Lab 9: Darwin[1]
CSCI 2101 – Fall 2017

Due: Part 1: Friday, Dec 1, 11:59 pm, Part 2: Wednesday, Dec 6, 11:59 pm
Collaboration Policy: Level 1
Group Policy: Pair-optional (with individual creature submissions)

Near the start of the semester, we talked about the WaTor World as an example of an environment that could naturally be modeled using objects. Now, near the end of the semester, and using your experience writing object-oriented programs over the course of the semester, you will tackle a simulator for a similar type of game: **Darwin**! Building your simulator will give you more experience writing large, multi-class programs and will demonstrate the power of modular decomposition and information hiding (and will, of course, be fun!).

1 The Darwin World

The world of Darwin is a two-dimensional grid of squares, populated by creatures (similar to the WaTor World). The boundaries of the world are impassable, and may be thought of as walls. Each square is either empty or populated by one creature, which is oriented in a particular direction (North, South, East, or West). Each creature is of a particular species, which determines how the creature behaves. More precisely, we can say that each species specifies a particular behavior "program", and each creature independently executes its behavior program (as determined by its species) as it interacts with the world.

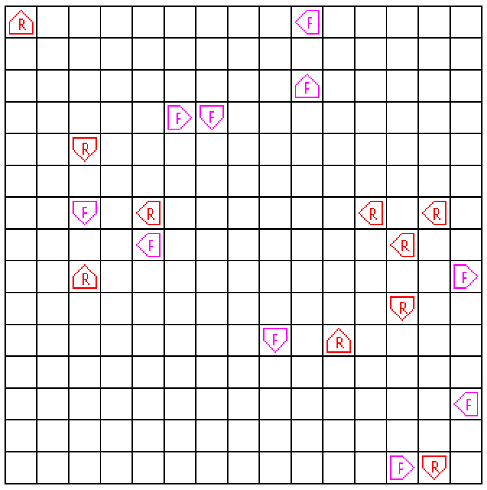

Figure 1: An example Darwin world.

Consider the example Darwin world shown in Figure 1. This world is populated by twenty creatures, half of which are of the Flytrap species and half of which are of the Rover species. The species of each creature is indicated by a color (magenta for Flytrap and red for Rover) and a character (F for Flytrap and R for Rover). Note that each creature has a particular orientation, indicating its current direction.

[1]Adapted from a lab originally developed by Nick Parlante

The simulation can be thought of as a series of rounds. During each round, every creature gets a "turn". During a creature's turn, the creature executes steps of its behavior program, which will ultimately result in one of the following actions:

1. The creature moves forward one space (according to its orientation).
2. The creature turns left.
3. The creature turns right.
4. The creature "infects" the creature directly in front of it, transforming the hapless victim into a creature of the infecting species.

Note that an action is *not* the same as a step – multiple steps of the behavior program may be executed in a single turn prior to taking one of the four actions listed above. More details on the steps of the behavior program are given below.

Once a creature has taken an action, its turn is over and the next creature in the round begins its turn. Once all creatures have had a turn, the next simulation round begins. The goal of the game, of course, is natural selection – every species attempts to infect as many creatures as possible to increase the population of the species. Note that unlike in the WaTor World, creatures in Darwin never actually die or reproduce – they just change species over time via infections.

2 Species Programming

The behavior program of each species is made up of a series of steps, or *instructions*. As an example, the program for the Flytrap species is shown in Table 1, where each step of the program is indicated by an *address*.

address	instruction	comment
0	`start:`	A named "label" marking a specific step in the program
1	`ifenemy doinfect`	If there is an enemy ahead, jump to label "doinfect"
2	`left`	Turn left
3	`go start`	Jump back to label "start"
4	`doinfect:`	Another label
5	`infect`	Infect the adjacent creature
6	`go start`	Jump back to label "start"

Table 1: The behavior program of the Flytrap species.

The addresses are not part of the actual program, but indicate the ordering of the steps. To summarize this behavior program, when a Flytrap takes its first turn, it checks to see if it is facing an enemy creature in the adjacent square. If so, the program jumps ahead to the "doinfect" label (address 4) and infects the creature that is there. If not, the program instead proceeds to address 2 and simply turns left. In either case, the flytrap has executed an action, and thus its turn ends. When its next turn begins, it picks up right where it left off in the program – in this particular program, the program then jumps back to the "start" label (either from address 3 or address 6) and continues from the beginning. Thus, while this particular program exhibits the same behavior on every turn, this need not be true of all behavior programs.

