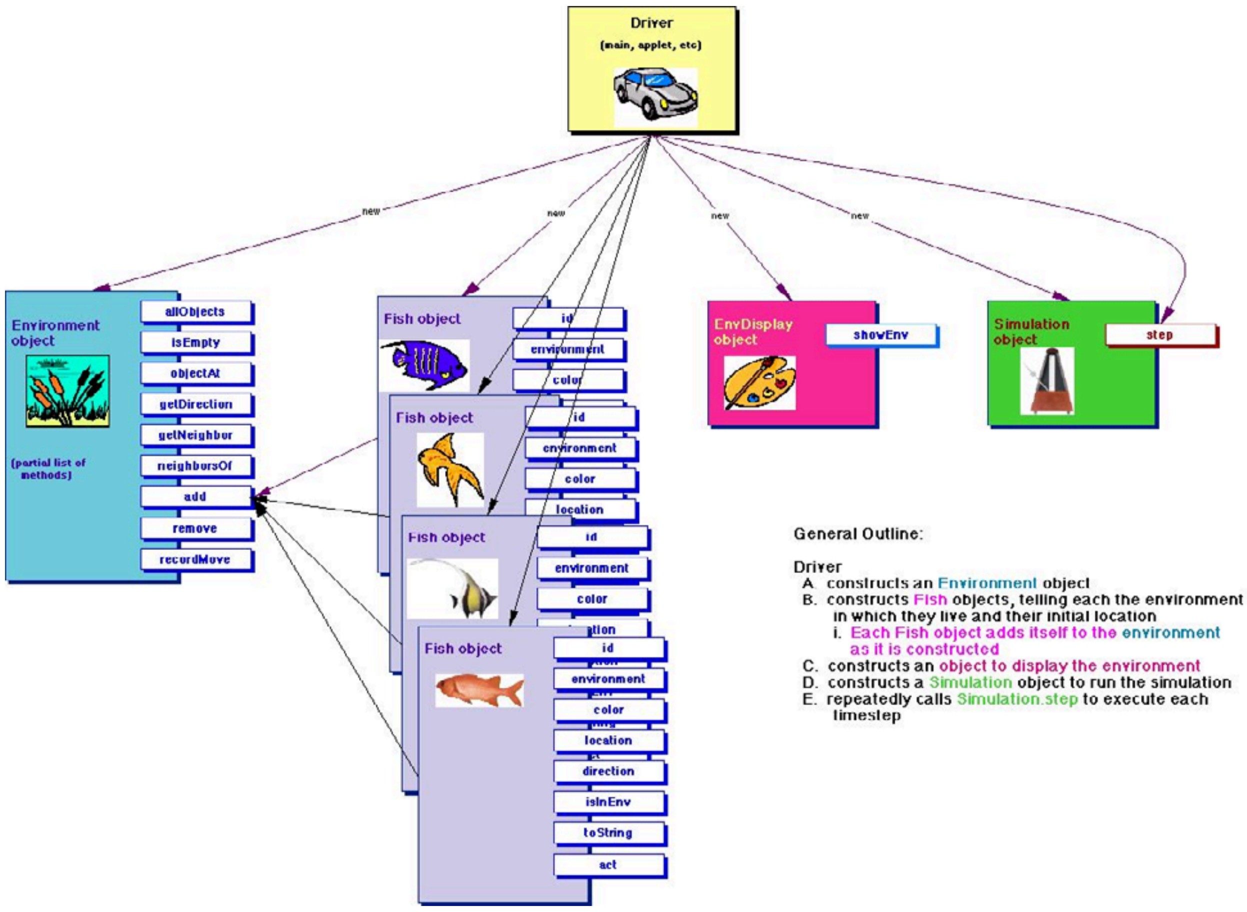
Chapter 2

Exercise Set 5: Problem 3 (page 45)

Step	Fish 1	Fish 2	Fish 3	Fish 4	Fish 5	Fish 6
0						
1						
2						
3						
4						
5						

Chapter 1

Object Diagram: Driver (page 17)



