Programming Lab 4: Darwin

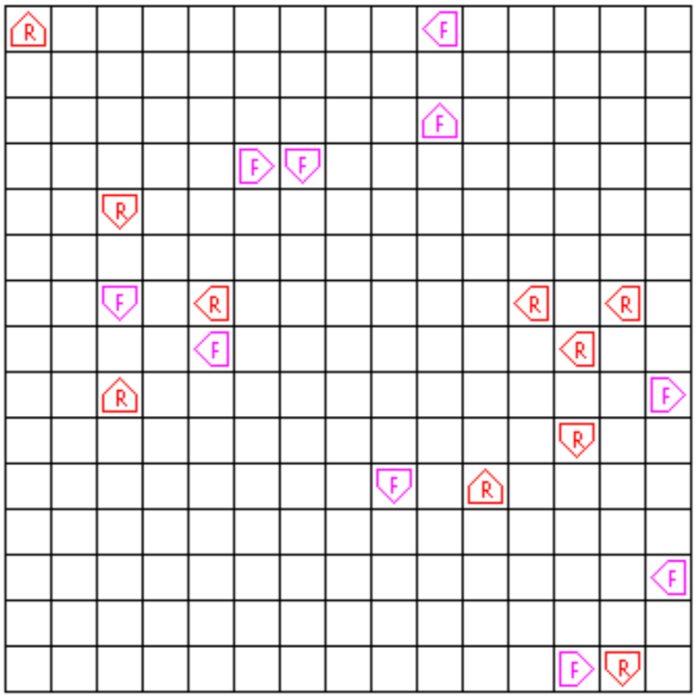
Data Structures and Advanced Programming

Objectives

- To give you more practice writing large, multi-class programs.
- To illustrate the importance of modular decomposition and information hiding. The entire program is broken down into a series of classes that can be developed and tested independently, without revealing representational details.
- To have fun with a problem that is algorithmically interesting in its own right.

Introduction and Credits

In this assignment, your job is to build a simulator for a game called Darwin invented by Nick Parlante. Much of this handout has been borrowed from a version that Stephen Freund created when he adapted the assignment for use at Williams College. This lab was originally developed by Nick Parlante, Eric Roberts, and Bob Plummer for use at Stanford University. Morgan McGuire has further updated the game to include 3D graphics and several other features. For a variety of reasons (related to ease of setup and differing emphasis), we will be using Steve's version of the assignment. If you are curious about Darwin 2.0 and want to explore it on your own, here is a link to Morgan's version: `http://cs.williams.edu/~morgan/cs136-s08/darwin2.0/`.



Lab

This is a larger program than our previous labs, and you will have two weeks to complete it. As part of the lab, you will develop a “creature" that lives in the Darwin world. In Darwin, creatures compete with one another to survive. After the project is due, we will run a tournament to see whose creature is most fit for life in the cruel world of a 2D grid.

The Darwin World

The Darwin program simulates a two-dimensional world divided up into small squares and populated by a number of creatures. Each of the creatures lives in one of the squares, faces in one of the major compass directions (North, East, South, or West) and belongs to a particular species, which determines how that creature behaves. For example, one possible world is shown on the previous page.

That sample world is populated with twenty creatures, ten of a species called Flytrap and ten of a species called Rover. In each case, the creature is identified in the graphics world with the first letter in its name. The orientation is indicated by the figure surrounding the identifying letter; the creature points in the direction of the arrow. The behavior of each creature –which you can think of as a small robot – is controlled by a program that is particular to each species. Thus, all of the Rovers behave in the same way, as do all of the Flytraps, but the behavior of each species is different from the other.

As the simulation proceeds, every creature gets a turn. On its turn, a creature executes a short piece of its program in which it may look in front of itself to see what’s there and then take some action. The possible actions are moving forward, turning left or right, or infecting some other creature standing immediately in front, which transforms that creature into a member of the infecting species. As soon as one of these actions is completed, the turn for that creature ends, and some other creature gets its turn. When every creature has had a turn, the process begins all over again with each creature taking a second turn, and so on. The goal of the game is to infect as many creatures as possible to increase the population of your own species.

