In this assignment, your job is to build a simulator for a game called Darwin invented by Nick Parlante.

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 of it, 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:

<table>
<thead>
<tr>
<th>step</th>
<th>instruction</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ifenemy 4</td>
<td>If there is an enemy ahead, go to step 4</td>
</tr>
<tr>
<td>2</td>
<td>left</td>
<td>Turn left</td>
</tr>
<tr>
<td>3</td>
<td>go 1</td>
<td>Go back to step 1</td>
</tr>
<tr>
<td>4</td>
<td>infect</td>
<td>Infect the adjacent creature</td>
</tr>
<tr>
<td>5</td>
<td>go 1</td>
<td>Go back to step 1</td>
</tr>
</tbody>
</table>

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 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 also create your own creatures by creating a data file 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. 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, we provide you with 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. These are similar to what we used in the Maze solving program.

**WorldMap**: This class handles all of the graphics for the simulation.
Documentation for these classes will be provided on the web page. Familiarize yourself with the classes before you begin.

**Strategy.** Here is a suggested course of action to implement Darwin:

1. Copy the starter files with the command
   
   ```
   cp -r /usr/mac-cs-local/share/cs136/labs/darwin ./darwin
   ```

2. You can use the command `darwin` in the Mac lab to run a sample solution. (This may not work until lab on Wednesday.) This will give you a chance to see how the program is supposed to behave. Run it with a command line like
   
   ```
   dawrin Hop.txt Rover.txt
   ```
   
   while inside the directory that has Creatures as a subdirectory.

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.

You must turn in a species of your own design by 11:59pm on Tuesday, May 11th. 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 during class. We will run all simulations on a 15x15 grid populated with 10 creatures from each of 4 species. **If you choose to implement any extra instructions, the creature that you submit for the tournament must only use standard instructions.**

**Submission.** Submit the following files via turnin:

- `World.java`
- `Species.java`
- `Creature.java`
- `Darwin.java`