CS 61B Project 1
Sharks and Fish
Due 3pm Wednesday, February 16, 2000

Warning: This project is time-consuming. Start early.

This is an individual assignment; you may not share code with other students.

Getting started: You will find the code for this assignment in $master/hw/pj1/. Start by copying it into your own pj1 directory.

In this project you will implement a simulation of an ocean containing sharks and fish. You will also write code to convert an ocean into a run-length encoding and back. The ocean is rectangular, but the edges are connected together in a topological donut or torus. This means that the top (North) and bottom (South) edges are considered adjacent, so if you start at the top edge and go up, you'll be at the bottom edge (just like in the video game Asteroids). Similarly, the East and West edges are connected (just like in Pac Man). The ocean is divided into square cells, which are indexed as follows (for a 4x3 ocean):

```
-----------------------------
| 0, 0 | 1, 0 | 2, 0 | 3, 0 |
-----------------------------
| 0, 1 | 1, 1 | 2, 1 | 3, 1 |
-----------------------------
| 0, 2 | 1, 2 | 2, 2 | 3, 2 |
-----------------------------
```

Note that the origin is in the upper left; the x-coordinate increases as you move right, and the y-coordinate increases as you go down. (This conforms to Java's graphics commands, though you won't need to use them directly in this lab.) You can also refer to locations such as (4, 0) or (-4, 3), which are both the same as (0, 0) in a 4x3 ocean. (More generally, the coordinates in an ixj ocean are taken modulo i for the x-coordinate, which is horizontal, and modulo j for the y-coordinate, which is vertical.) Any pair of integers will give you a valid position in the grid by "wrapping around" at the edges.

There are two kinds of entities in this ocean: sharks and fish. The sharks and fish breed, eat, and die in the ocean. Each cell of the grid can be occupied by a single shark or fish, or it can be empty.

Part I: Simulating Sharks and Fish

This part is worth 40% of your total score. (8 points out of 20).

The ocean is parametrized by its size and the initial placement of sharks and fish in the ocean. The simulation is also governed by one other parameter, called the starveTime for a shark. This is the number of simulation timesteps that a shark can live without eating.

The simulation proceeds in timesteps. The rules for how the ocean looks at any given timestep depend only on the occupants of the cells during the previous timestep. Therefore, to obtain correct behavior, you will sometimes be working with two copies of the ocean simultaneously; one representing the ocean during one timestep, and the other representing the ocean during the very next timestep. (If you are foolish enough to try to implement a timestep using just a single ocean object, you will modify the values of cells whose old values are still needed to compute the new values for other cells, and thus you will compute the wrong answer.)

The contents of any particular cell of the ocean depend only on the contents of that cell and its neighbors in the previous timestep. "Neighbors" refers to the eight adjacent cells, including cells immediately to the north, south, east, and west, as well as four diagonal neighbors. Here are the rules:

1) If a cell contains a fish, and any of its neighbors is a shark, then the fish is eaten by a shark, and therefore disappears.

2) If a cell contains a fish, and all of its neighbors are either empty or are other fish, then the fish stays where it is.

3) If a cell contains a shark, and any of its neighbors is a fish, then the shark stays where it is, but we assume that it has eaten in this timestep. (Note that we may have multiple sharks sharing the same fish. This is fine; we'll assume they all get enough to eat.)

4) If a cell contains a shark, and none of its neighbors is a fish, it gets hungrier. If the last timestep in which the shark ate was more than starveTime timesteps ago, then the shark dies (disappears). An example demonstrating this rule appears below.

5) If a cell is empty, at least two of its neighbors are fish, and none of its neighbors is a shark, then a new fish is born in that cell.

6) If a cell is empty, at least two of its neighbors are sharks, and at least two of its neighbors are fish, then a new shark is born in that cell. Sharks are well-fed at birth, so they can survive starveTime timesteps without eating.

7) If a cell is empty, and rules 5 and 6 do not apply, then the cell remains empty. (This is true if fewer than two of its neighbors are fish, or exactly one of its neighbors is a shark.)

Note on starveTime

The following example will demonstrate exactly when sharks will die. Suppose the starveTime is 3. At the start of the simulation, say it is time 0, and we assume that all of the sharks have just been born. A shark that does not ever get to eat will exist in timesteps 0 through 3, but at timestep 4, the shark will not exist. Similarly, if a shark is fed at time 3, but not after that, it will live through timestep 6, but will not exist in timestep 7.

```
Time 0   shark A born                shark B born
Time 1
Time 2
Time 3                               shark B eats a fish (born in timestep 2)
Time 4   shark A no longer exists
Time 5
Time 6
Time 7                               shark B no longer exists
```

A shark is well-fed at birth, so a shark born in timestep 3 remains alive during timestep 6 but has died by timestep 7 if it never got to eat.