Species Programming

In order to know what to do on any particular turn, a creature executes some number of instructions in an internal program specific to its species. For example, the program for the Flytrap species is shown below:

step	instruction	comment
1	ifenemy	4
2	left	Turn left
3	go 1	Go back to step 1
4	infect	Infect the adjacent creature
5	go 1	Go back to step 1

The step numbers are not part of the actual program, but are included here to make it easier to understand the program. On its turn, a Flytrap first checks to see if it is facing an enemy creature in the adjacent square. If so, the program jumps ahead to step 4 and infects the hapless creature that happened to be there. If not, the program instead goes on to step 2, in which it simply turns left. In either case, the next instruction is a go instruction that will cause the program to start over again at the beginning of the program.

Programs are executed beginning with the instruction in step 1 and ordinarily continue with each new instruction in sequence, although this order can be changed by certain instructions in the program. Each creature is responsible for remembering the number of the next step to be executed. The instructions that can be part of a Darwin program are listed below:

hop: The creature moves forward as long as the square it is facing is empty. If moving forward would put the creature outside the boundaries of the world or would cause it to land on top of another creature, the hop instruction does nothing.

`left`: The creature turns left 90 degrees to face in a new direction.

`right`: The creature turns right 90 degrees.

`infect` n: If the square immediately in front of this creature is occupied by a creature of a different species (an “enemy") that creature is infected to become the same as the infecting species. When a creature is infected, it keeps its position and orientation, but changes its internal species indicator and begins executing the same program as the infecting creature, starting at step n of the program. The number n is optional. If it is missing, the infected creature should start at step 1.

`ifempty` n: If the square in front of the creature is unoccupied, update the next instruction field in the creature so that the program continues from step n. If that square is occupied or outside the world boundary, go on with the next instruction in sequence.

`ifwall` n: If the creature is facing the border of the world (which we imagine as consisting of a huge wall) jump to step n; otherwise, go on with the next instruction in sequence.

`ifsame` n: If the square the creature is facing is occupied by a creature of the same species, jump to step n; otherwise, go on with the next instruction.

`ifenemy n`: If the square the creature is facing is occupied by a creature of an enemy species, jump to step n; otherwise, go on with the next instruction.

`ifrandom n`: In order to make it possible to write some creatures capable of exercising what might be called the rudiments of “free will," this instruction jumps to step n half the time and continues with the next instruction the other half of the time.

`go n`: This instruction always jumps to step n, independent of any condition.

A creature can execute any number of `if` or `go` instructions without relinquishing its turn. The turn ends only when the program executes one of the instructions: `hop`, `left`, `right`, or `infect`. On subsequent turns, the program starts up from the point in the program at which it ended its previous turn. The program for each species is stored in a file in the subfolder named Creatures in the assignment folder. Each file in that folder consists of the species name and color, followed by the steps in the species program, in order. The program ends with a blank line or a line containing only “.". Comments may appear after the blank line or at the end of each instruction line. For example, the program file for the Flytrap creature looks like this:

```
Flytrap
magenta
ifenemy 4
left
go 1
infect
go 1
.
The flytrap sits in one place and spins.
It infects anything which comes in front.
Flytraps do well when they clump.
```

There are several presupplied creature files:

`Food`: This creature spins in a square but never infects anything. Its only purpose is to serve as food for other creatures. As Nick Parlante explains, “the life of the Food creature is so boring that its only hope in life is to be eaten by something else so that it gets reincarnated as something more interesting."

`Hop`: This creature just keeps hopping forward until it reaches a wall. Not very interesting, but it is useful to see if your program is working.

`Flytrap`: This creature spins in one square, infecting any enemy creature it sees.