Note that the instructions of a behavior program resemble a rudimentary programming language, in which flow control is accomplished not via conditionals and loops but instead via simple "jump-to-address" style instructions[2].

2.1 Instruction Listing

Some instructions consist only of the instruction name (e.g., `hop`), while others take an argument interpreted as the name of a program label (e.g., `go`). A complete listing of the valid instructions in a behavior program are given below.

`hop`: The creature moves forward one square and ends its turn, assuming that the target square is unoccupied. If the creature is facing a wall (i.e., an edge of the world) or another creature, the hop instruction does nothing (but still ends the creature's turn).

`left`: The creature turns left 90 degrees and ends its turn.

`right`: The creature turns right 90 degrees and ends its turn.

`infect label`: Infects the creature immediately in front of this creature and ends this creature's turn. When a creature is infected, it keeps its position and orientation, but changes its species and begins executing the same program as the infecting creature, starting at the specified label of the program. The `label` argument is optional – if it is omitted, the infected creature starts at address 0 (the start of the program). Infection has no effect on a creature that is already the same species as the infecting creature, or if there is no creature immediately in front of this creature. In all cases, executing the `infect` instruction ends the creature's turn.

`ifempty label`: If the square in front of the creature is unoccupied, continues execution of the program from the address indicated by `label`. If the forward square is occupied by a creature or is outside the world boundary, does nothing.

`ifwall label`: If the creature is facing the border of the world (which we imagine as consisting of a huge wall), continues execution of the program from the address indicated by `label`. Otherwise, does nothing.

`ifsame label`: If the creature is facing a creature of the same species, continues execution of the program from the address indicated by `label`. Otherwise, does nothing.

`ifenemy label`: If the creature is facing a creature of a different (i.e., enemy) species, continues execution of the program from the address indicated by `label`. Otherwise, does nothing.

`ifrandom label`: With equal probability, either continues execution of the program from the address indicated by `label`, or does nothing. This decision is made independently each time the instruction is executed (i.e., each time there is a 50% chance of jumping).

`go label`: Continues execution of the program from the address indicated by `label` (i.e., an unconditional jump).

[2]This mechanism is actually quite close to how computers execute all programs...take CSCI 2330 to learn more!

`label:` A label in the program, serving only as a target address for other instructions. Executing a label instruction has no effect. The name of a label instruction may be any alphanumeric word, as long as it is not one of the other instruction names or the name of an existing label in the program. **Note that a label instruction must include the colon after the label name (e.g., `mylabel:`)** – this is what designates the instruction as a label! The label instruction is the *only* instruction that includes any non-alphanumeric characters.

2.2 Program Files

A behavior program is stored in a plain text file (e.g., `flytrap.txt`). In addition to the actual instructions of the behavior program, the program file contains a few extra pieces of data. In particular:

- The first line of the file must contain the name of the species (e.g., `Flytrap`).
- Any line that begins in `#` is considered a comment and is ignored.
- Blank lines in the file are ignored and may appear anywhere (except for the very first line).

Shown below is the full contents of the program file for the Flytrap species, which is stored in the file `flytrap.txt`:

```
Flytrap

# The flytrap sits in one place and spins.
# It infects anything which appears in front.
# Flytraps do well when they clump.

start:
ifenemy doinfect
left
go start

doinfect:
infect
go start
```

2.3 Sample Species

There are several pre-supplied species files, which will be useful for testing:

`Food:` This creature spins but never hops or 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 encounters. If it can't move forward, it randomly turns left or right.

`Medusa`: This creature is a modified rover that turns enemies to stone (i.e., they freeze in place after infection). Useful for testing that the infect instruction works properly.

Of course, you also can (and will!) create your own species by writing behavior program files in the format described above.