Your task

Fill in the implementation of the Ocean class. We have already provided you with the public method definitions you will need. You are required to provide implementations of all the methods whose prototypes appear in Ocean.java. Among these methods is a constructor that takes two integers as input, representing the size of the ocean, and returns an ocean of the specified size. For example, the statement

```
    Ocean sea = new Ocean(i, j);
```

should create an ixj Ocean object. In your implementation, you may define any fields, additional methods, additional classes, or other .java files you wish, but you cannot change the prototypes in Ocean.java. We will test your code by calling your methods directly, so it is important that you follow this rule. You should read Ocean.java carefully for an explanation of what methods you must write. The most important of these is timeStep, the method that performs your simulation.

Your Ocean class should represent the ocean as a simple two-dimensional array. It is up to you decide how to represent each element of the array (including whether it is empty or contains a shark or fish, and if it's a shark, how recently it last ate). We have provided Java classes to help you debug your implementation and animate your ocean, in these files:

```
    Simulation.java
    SimText.java
```

The Simulation and SimText classes (which consist primarily of a simulation driver called main) generate random input to initialize the ocean, and animate the sequence of oceans returned by the timeStep method of your Ocean class. So that they can initialize your Ocean and monitor the fish and sharks in your Ocean during the simulation, you must implement the methods addFish, addShark, cellContents, width, and height.

The Simulation and SimText programs take up to three parameters. The first two specify the width and height of the ocean. The third parameter specifies the value of starveTime. For example, if you run

```
    java Simulation 25 30 1
```

then Simulation will animate a 25x30 ocean with a starveTime of 1. If you run "java Simulation" with no parameters, by default Simulation will animate a 50x25 ocean with a starveTime of 3. With some choices of parameters, the ocean quickly dies out; with others, it teems forever.

In the animation produced by Simulation, sharks are red squares and fish are green squares. In the animation produced by SimText, sharks are 'S' characters and fish are '~' characters. Simulation is more fun to watch, but SimText may be easier to use for debugging and modem access.

Part II: Converting a Run-Length Encoding to an Ocean

This part is worth 25% of your total score. (5 points out of 20).

For a large ocean, an Ocean object can consume quite a bit of memory or disk space. For long-term storage, we can store an Ocean more efficiently if we represent it as a "run-length encoding." Imagine taking all the rows of cells in the ocean, and connecting them into one long strip. Think of the cells as being numbered thusly:

```
                        -----------------------------
                        |  0  |  1  |  2  |  3  |
                        -----------------------------
                        |  4  |  5  |  6  |  7  |
                        -----------------------------
                        |  8  |  9  | 10  | 11  |
                        -----------------------------
```

Typically, many regions of this strip are "runs" of many empty cells in a row, or many fish in a row, or many equally-hungry sharks in a row. Run-length encoding is a technique in which a sequence of identical consecutive cells are represented as a single record or object. For instance, the following strip of fish (F), sharks fed two timesteps ago (S), and empty cells (.):

```
              -------------------------------------------------
              | F | F | F | S | S | S | S | S | . | . | . | . |
              -------------------------------------------------
```

could be represented with just three records, each representing one "run":

```
                              ----------------
                              | F3 | S5 | .4 |
                              ----------------
```

"F3" means that there are three consecutive fish, followed by "S5", meaning five consecutive sharks fed two timesteps ago, and then ".4": four empty cells. With this encoding, a huge ocean with just a few fish or sharks can be stored in a tiny amount of memory. (Note, however, that a shark that just ate cannot be represented together with a shark that hasn't eaten in the last timestep. For a correct encoding, you must separate sharks based on their last feeding time!) If you are familiar with .GIF image files (often encountered on the Web), you might be interested to know that they use run-length encoding to reduce their size.

Your task is to implement a RunLengthEncoding class, which represents a run-length encoding as a linked list of "run" objects. It is up to you whether to use a singly- or doubly-linked list, but a doubly-linked list may make Part IV easier.

For Part II, implement two constructors for RunLengthEncodings. One constructs a run-length encoding of an empty ocean, and the other constructs a run-length encoding based on two arrays provided as parameters to the constructor. These arrays represent the runs that your run-length coding should contain, so you are simply converting arrays to a linked list.