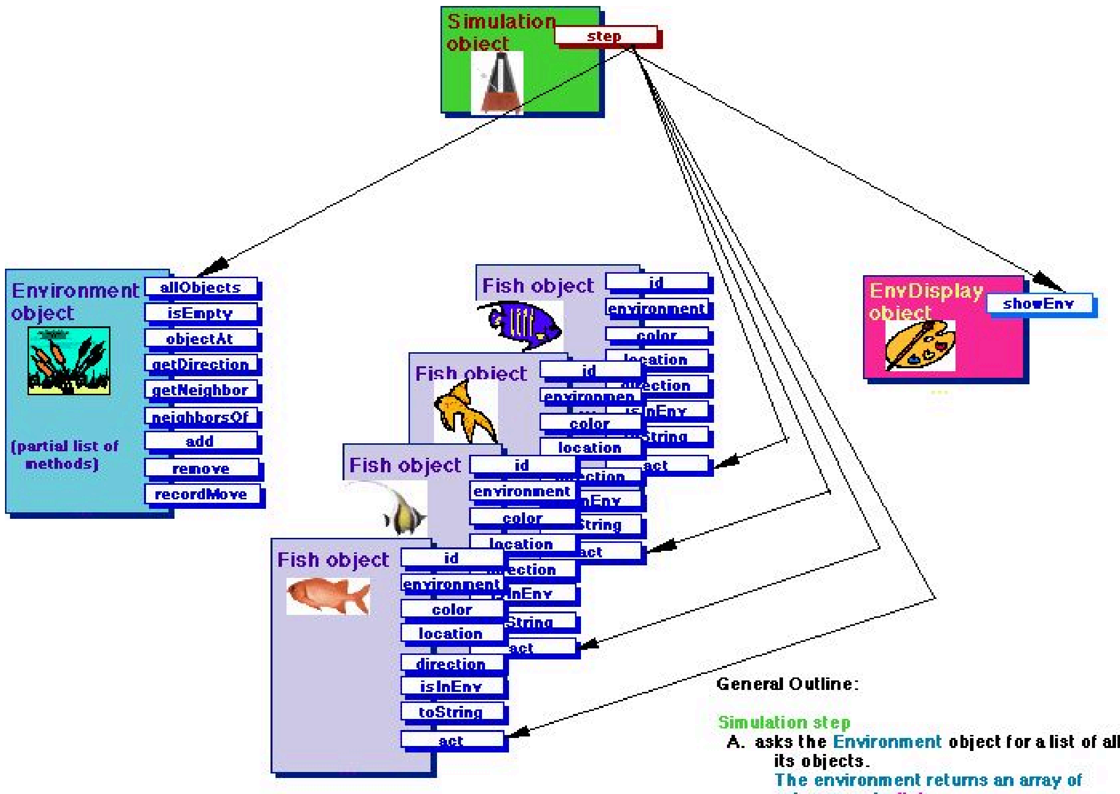
General Outline:

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 Simulation.step to execute each timestep

Chapter 2

Object Diagram: Simulation – step (page 23)



General Outline:

Simulation step

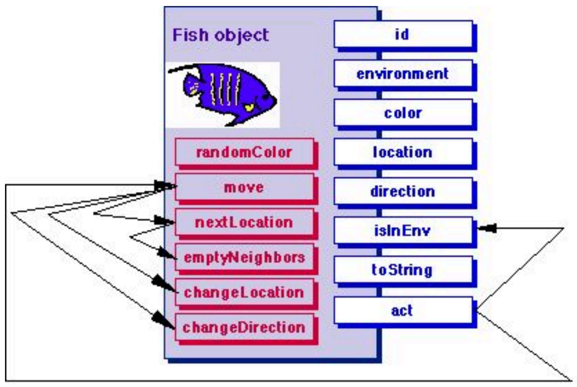
A. asks the Environment object for a list of all its objects.
   The environment returns an array of references to fish.