3 The Darwin Simulator

Your task is to complete a program that can execute rounds of a Darwin simulation. The program is large enough that it is broken down into a number of separate classes that work together to provide the full simulation. The most significant components of the simulator that you will need to complete are (1) code to represent the Darwin world, (2) code to read in a species behavior program, and (3) code to execute a species behavior program. Finally, you will need to write some code to actually set up and run the simulation.

3.1 Provided Classes

The following classes are provided to you in full and **should not be modified** in any way:

`Opcode`: This `enum` type contains all possible instruction types in a species behavior program.

`Instruction`: This class represents one instruction of a species behavior program. An instruction consists of an `Opcode` specifying the instruction type and possibly a label.

`Position`: This class represents a particular (x, y) point in the world.

`Direction`: This `enum` type contains all compass directions (North, South, East, and West).

`WorldMap`: This class handles all of the graphics for the simulation (i.e., drawing the graphical grid and the creatures).

3.2 Classes to Write

You are responsible for writing the following classes:

`World`: This class represents a two-dimensional Darwin world, into which you can place creatures.

`Species`: This class represents a species, and provides operations for reading in a behavior program from a file and then interacting with the behavior program.

`Creature`: This class represents an individual creature and provides operations to execute steps of the behavior program on the current creature.

`Darwin`: A wrapper for the `main` method, which is responsible for setting up the world, populating it with creatures, and running the main simulation loop. The details of these operations are generally handled by the other classes. New creatures should be created in random empty locations, pointing in random directions.

Skeletons of the classes to write are provided to you. In particular, all public methods that you should need are already defined (though you may define other public methods if you think it improves the design). You will almost certainly want to add private helper methods to these classes as you implement them.

Complete Javadoc for all classes (both full classes and skeletons) is provided and can be accessed from BlueJ under Tools, then Project Documentation (from the main window). You are highly encouraged to make use of the Javadoc as you are working on your simulator.

3.3 Simulator Interface

The interface of the final simulator (provided by the `main` method of the `Darwin` class) should first prompt for the filename of a species behavior program and a color to use for that species. It should then continue to prompt for additional species, until no filename is entered, at which point the initial world is populated and the graphical simulation is run. The standard world should be 15x15 and should start with 10 creatures per species with randomly-chosen positions and orientations.

An example of the setup interface is shown below:

```
Enter a species filename: creatures/Rover.txt
Enter a species color: red
Enter a species filename: creatures/Flytrap.txt
Enter a species color: green
Enter a species filename:
Starting simulation!
```

While I will not test your simulator on malformed species files, you will want your interface to respond gracefully to invalid input, such as a missing species file or a file that contains an invalid instruction name. You can use the included `BadSpeciesException` class as an easy way to flag a problem with a species file.

4 Implementation Plan

Below is a suggested plan of action for tackling the program. Each item indicates whether it falls under Part 1 or Part 2 of the lab. Don't neglect the intermediate testing! This program has a lot of interconnected components, and trying to move on with incomplete prior pieces is not likely to be a good strategy.

1. (part 1) Write the `World` class. This should be fairly straightforward using a 2D array. **Test this class thoroughly before proceeding. Write a `main` method in the `World` class to verify that all of the methods work.**

2. (part 1) Write the `Species` class. Most of the work here will be parsing the program file and storing it in the `Species`. A suggested approach is to use a `Scanner` to read the file line-by-line (i.e., using `nextLine`) and process each line independently (you may find the `split` method of `String` useful here). **Test this class thoroughly before proceeding. Write a main method in the `Species` class and verify that all of the methods work.**

   It's also a good idea to try to identify invalid species files (e.g., if an opcode requiring a label is found without a label). A graceful way to bail out is to throw an exception if you identify a case like this – you can use the included `BadSpeciesException` to easily do this, like so:

   ```
   if (somethingBad) {
       throw new BadSpeciesException("the file is invalid!");
   }
   ```

   By adding checks like this, you will help with debugging (both now, when you're just reading species files, as well as later, when you're writing your own species files).

3. (part 2) Fill in the *basic* details of `Creature` class. Implement only enough to create creatures and have them display themselves on the world map. Next, implement `execute` for the simple instructions (`left`, `right`, `go`, `hop`). **Test the basic `Creature` class thoroughly before proceeding. Write a main method in that class and verify that all of the methods work.**

4. (part 2) 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.