Next, implement the restartRuns and nextRun method. These two methods work together to return all the runs in the run-length encoding, one by one. Each time nextRun() is invoked, it returns a different run--represented as a TypeAndSize object--until every run has been returned. Note that a TypeAndSize object is nothing more than a way to return two integers at once; it is not suitable as a way to represent a run in your run-length encoding, because it does not encode a shark's hunger. The first time nextRun() is invoked, it returns the first run in the encoding, which contains cell (0, 0). After every run has been returned, nextRun() returns null, which lets the calling program know that there are no more runs in the encoding.

The restartRuns() method resets the enumeration, so that nextRun() will once again enumerate all the runs as if nextRun() were being invoked for the first time.

Finally, implement a toOcean method, which converts a run-length encoding to an Ocean object. To accomplish this, you will need to implement a new addShark method in the Ocean class, so that you can specify the hunger of each shark you add to the ocean. This will be important later, so that you can convert an Ocean to a run-length encoding and back again without forgetting how hungry each shark was. Read RunLengthEncoding.java carefully for an explanation of what methods you must write.

The fields of the Ocean class MUST be private, and the RunLengthEncoding class cannot manipulate these fields directly. Hence, the toOcean method will rely upon the Ocean constructor and the addFish and addShark methods.

You cannot change any of the prototypes in RunLengthEncoding.java, nor the file TypeAndSize.java. Again, we will test your code by calling your methods directly.

Part III: Converting an Ocean to a Run-Length Encoding

This part is worth 25% of your total score. (5 points out of 20).

Next, write a RunLengthEncoding constructor that takes an Ocean object as its sole parameter and converts it into a run-length encoding of the Ocean. To accomplish this, you will need to implement a sharkFeeding method in the Ocean class, which tells you how hungry a given shark is. Read Ocean.java and RunLengthEncoding.java carefully for an explanation of what methods you must write.

The fields of the Ocean class MUST be private, so the toOcean method will rely upon the width, height, cellContents, and sharkFeeding methods.

Testing

This is worth 1 point out of the 5, but should probably be done as soon as you can during Part III.

Your RunLengthEncoding implementation is required to have a check() method, which walks through the run-length encoding and checks its validity. Specifically, it should print a warning message if any of the following problems are found:

- If two consecutive runs have exactly the same type of contents. For instance, an "F12" run followed by an "F8" run is illegal, because they should have been consolidated into a single run. (Don't forget, though, that sharks are divided based on how recently they've eaten.)
- If the sum of all run lengths doesn't equal the size (in cells) of the Ocean; i.e. its width times its height.

You may find that the check() method is very useful in helping to debug your RunLengthEncoding constructors and addFish() and addShark() in Part IV.

Part IV: Adding a Fish or Shark to a Run-Length Encoding

The last part is the hardest, but it is only worth 10% of the total score (2 points out of 20), so don't panic if you can't finish it.

Implement the addFish and addShark methods of the RunLengthEncoding class, which are similar to the addFish and addShark methods of the Ocean class. However, this code is much trickier to write. Observe that addFish() and addShark() can lengthen, or even shorten, an existing run-length encoding. To add a shark or fish to a run-length encoded ocean, you will need to find the right run in the linked list, and perhaps break it apart into two or three runs. If the new shark or fish is adjacent to other sharks or fish, you should consolidate runs to keep memory use down. (Your check method ensures that your encoding is as compact as possible.)

Your addFish() and addShark() routines should run in time proportional to the number of runs in the encoding. Hence, you may not convert the run-length encoding to an Ocean object, add the fish or shark to the Ocean, and then convert back to a run-length encoding; this is simply much too slow.

Submitting your Solution

Make sure that your program compiles and runs with the autograding program Test.class, which should be provided a few days after this project is handed out. The autograder will assign your project a score, which forms part of your final project score. Please do not try to fake out the autograder by manually encoding the output it expects; if you do, a human reader will catch you, and we will prosecute cheating harshly.

Change (cd) to your pj1 directory, which should contain Ocean.java, RunLengthEncoding.java, and any other .java files you wish to submit. If you have any comments you would like to make to the grader, such as parts of your solution that you know are not correct, create a file called GRADER and put your comments there. In either GRADER or a comment in Ocean.java, include your name, login, and lab section. From your pj1 directory, type "submit pj1".

After submitting, if you realize your solution is flawed, you may fix it and submit again. You may submit as often as you like. Only the last version you submit will be graded, unless you inform your reader that you would prefer to have an earlier submission graded instead. If your submission is late, you will lose 1% of your earned score for every two hours (rounded up) your project is late.