B. asks each Fish to act for this timestep

C. asks the EnvDisplay object to display the new state of the environment at the end of the timestep

Chapter 2

Object Diagram: Fish – act (page 34)



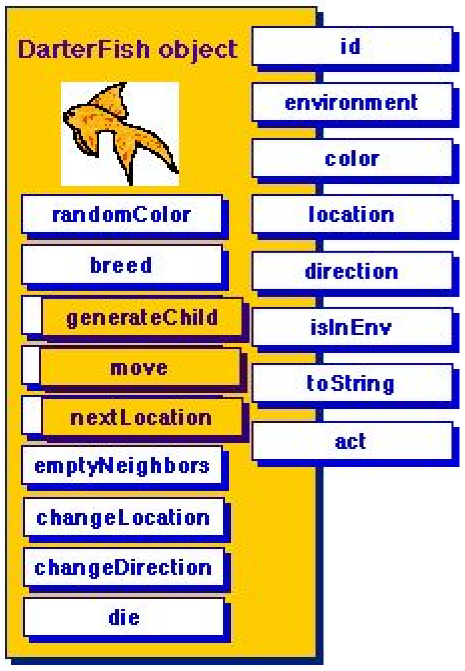
General Outline:

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

Chapter 4

Object Diagram: DarterFish (page 65)

DarterFish



Appendix D

Supplemental Example: Fast Fish

Problem Specification

The marine biologists decided that they would also like faster-moving fish in the simulation. In particular, the biologists decided that:

A fast fish can move one or two cells away in a single timestep, in any of the four directions. It can only move to an empty location two cells away if the first cell in the two-step sequence is empty also. In other words, if an immediately adjacent location is empty, the fast fish looks at the cell beyond it in the same direction to see if it is empty also. The left diagram below shows all the possible neighboring locations (`~`) of a fast fish (`F`). The diagram on the right shows how neighboring fish (`N`) can keep the fast fish from moving to some of its empty neighboring locations (the ones marked `X`). Fast fish, like other fish, move only if they do not breed.

```
        ~                  X
        ~                  N
    ~ ~ F ~ ~          X N F ~ ~
        ~                  ~
        ~                  ~
```

Implementation of the `FastFish` Class

The specification for a fast fish states that in a single timestep it can move one or two cells away in any of the four neighboring directions. It can only move to empty cells, however, and it can only move to a cell two units away if the cell it would be swimming through to get there is also empty.

The key to the movement of fast fish is the way they choose the empty neighboring locations to which they could move. Fast fish move within a wider neighborhood than just their immediately adjacent neighbors. I decided to create a new method, `findMoveLocs`, that would find the possible move locations in the wider neighborhood. I then redefined the `nextLocation` method to call `findMoveLocs` rather than calling `emptyNeighbors`. It also no longer needs to remove the location behind the fish from the list of possible moves. The following code is for the redefined `nextLocation` method, without debugging messages and with the key changes in boldface.

```
protected Location nextLocation()
{
  // Get list of possible move locations.
  ArrayList moveLocations = findMoveLocs();

  // If there are no possible moves, then we're done.
  if ( moveLocations.size() == 0 )
    return location();

  // Randomly choose one neighboring empty location and return it.
  Random randNumGen = RandNumGenerator.getInstance();
  int randNum = randNumGen.nextInt(moveLocations.size());
  return (Location) moveLocations.get(randNum);
}
```

The neighborhood for a fast fish consists of its immediately adjacent empty neighbors and some of their immediately adjacent neighbors. The first thing the `findMoveLocs` method does, therefore, is to call the inherited `emptyNeighbors` method to get a list of the fish's immediately adjacent empty neighbors. Next, `findMoveLocs` steps through that list, putting the empty neighbors, which are among the possible move locations, in a new list, and checking for empty neighbors two cells away. For example, if the cell to the fish's north is in the list of empty neighbors, then the method looks to see if the cell north of that northern neighbor is also empty. It does this by storing the immediate neighbor (called `neighborLoc`) in a local variable, asking the environment for the direction to `neighborLoc` (called `dirToNeighbor`), asking the environment for the next location in the same direction (called `nextOver`), and then checking to see if `nextOver` is a valid, empty location in the environment. If it is, then the further cell's location is added to the list of possible move locations.