5. (part 2) Go back to `Creature` and implement the rest of the `execute` method. Test as you go – implement an instruction or two and then verify that a creature will behave correctly using your partially written `Darwin` class.

6. (part 2) Finish up the `Darwin` class. Populate the board with creatures of different species and make your main simulation loop iterate over all the creatures, giving each a turn. Run the simulation for several hundred iterations or so. You can always stop the program by closing the Darwin window.

5 Implementation Advice

Here are specific tips about various parts of the program.

5.1 Working with Colors

The color of each species (and thus, each creature) is indicated by a `Color` object. These `Color` objects are defined as public constants within the `Color` class – e.g., the color red exists as the constant `Color.RED`. While you can also create your own colors by calling the `Color` constructor and passing it red, blue, and green color components, you shouldn't need to do so, and instead can stick with the predefined color constants.

One thing you will need for the main `Darwin` interface, however, is to convert a user-provided color (e.g., the string "green") into its corresponding `Color` object (e.g., the object `Color.GREEN`). While you could write a giant block of conditionals that checks the string against every possible color name, this approach is cumbersome. A more compact approach is to use a fancy Java feature called *reflection*, which allows you to programmatically interact with the source code itself. Below is a method using reflection that will convert the name of a color (i.e., a `String`) to the associated `Color` object by looking up a constant (or "field") with that name in the `Color` class:

```
    /**
     * Get the Color specified by the given name. For a list of valid
     * color names, see the javadoc for the Color class.
     * @param colorName The name of the color to lookup.
     * @return The specified color, or null if no such color exists.
     */
    private static Color colorFromString(String colorName) {
        try {
            Field field = Color.class.getField(colorName);
            return (Color) field.get(null);
        } catch (Exception e) {
            return null; // no such color
        }
    }
```

Feel free to use this method verbatim in your program. You needn't understand exactly how the method above works – reflection is an advanced, fairly esoteric topic and you won't be asked about it on the final exam.

5.2 Switch Statements

There is one more type of control-flow mechanism (besides conditionals and loops) that is present in Java and many other languages and may be quite useful in this program. This mechanism is called a **switch statement**. Basically, a switch statement is a compact and flexible way to pick some action out of a set of actions based on the value of some expression. For example, suppose we wanted to print out the textual representation of an int from 1 to 5. We could write the following, using a conventional block of conditionals:

```
if (val == 1) {
    System.out.println("one");
} else if (val == 2) {
    System.out.println("two");
}  else if (val == 3) {
    System.out.println("three");
} else if (val == 4) {
    System.out.println("four");
} else {
    System.out.println("something else");
}
```

Or, we could instead write this using a `switch` statement, as follows:

```
switch (val) {
    case 1:
        System.out.println("one");
        break;
    case 2:
        System.out.println("two");
        break;
    case 3:
        System.out.println("three");
        break;
    case 4:
        System.out.println("four");
        break;
    default:
        System.out.println("something else");
        break;
}
```

A `switch` statement essentially jumps to the matching case (if one exists), or to the `default` case if none of the other cases match and a default case is provided. A switch statement can have any number of cases, and the cases can have any values you want (and needn't be contiguous, e.g., you could omit case 3 or have a case 8 in the above).

Perhaps the most important note about `switch` statements is to note the `break` lines at the end of every case. While the `break` statement is most often used to immediately exit from a loop, they serve a similar purpose within a switch statement (immediately exit past the end of the switch block). The reason they are so important in most switch statements is that they **aren't implicit** – if you *don't* include the `break` at the end of the switch case, the code will "fall-through" and continue executing subsequent cases. For example, suppose we instead wrote this:

```
switch (val) {
    case 1:
        System.out.println("one");
    case 2:
        System.out.println("two");
    case 3:
        System.out.println("three");
    case 4:
        System.out.println("four");
    default:
        System.out.println("something else");
}
```

If the value of `val` is 2 and we ran the above code, the output would be "two", "three", "four", "something else" (all of them in a row), since the code would jump to case 2, but then fall-through all the rest of the cases.

Fall-through can be very useful in some cases (and can't be replicated using conventional conditionals), but the majority of the time you *don't* want fall-through, and therefore need to remember the `break` statements to prevent it. Note that a `break` statement at the end of the `default` case is redundant, but is often included anyways as a matter of convention.