`Rover`: This creature walks in straight lines until it is blocked, infecting any enemy creature it sees. If it can’t move forward, it turns.

You can create your own creatures by adding a data file to the Creatures directory in the format described above.

Your Assignment

Your mission is to write the Darwin simulator. The program is large enough that it is broken down into a number of separate classes that work together to solve the complete problem. You are responsible for implementing the following classes:

> `Darwin`: This class contains the main program, which is responsible for setting up the world, populating it with creatures, and running the main loop of the simulation that gives each creature a turn. The details of these operations are generally handled by the other modules. New creatures should be created in random empty locations, pointing in random directions.
>
> `Species`: This class represents a species, and provides operations for reading in a species description from a file and for working with the programs that each creature executes.
>
> `Creature`: Objects of this class represent individual creatures, along with operations for creating new creatures and for taking a turn.
>
> `World`: This class contains an abstraction for a two-dimensional world, into which you can place the creatures.

Skeletons of these classes are provided in the starter folder. You should not need to add any additional public methods to these classes (although you may if you think it improves the design). You will, however, probably want to add additional protected methods as you implement the classes. In addition, the starter code contains three helper classes that you should use **without modification**:

- `Instruction`: This simple class represents one instruction out of the Species's instruction set.
- `Position`: This class represents (x,y) points in the world and constants for compass directions. This class is similar to the `Cell` class we used in the Game of Life lab.
- `WorldMap`: This class handles all of the graphics for the simulation.

Download the starter code for this lab from GitHub and familiarize yourself with the classes before you begin.

Strategy

Here is a suggested course of action to implement Darwin:

1. Pull the latest version of the `datastructures` repository from GitHub and make sure that you have the `DarwinStarter` folder.

2. The starter code contains a JAR will with a complete version of the simulator. Run it from within the DarwinStarter directory with the following command:

```
java -jar darwin.jar Flytrap.txt Rover.txt
```

   In previous labs, you have been able to test the classes you were developing against complete versions contained within the starter code JAR files. For this lab, the code contained within the JAR file has been obfuscated to prevent you from doing that. Instead, you should test your code as you go along through simple `main` methods that you write within the classes you develop. More on that below.

3. Write the `World` class. This should be straight-forward if you use a Matrix object or a 2-dimensional array to represent the world. **Test this class thoroughly before proceeding. Write a main method in the `World` class and verify that all of the methods work.**

4. Write the `Species` class. The first step will be parsing the program file and storing it in the Species. Note that the first instruction of a program is at address 1, not 0. **Test this class thoroughly before proceeding. Write a main method in the `Species` class and verify that all of the methods work.**

5. Fill in the *basic* details of Creature class. Implement only enough to create creatures and have them display themselves on the world map. Implement `takeOneTurn` for the simple instructions (`left`, `right`, `go`, `hop`). **Test the basic `Creature` thoroughly before proceeding. Write a main method in that class and verify that all of the methods work.**

6. Begin to implement the simulator in the `Darwin` class. Start by reading a single species and creating one creature of that species. Write a loop that lets the single creature take 10 or 20 turns.

7. Go back to `Creature` and implement more of the `takeOneTurn` method. Test as you go – implement an instruction or two, and verify that a `Creature` will behave correctly, using your partially written `Darwin` class.

8. Finish up the `Darwin` class. Populate the board with creatures of different species and make your main simulation loop iterate over the creatures giving each a turn. The class should create creatures for the species given as command line arguments to the program when you run it. See `Darwin.java` for more details. Run the simulation for several hundred iterations or so. You can always stop the program by pressing `control-C` in the terminal window or closing the Darwin Window.

9. Finally, finish testing the implementation by making sure that the creatures interact with each other correctly. Test `ifenemy`, `infect`, etc.