The code that follows is for the `findMoveLocs` method, which is shown without debugging messages.

```
protected ArrayList findMoveLocs()
{
  Environment env = environment();
  // Generate a list of all the immediately adjacent empty neighbors.
  ArrayList emptyNbrs = emptyNeighbors();

  // Build list of possible move locations.
  ArrayList moveLocations = new ArrayList();
  for ( int index = 0; index < emptyNbrs.size(); index++ )
  {
    // Add this location to list of possible moves.
    Location neighborLoc = (Location) emptyNbrs.get(index);
    moveLocations.add(neighborLoc);
```

```
      // Find the next location over in the same direction.
      Direction dirToNeighbor =
                            env.getDirection(location(), neighborLoc);
      Location nextOver = env.getNeighbor(neighborLoc, dirToNeighbor);

      if ( env.isEmpty(nextOver) )
      {
        // Next location over is empty, so add it too.
        moveLocations.add(nextOver);
      }
    }

    return moveLocations;
  }
```

As with the darter and slow fish, I decided to make all fast fish one color (cyan) to help them show up as I ran the simulation. Since `FastFish` does not use any additional instance variables, the `FastFish` constructors are very simple. The redefined `generateChild` method is just as straightforward.

Analysis Question Set 1:

1. What if location `nextOver` is outside the bounds of the environment? Will `findMoveLocs` handle this case correctly? (Hint: read the class documentation for the `Environment` class. What is the behavior of the `isEmpty` method?)

2. Consider the following four-by-four environment with a single fast fish (`F`) in it.

```
F  ~  ~  ~
~  ~  ~  ~
~  ~  ~  ~
~  ~  ~  ~
```

   What locations would you expect to be in the `ArrayList` returned by `findMoveLocs`, given the problem specification? Using paper and pencil, "hand-execute" (or "mind-execute") the code. Keep track of the loop index, `index`, the contents of `moveLocations`, and the values of the other local variables for every step through the loop.

Testing the `FastFish` Class

As I did with `DarterFish` and `SlowFish`, I decided to start my testing of `FastFish` with regression tests to make sure that I had not done anything to break the `Fish`, `DarterFish`, or `SlowFish` classes (or other classes in the marine biology simulation). My regression tests ran without any problem.

Next, I developed the black box test cases to test the `FastFish` class, basing them on the test cases for `Fish`. The list below shows some of the interesting boundary cases I identified. (It does not show all the new and modified test cases.)

- A fast fish with no empty adjacent locations but four empty locations two cells away should not move.
- A fast fish with exactly one empty adjacent location should always move there.
- A fast fish with four empty adjacent locations but no empty locations two cells away should move to the four empty neighbors with equal probability.
- A fast fish with four empty adjacent locations and four empty locations two cells away should move to each of the eight possible move locations with equal probability.

I then considered the code in the `nextLocation` and `findMoveLocs` methods in the `FastFish` class to see if I needed to develop additional test cases.

Analysis Question Set 2:

1. Why did Pat introduce the `findMoveLocs` method instead of redefining the behavior of the `emptyNeighbors` method?

2. Based on the code in `nextLocation` and `findMoveLocs`, did Pat need to develop additional code-based test cases?

3. Sometimes when adding functionality to a program, you realize that you could have designed it differently in the first place in a way that would make it easier to modify now. You might even decide that it's worth going back and changing the original before making your new changes. (This is sometimes called *refactoring*.) For example, Pat could have decided to break up the `nextLocation` method in the `Fish` class and create a `findMoveLocs` method there. If Pat had done this, which pieces of the current `nextLocation` method in `Fish` would stay in `nextLocation` and which would move to `findMoveLocs` ? What methods would `FastFish` have to redefine? What would be the impact on `DarterFish` and `SlowFish` ? What are the advantages or disadvantages of this solution compared to the solution Pat actually implemented?

