In this assignment, your job is to build a simulator for a game called *Darwin* invented by Nick Parlante at Standford. The assignment has a four-fold purpose:

1. To give you a chance to write a large multi-module program.

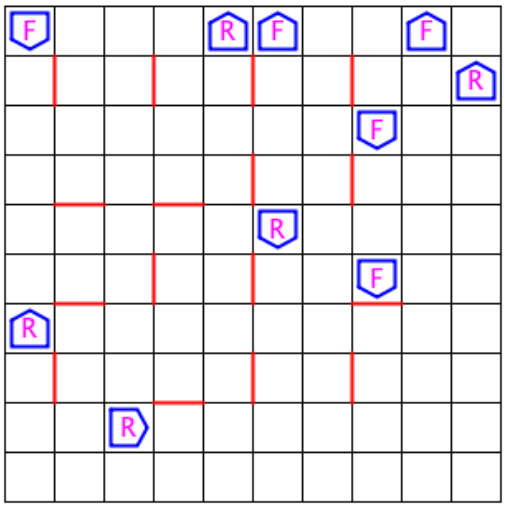
2. To illustrate the importance of modular decomposition. The entire program is broken down into a series of modules that can be developed and tested independently.

3. To stress the notion of ADTs as a mechanism for sharing data between modules without revealing the representational details.

4. To let you have fun with an application that is captivating and algorithmically interesting in its own right.

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. Walls are randomly placed to separate some cells. For example, one possible configuration of the world is shown below:



This sample world is populated with ten creatures, five of a species called *Flytrap* and five 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 arrow 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
0	IFENEMY 3	*If there is an enemy ahead, go to step 3*
1	LEFT	*4 Turn left*
2	GOTO 0	*Go back to step 1*
3	INFECT	*Infect the creature in the cell I am facing*
4	GOTO 0	*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 **`GOTO`** 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 0 and ordinarily continue with each new instruction in sequence, although this order can be changed by certain instructions in the program (such as IFENEMY). 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, however the creatures turn still ends.
`LEFT`	The creature turns left 90 degrees to face in a new direction.
`RIGHT`	The creature turns right 90 degrees.
`INFECT`	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. If a creature tries to infect an empty square or another creature of its same species, nothing happens but the creature's turn ends. 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 0.
`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.
`GOTO` *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 that consists of the species name on the first line, the number of total instructions on the next line, and starting on the 3rd line, the steps in the species program, in order, one per line. The function to read a file and create a list of instructions is given to you in the BaseSpecies class.

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 (especially when testing walls).

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

`Rover`
: This creature wanders around, randomly moving forward or turning, infecting any enemy creature it sees. If it is blocked by a wall or other creature, it turns.

`LandMine`
: This creature just turns so it's not facing a wall or another of its own kind, and then keeps trying to infect the cell in front of it, whether or not another creature is present.

You can also create your own creatures. In particular, you can design a creature for the Darwin Contest I will hold at the end of the project (more info on that later).

Your assignment

Your mission in this assignment is to write the Darwin simulator and get it running. Since this is a large program, it is a more challenging task than any of the ones you have faced to date. The program is broken down into several separate modules that work together to solve the complete problem. Part of the reason for this is to experience a bit of what programming can be like in the real world. While you're going to program 4 modules for this assignment, it would not be uncommon to have 4 separate people or groups coding each of these 4 modules concurrently. This means you have to strictly follow the ADT/design given to you for the other 3 modules, as making any changes would require the other 3 groups to change things as well.

To make this process simpler, I have given you all 4 modules already completed in compiled form (.class files), so you can test any 1 of your modules with the other 3 of my modules to make sure it works properly and follows the ADT. Please don't try to do anything like reverse engineer the source code, just use the .class files without opening them.

Of the modules, you are responsible for the following four, for each you will create a new

class that inherits from the given abstract base class:

`Creature - inherits from BaseCreature`
: This module defines a data type representing an individual creature. It stores the Creature's Species, it's current location on the world grid, the direction it's currently facing, the instruction from its Species program it should execute next, and the World object it lives in so it can interact with and affect that world. It implements several helper functions and a function to take one turn of a creature.

`Species - inherits from BaseSpecies`
: This module defines a data type representing a species, and provides operations for reading in a species description from a file and for working with the programs that each creature executes.

`World - inherits from BaseWorld`
: This module represents the actual Darwin world. It holds the worldGraph and worldMap instances, a random number generator, and potentially other objects. Implements some helper functions and a function to add the random walls in the world.