Possible Extensions

There are many ways to extend the program to simulate more interesting Species behavior. Here are just a few ideas if you wish to extend Darwin for extra credit:

1. Give creatures better eyesight.

   - Add `if2enemy n`. This instruction checks if there is an enemy two steps in front of a creature. This can help make fly traps much more lethal.
   - Add `ifenemyleft n` and `ifenemyright n`. These variations on the `ifenemy` instruction can check if there is an enemy to either side.

   You can add similar versions of the other tests too.

2. Give creatures memory. This can be as simple as permitting each creature to store a single integer. You can then add the following instructions to the instruction set:

   - `set n` to set a creature's memory;
   - `ifeq v n` to jump to address n in the program if a creature's memory contains v; and
   - `inc` and `dec` to add and subtract from memory.

   You can get more coordinated activity by using this type of memory.

3. Give creatures the ability to communicate. Once creatures have memory, let them ask the creature on the square in front of them what is on its mind with `ifmemeq v n`. This instruction reads the memory of the creature in the square in front of a creature and jumps to n if that value is v. You can also add a `copymem` instruction that copies the value in the memory value of the creature in front of you to your own memory. These instructions permit creatures to form quite successful "phalanx" formations.

4. Make creatures mutate. Perhaps copies of creatures aren't quite the same as originals – when a creature infects another creature, make there be a chance that the infected creature will be a mutation of the infecting creature's species. This will require creating new `Species` that are close, but not quite exact, copies of an existing `Species`. Taken to its extreme, you can make species evolve over time to become better and better at surviving in the Darwin world. This type of genetic algorithm is fascinating to watch in practice – come talk to me about this one if you want to try it.

Of course, you are free to implement any other extensions you find interesting. There are lots of possibilities? be creative!

Thought Questions

Please answer the following questions in a comment at the top of your `Darwin.java` file:

1. In competitions between Flytraps and Rovers, one species usually eliminates the other. The species that proves victorious varies from contest to contest, however. Suppose we wanted to write a program to determine which species wins more often. How would you go about writing such a program? Would it be guaranteed to produce an answer?

2. Generally speaking, programming languages are either *compiled* or they are *interpreted*. In compiled languages, one runs an application to translate source code into machine code. Then, the generated code runs directly on the target system. By contrast, interpreted languages are run within another computer program (called an *interpreter*) that reads in the code and knows how to execute it. Our simulator is an interpreter for the Creature programming language. Java is a compiled language (`javac` stands for “Java compiler"), but the java compiler targets an abstract machine known as the Java Virtual Machine (or JVM for short). Specifically, the Java compiler compiles Java source code into Java bytecode. Early versions of the JVM were straight interpreters of Java bytecode. Modern versions of the JVM now include a *just-in-time compiler* (or JIT for short). Research the concept of a just-in-time compiler. Then, explain in your own words what a JIT does and how one works.

3. The programming language that we've implement for creatures is similar to low-level assembly programming languages. In particular, branching in our code is handled by jumping to a specific line number using the `go` and `ifX` commands. C and C++ are two more examples of programming languages that have a `goto` command to support this type of branching. In 1968, computing pioneer Edsger Dijkstra famously published a letter entitled “Go To Statement Considered Harmful." In that paper, he writes, “The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program." Given your experience writing Creature code, do you agree with him? Justify your answer.

Submission

You will submit this project in two phases:

1. You must turn in preliminary versions of `World.java` and `Species.java` by the “Check Point 1" deadline that's posted on Canvas. You should test these classes by themselves and provide tests to demonstrate that they work properly.

2. You must turn in the following five files by the start of class on the project's ultimate due date:

   - Final version of `World.java`
   - Final version of `Species.java`
   - `Creature.java`
   - `Darwin.java`
   - A Species of your own design. It can be as simple or as complex as you like. We will pit your creatures against each other to watch them battle for survival. Fabulous door prizes

will be awarded. We will run all simulations on a 15x15 grid populated with 10 creatures from each of 4 species.

As ever, you should submit your work on GitHub and commit your work regularly.