Exercise Set 1:

1. Make a copy of the `3species.dat` initial configuration file, rename it `4species.dat` and modify it to include some fast fish. Run the marine biology simulation with this file to see how the behavior has changed. (Again, you may find it easier to see the differences in types of movement without breeding and dying behavior.)

2. Redefine the `toString` method in `FastFish` to clarify that this is a fast fish. This makes it easier to keep track of slow, fast, and normal fish in the debugging output. Turn on debugging in the `Simulation step` method, if it isn't on already, and run the simulation again. This will let you observe the changed behavior at a greater level of detail. Run the simulation for several timesteps with debugging turned on. What evidence do you have that the program is working or not working?

3. Run the simulation program with fast fish for 10 timesteps and record the results of your tests. What percentages of fast fish bred and died during your test run? Continue the program for another 10 timesteps. What percentages of fast fish bred and died over 20 timesteps?

4. Modify the `FastFish` class so that fast fish can move one or two cells forward or to the side, but can only move one cell backward.

5. Modify the `FastFish` class so that fast fish can move one or two cells forward or one cell to either side, but never move backward in a single timestep.

6. Refactor the `Fish` class so that it has a `findMoveLocs` method that creates an `ArrayList` of the locations that the fish could move to. Its `nextLocation` method should call `findMoveLocs` and then randomly choose one of the possible move locations. Test the `Fish` class to make sure that its behavior is unchanged. Finally, modify the `FastFish` class to work with the modified `Fish` class, and test it. (See Question 3 in the Analysis Question Set on the previous page.)

Quick Reference for `FastFish`

This quick reference lists the constructors and methods associated with the specialized `FastFish` class, introduced in this example. 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)

```
FastFish Class (extends Fish)

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

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

Code for `FastFish` Class

```
// Class: FastFish class
//
// Author: Alyce Brady
//
// This class is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation.
//
// This class is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.

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

/**
 *  Marine Biology Simulation:
 *  The FastFish class represents a fish in the Marine Biology
 *  Simulation that moves very quickly. A fast fish looks for empty
 *  neighbors that are one or two cells away from it.  It can only "see"
 *  an empty location two cells away if the cell in between is empty
 *  also.  In other words, if an immediately adjacent location is empty,
 *  the fast fish looks at the cell beyond it in the same direction to
 *  see if it is empty also.  The left diagram below shows all the
 *  possible neighboring locations (~) of a fast fish (F). The diagram
 *  on the right shows how neighboring fish (N) can keep the fast fish
 *  from seeing some of its empty neighboring locations (the ones
 *  marked X).
 *  <pre>
 *          ~                   X
 *          ~                   N
 *      ~ ~ F ~ ~           X N F ~ ~
 *          ~                   ~
 *          ~                   ~
 *  </pre>
 *
 *  <p>
 *  FastFish objects inherit instance variables and much of their
 *  behavior from the Fish class.
 *
 *  @author Alyce Brady
 *  @author APCS Development Committee
 *  @version 1 June 2002
 **/

public class FastFish extends Fish
{
  // constructors

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

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

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


  // redefined methods

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

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

  /** Finds this fish's next location.
   *  Fast fish may move to an empty adjacent location one or two
   *  cells away in any direction.  To move to a location two cells
   *  away, though, the intervening location must also be empty.
   *  If the fast fish cannot move, <code>nextLocation</code>
   *  returns the fish's current location.
   *  @return    the next location for this fish
   **/
  protected Location nextLocation()
  {
    // Get list of possible move locations.
    ArrayList moveLocations = findMoveLocs();
    Debug.print("Possible new postions are: " +
                 moveLocations.toString());

    // If there are no possible moves, then we're done.
    if ( moveLocations.size() == 0 )
      return location();

    // Randomly choose one neighboring empty location and return it.
    Random randNumGen = RandNumGenerator.getInstance();
    int randNum = randNumGen.nextInt(moveLocations.size());
    return (Location) moveLocations.get(randNum);
  }

