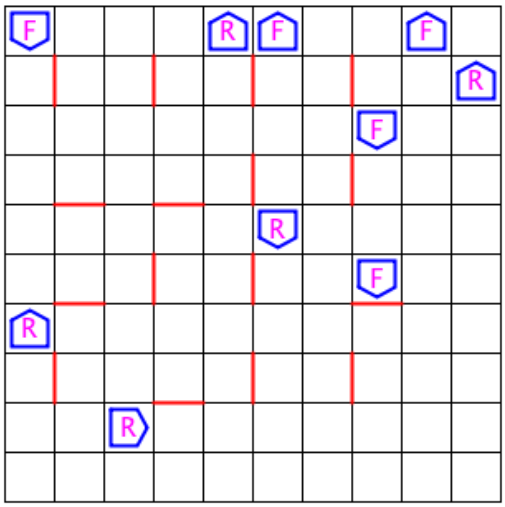
Overview

In this assignment, your job is to build a simulator for a game called *Darwin* invented by Nick Parlante at Standford. The goal is to practice implementing graph methods, as well as gaining experience in working with a larger software project consisting of many files. In this context many good coding principles become vitally important, such as good use of inheritance, code organization through separate classes and/or functions, modular testing of one piece of code at a time, etc. Plus hopefully it's fun to work with an application that is interesting both algorithmically and in design.

The Darwin world

The Darwin program simulates a two-dimensional world divided into a grid of cells that is populated by a number of *creatures*. Each creature occupies one single cell, faces North, East, South, or West, and belongs to a particular *species,* which determines how that creature behaves in general. Walls are randomly placed in the grid world to make creature movement more complicated/interesting.

For example, one possible state of a world at a particular moment is shown below:



This sample world is a 10x10 grid populated with ten creatures, five of a species called *Food* and five of a species called *Rover*. The red lines indicate walls that creatures may not move directly through (the edges of the world are also blocking walls). The graphics displayed for each cell that currently contains a creature include the first letter of the creature's species name, and an arrow surrounding that letter to indicate the current direction the creature is facing.

The behavior of each creature is controlled by its species algorithm specified as a list of instructions in the file *species_name.txt*. Thus, all of the Rover creatures in the above world act by following a single algorithm, though each individual creature may be at a different place in that algorithm's code. All of the Food creatures in the above world similarly follow a single algorithm, but a different one from the Rovers, specifically the algorithm in the file *Food.txt*.

The simulation proceeds in discrete steps, where each step consists of allowing every creature on the world grid to perform a single action (the order in which the creatures act is implementation-specific). The possible actions an individual creature may take are moving forward, turning left or right, or *infecting* another creature of a different species that is in the cell immediately in front of the acting creature. The *INFECT* action (if successful) transforms the infected creature into a new creature of the same species as the one doing the infecting. The goal of the game for each species type is to infect creatures of other species in order to become the dominant species overall in the world. The game ends in a complete victory if a single species infects every creature in the current world, or if a set number of steps are completed without a single species taking over, the species with the most creatures at the end of the simulation is considered the winner.

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 a species named *Flytrap* is shown below:

step	instruction	comment
0	IFENEMY 3	*If there is an enemy ahead, go to step 3*
1	LEFT	*Turn left*
2	GOTO 0	*Go back to the first instruction*
3	INFECT	*Infect the creature in the cell I am facing*
4	GOTO 0	*Go back to the first instruction*

The step numbers are not part of the actual program, but are included here to make it easier to understand the algorithm behavior. On its turn, a *Flytrap* first checks to see if it is facing an enemy creature (any creature of a different species) in the next cell. If so, the program jumps ahead to step 3 and infects that hapless creature. If not, the program instead goes on to simply turn left. In either case, the next instruction (goto 0) will cause the program to loop back to the first instruction and run again.

Programs are executed sequentially except when a goto or if-X instruction triggers a jump to a specific line in the program. NOTE: **Each individual creature is responsible for tracking its current location in the general species algorithm** (i.e. which step # it should execute next).

The valid instructions for a Darwin species algorithm are listed below. The first 4 represent *terminal actions*; as soon as any one of these is executed, the current creature's turn is over and another creature will get to act. The last 6 instructions represent decision-making based on the current state of the creature's environment; a creature may execute as many of these as it wants in a single turn in order to choose one of the 4 *terminal actions*. Each individual creature starts its turn at the point in the program at which it ended its previous turn (this is why each creature needs to track where it is in the overall species algorithm at the end of teach turn).

`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.

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 instructions in the species program, in order, one per line. The function to read a file and create a list of instructions for a species is given to you in the BaseSpecies class.

There are a 4 creature files provided, each of which specifies a very simple creature program, as described below:

`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 moves in the direction it is facing until something appears in front of it to block its way. If the "something" is a creature of another species the Rover will infect, otherwise the Rover do an about face by turning twice and then continue in the opposite direction

You should also create your own more interesting creatures! In particular, you should design a creature for the contest to be held during our final exam slot (more info on that later).

Your assignment

Your main task is to program the Darwin World simulator. 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 only 2 of the 4 main 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 must implement Creature and WorldGraph (see below for detailed descriptions); for each you will create a new class from scratch that inherits from the abstract base class with the same name. I have provided completed code for the 2 modules Species and World, so you can see examples of implementing a class that

inherits from a base abstract class (abstract just means you can't actually create an instance of the class, though you can create an instance of a class that inherits from the abstract class with the abstract class type, e.g.

BaseSpecies mySpec = new Species("Flytrap.txt");

More info on the classes you must create:

`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.

`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.

The 2 provided abstract base classes have relatively detailed comments for each function that you need to override in your inheriting class.

The following classes have also 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.

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 in the sense that it includes many files, you only need to implement a few distinct parts. Also you have the advantage of being able to immediately run a complete program that solves the entire assignment, and thus can test each of your 2 classes individually using my .class file for the other module.

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 WorldGraph module first. Then make sure you completely understand how WorldGraph, World, and Species work and interact before you begin writing your Creature module.
- 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! Part of your grade will be good software design.
- 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 in the past 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 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

```
Species.java
World.java
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 =)

What to Turn In

1. due by 3pm Sunday 3/13: one or two creature files containing your best creature design(s) that you wish to compete in the contest. You must submit at least 1 creature, and you may submit 2 if you can't decide which is your favorite (as usual put them both in a single folder and submit just one file with the compressed/zipped folder. You may not submit more than 2 creatures.
2. Due by 11:59pm Monday 3/14: A zip folder named with BOTH partners usernames (e.g. sgoings_wedinb) containing **everything your program needs to run** (including the files you did not modify). Make sure you comment your files well.

*** no exceptions, submissions after this time must be approved by a college dean!

Good luck and have fun!!!