`WorldGraph - inherits from BaseWorldGraph`
: The WorldGraph class creates the actual "physical" world as a graph where each cell in the theoretical world grid is represented as a Vertex and edges between vertices represent a connected path that may be followed by Creatures from one to another. Creatures can only move in the 4 cardinal directions north, south, east, and west (or up, down, left, and right), they may not move diagonally. They also may not move through walls, either the external walls of the grid or the internal walls that are added. Therefore if there is a wall between 2 cells of the grid there should NOT be an edge between the 2 corresponding vertices.

Each of the abstract base classes given to you corresponding to the 4 modules above have comments for each function as to what it should do.

The following modules have been provided for you:

`Darwin`
: This module contains the main program, which is responsible for reading the config file, dealing with command line arguments, 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.

`WorldMap`
: This module handles all of the graphics for the simulation. Uses `Draw` class to do the actual drawing to the screen.

I have also provided you with 2 helper classes, `Point` and `Instruction`, and 2 enum

classes, `Direction` and `Operation`. For more on enums see the comments in the `Direction` enum.

Even though the program is big, the good news is that you do not have to start completely from scratch. In fact, you have the advantage of being able to start with a complete program that solves the entire assignment. This means that you get to play with a working Darwin program immediately beginning on day 1.

Important Notes (or things learned from last term):

- If a data field is already present in one of the base classes that you inherit from, DO NOT redefine it in your own class. This causes all sorts of problems and lots of your favorite null pointer exceptions. These fields are all declared as "protected" in the base classes, so you can just use them as if they were defined in your own class already.
- Write your modules in the order WorldGraph, World, Species, Creature. The first 3 you could switch around but you should definitely code Creature last, after you understand what's going on in the other 3.
- USE the functions I've given you! Many of them you have to actually do the implementing, but if I put an abstract method in a base class it must be because I think you may need it somewhere. For example in WorldGraph you will have convenient methods like inRange and adjPoint, or in the Direction enum you have getLeft() and getRight(). Don't reinvent the wheel, there's enough for you to do already =)
- TEST each of your modules with my versions of the other 3. If it doesn't work with my other modules, you will lose points, this is part of the purpose of the assignment.
- One thing that confused many last term was the interaction between a creature and species in terms of knowing what instruction to execute next. The Species has the list of instructions in order for that species, however each Creature of that species may be currently on a different instruction. What the Creature stores therefore, is an index k that represents where its current instruction is in the Species list of instructions. The Species class has the method getInst(int k) that simply returns the kth instruction from its list of instructions, so you can easily call that method from a given Creature to see what instruction its on. This also means that if you're executing one of the "if" instructions and its true, you just update the Creature's k to be whatever argument came with that instruction.
- The takeOneTurn() method in the Creature class is by far the most complex thing you will do in the assignment. You're basically writing a mini-interpreter for the simple instruction set a Darwin Creature is allowed to draw from. I made mine with a switch statement (here's an example of when they're convenient) and recursion. You could easily use if/else statements and a while loop. Part of your grade for this assignment as usual is coding standard, which means nice readable code among other things. Maybe it would make sense to break this very long function up into several smaller ones?

I provide you with the following files:

Abstract ADTs

```
BaseSpecies.java
BaseCreature.java
BaseWorld.java
BaseWorldGraph.java
```

Compiled complete files

```
Species.class
Creature.class (and Creature$1.class)
World.class
WorldGraph.class (and WorldGraph$Vertex.class)
```

Source code

```
Darwin.java
WorldMap.java
Draw.java
Point.java
Instruction.java
Direction.java
Operation.java
```

To get started, you can use these files to build a complete project for the entire Darwin system. Just type `javac Darwin.java` and the compiler will know to use the .class files for the modules that the actual source code hasn't been written yet. Try typing `javac Darwin.java -h` to see a list of all of the command line arguments you can use to change the details of the simulation environment. Of course as soon as you add a file such as Creature.java of your own, when you compile Darwin.java it will compile that file and overwrite the original Creature.class file, so you might want to **save your original download in a separate folder than the one you work in**. I can see on Moodle how many times something is downloaded and this assignment was downloaded something like 5 times per person on average last term, I'm just trying to save you the hassle =)

Your job is to implement the modules given above that you are responsible for.
When you have written your own implementation of any of these modules, you can remove the `.class` file from the project and add your `.java` file in its place. The ADTs have all been given to you in the abstract base classes; your job is to write a new implementation for each of these four modules that implements the functions that are part of that ADT description.

What to Turn In

As always, you should create a zip folder with your carleton username(s) containing **everything your program needs to run** (including the files you did not modify). Make sure you comment your files well. Good luck and have fun!!!