  /** Finds locations to which this fish might move.
   *  Fast fish can move further than to just their immediate neighbors.
   *  They can move up to two squares in each of the 4 directions, if
   *  not blocked.
   *  @return    a list of locations to which this fish might move
   **/
  protected ArrayList findMoveLocs()
  {
    Environment env = environment();
    // Generate a list of all the immediately adjacent empty neighbors.
    ArrayList emptyNbrs = emptyNeighbors();
    Debug.println("Has adjacent empty positions: " +
                  emptyNbrs.toString());

    // Build list of possible move locations.
    ArrayList moveLocations = new ArrayList();
    for ( int index = 0; index < emptyNbrs.size(); index++ )
    {
      // Add this location to list of possible moves.
      Location neighborLoc = (Location) emptyNbrs.get(index);
      moveLocations.add(neighborLoc);

      // Find the next location over in the same direction.
      Direction dirToNeighbor =
                          env.getDirection(location(), neighborLoc);
      Location nextOver = env.getNeighbor(neighborLoc, dirToNeighbor);

      if ( env.isEmpty(nextOver) )
      {
        // Next location over is empty, so add it too.
        moveLocations.add(nextOver);
      }
    }

    return moveLocations;
  }

}
```

Answers for `FastFish` Exercises

Analysis Question Set 1

1. Yes, this case is handled correctly. The `findMoveLocs` method will work correctly even when the `nextOver` variable refers to a location outside the bounds of the environment, because `isEmpty` returns `true` only if the location is both in bounds and empty.

2. Given the problem specification, one would expect `findMoveLocs` to return an `ArrayList` containing the locations (0, 1), (0, 2), (1, 0), and (2, 0). Students should be able to "mind-execute" the code to verify their understanding of it. The `emptyNbrs` local variable is initialized to the `ArrayList` returned by the `emptyNeighbors` method, which contains two locations, (0, 1) and (1, 0). The `moveLocations` local variable is then initialized to be an empty `ArrayList`. The code enters the loop, setting `index` to 0. This is less than the size of the `emptyNbrs ArrayList` (which is 2), so we enter the loop. The `neighborLoc` variable is set to the first location in `emptyNbrs`, (0, 1), which is also added to the previously empty `moveLocations ArrayList`. The `dirToNeighbor` local variable is set to `Direction.EAST` (the direction from the current location, (0, 0), to (0, 1)). Then the `nextOver` variable is set to (0, 2), which is the location east of (0, 1). Since that location is in bounds and is empty, it is added to `moveLocations`, which now contains (0, 1) and (0, 2). The looping variable, `index`, is incremented from 0 to 1. This is still less than the size of the `emptyNbrs` list, so we re-enter the loop. This time `neighborLoc` is set to (1, 0), the second location in `emptyNbrs`, which is then added to `moveLocations`. The `dirToNeighbor` variable is set to `Direction.SOUTH`, the direction from (0, 0) to (1, 0), and `nextOver` is set to (2, 0). Location (2, 0) is empty, and so it is added to `moveLocations`. The looping variable, `index`, is incremented from 1 to 2, which is NOT less than the size of the `emptyNbrs` list, so we are done with the loop. The final contents of `moveLocations` when it is returned to `nextLocation` is {(0, 1), (0, 2), (1, 0), (2, 0)}.

Analysis Question Set 2

1. The method `emptyNeighbors` is also used by `breed`, and the problem specification does not say that fast fish breed into all empty locations in the wider neighborhood.

2. The boundary cases listed cover the test conditions from these two methods. You could ask students to identify which boundary cases test the boundaries of the loop in `findMoveLocs` and which ones test each of the conditional statements in `nextLocation` and `findMoveLocs`.

3. The `nextLocation` in `Fish` would be exactly the same as the `nextLocation` in `FastFish` in the current solution. The `findMoveLocs` in `Fish` would call `emptyNeighbors` and remove the location behind. `FastFish` would redefine only the `findMoveLocs` method. Neither `DarterFish` nor `SlowFish` would be affected in any way; both would redefine `nextLocation`, as they do now. `SlowFish` would make use of the new `findMoveLocs` method in `Fish` through its call to `super.move()`. `DarterFish`, with its deterministic `nextLocation` method, doesn't have any use for the `findMoveLocs` method.

Exercise Set 1

1. Students should observe that each species moves in the way its `move` method has been defined.

2. Students are being asked to redefine the `toString` method in the `FastFish` class. One simple example is given below. Encourage your students to find other ways to make the fast fish information stand out, especially when there is more than one species in the environment.

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

3. Answers will vary.

4. One possible answer is shown below with changed code in bold.

```
protected ArrayList findMoveLocs()
{
  Environment env = environment();
  ArrayList emptyNbrs = emptyNeighbors();
  Direction oppositeDir = direction().reverse();

  ArrayList moveLocations = new ArrayList();
  for (int index = 0; index < emptyNbrs.size(); index++ )
  {
    Location neighborLoc = (Location) emptyNbrs.get(index);
    moveLocations.add(neighborLoc);

    Direction dirToNeighbor =
                      env.getDirection(location(), neighborLoc);
    Location nextOver = env.getNeighbor(neighborLoc, dirToNeighbor);


    if (env.isEmpty(nextOver) &&
                           (! dirToNeighbor.equals(oppositeDir)))
    {
      moveLocations.add(nextOver);
    }
  }

  return moveLocations;
}
```

5. One possible solution is shown below with changed code in bold.

```
protected ArrayList findMoveLocs()
{
  Environment env = environment();
  ArrayList emptyNbrs = emptyNeighbors();
  Direction oppositeDir = direction().reverse();
  Location locationBehind = env.getNeighbor(location(), oppositeDir);

  ArrayList moveLocations = new ArrayList();
  for (int index = 0; index < emptyNbrs.size(); index++ )
  {
    Location neighborLoc = (Location) emptyNbrs.get(index);
    moveLocations.add(neighborLoc);

    Direction dirToNeighbor =
                           env.getDirection(location(), neighborLoc);
    Location nextOver = env.getNeighbor(neighborLoc,  dirToNeighbor);

    if (env.isEmpty(nextOver) && dirToNeighbor.equals(direction()))
    {
      moveLocations.add(nextOver);
    }

  }
  moveLocations.remove(locationBehind);

  return moveLocations;
}
```

6. The following are the modified methods for refactored `Fish` class. The `findMoveLocs` method has been added to the `Fish` class and the only class method to change is `nextLocation`. The only change needed for the `FastFish` class is to remove the redefined `nextLocation` method.

```
// Additional method findMoveLocs in Fish class

  /** Finds locations to which this fish might move.
   *  A fish may move to any empty adjacent locations except the one
   *  behind it (fish do not move backwards).
   *  @return    a list of locations to which this fish might move
   **/
  protected ArrayList findMoveLocs()
  {
    // Get list of neighboring empty locations.
    ArrayList emptyNbrs = emptyNeighbors();
    Debug.println("Has adjacent empty positions: " +
                    emptyNbrs.toString());

    // Remove the location behind, since fish do not move backwards.
    Direction oppositeDir = direction().reverse();
    Location locationBehind = environment().getNeighbor(location(),
                                                         oppositeDir);
    emptyNbrs.remove(locationBehind);

    // If no valid empty neighboring locations, then we're done.
    return emptyNbrs;
  }


// Modified version of nextLocation method in Fish class

  /** Finds this fish's next location.
   *  Finds the possible move locations for this fish and then randomly
   *  chooses one.  If this fish cannot move, <code>nextLocation</code>
   *  returns its current location.
   *  @return    the next location for this fish
   **/
  protected Location nextLocation()
  {
    // Get list of possible move locations.
    ArrayList moveLocations = findMoveLocs();
    Debug.print("Possible new locations are: " +
                  moveLocations.toString());

    // If there are no possible moves, then we're done.
    if ( moveLocations.size() == 0 )
      return location();

    // Randomly choose one neighboring empty location and return it.
    Random randNumGen = RandNumGenerator.getInstance();
    int randNum = randNumGen.nextInt(moveLocations.size());
    return (Location) moveLocations.get(randNum);
  }
```

6/2003 I.N. 997121