In the context of this lab, the most likely place you'll find a `switch` statement useful is operating on an `enum` type – a typical pattern is switching on an `enum` value, and then having a case corresponding to each possible value of the `enum`. See the provided `Position` and `Direction` classes for examples of using switch statements on `enum` types. In particular, you may wish to use a `switch` statement in your `execute` method where you are acting based on `Opcode` values.

Lastly, switch statements can generally only operate on integer types (primarily `ints`, `chars`, and `enums`, all of which are internally just ints). Recent versions of Java also support switches on arbitrary strings, but this feature is not supported in most languages (even those that do have `switch` statements). As a matter of interest, Python does not have switch statements at all.

5.3 Simulation Loop

The core of the simulation in the `Darwin` class is the "simulation loop" – this essentially consists of an infinite loop in which simulation rounds are executed. In principle, this continues forever until the program is externally terminated (such as by closing the program window). Note that this is one of the few scenarios in which writing an infinite loop is appropriate.

One very important note about the simulation is that by default, the computer will execute the simulation as fast as possible, and each round will complete in a tiny fraction of a second (making it nearly impossible to see what's happening). Thus, you should explicitly slow down the simulation by calling `WorldMap.pause` in between each simulation round. This added delay will allow you actually see the progress of the simulation. You can set the pause to be whatever you like, but 500 ms is probably reasonable.

Lastly, one note about fairness in the simulation loop: if you implement things in a straightforward way (which you should), you will probably end up with a program in which all creatures of a given species move, followed by all creatures of the next species, and so forth. This approach is arguably unfair, since the creatures of one species always gets to move first. One way to address this is to randomize the order of creatures before starting the simulation. You can randomize the order of a list using the `Collections.shuffle` method:

```
Collections.shuffle(myList); // after executing, myList will have a random order
```

5.4 Drawing the World

All code necessary to draw the graphical world representation is contained in the `WorldMap` class. You must first initialize the graphical map by calling `WorldMap.initialize`, and then can update the map squares using `WorldMap.update`, `WorldMap.clear`, and `WorldMap.updateAndClear`.

You will find things simpler to manage if you initialize the map in the main `Darwin` simulator, and then update individual squares *only* from within the `Creature` class. In other words, each `Creature` is responsible for drawing itself and updating the board whenever the creature moves or otherwise changes something about the world state.

Lastly, there is a significant amount of drawing-related code in the `WorldMap` class that you needn't understand and probably shouldn't spend any time studying (basically everything past `pause`). Graphical code in Java tends to be somewhat ugly.

6 Logistics

6.1 Phases

You will submit your program in two phases:

- By the phase 1 deadline, you must submit preliminary versions of `World.java` and `Species.java`. These files should be essentially complete (though you are welcome to make changes later) and should contain `main` methods that demonstrate that their methods work.
- By the phase 2 deadline, you must submit the entire simulator program (including `World.java`, `Species.java`, `Creature.java`, and `Darwin.java`). In addition, you must submit a species file of your own design. It can be as simple or as complex as you like.

6.2 Tournament

On the last day of class (Thursday, December 7), we will run a tournament in which your submitted species will battle for survival in real-time. Fabulous prizes and bragging rights will be awarded to the winners. The simulator configuration used in the tournament will consist of a 15x15 grid, populated with 10 randomly-placed creatures from each of 4 species.

7 Demo Simulator

To enable you to work on your creatures before your complete simulator is working, you have been provided with a complete working simulator, `darwin.jar`. This is a *jarfile*, which essentially packages a bunch of compiled Java `.class` files (but importantly, *not* the actual source code that produced the `.class` files). You can use the demo simulator to run rounds of Darwin using the included creatures or any creatures that you write yourself.

To run the demo simulator from within BlueJ, follow these steps:

1. Open BlueJ.
2. From the Project menu, select "Open ZIP/Jar".
3. Select `darwin.jar`.

This will open a new BlueJ window showing all of the classes of the demo simulator. From here, you can run the `main` method of the `Darwin` class, which will provide the standard text interface for specifying the species that you want to simulate. Remember that since the jarfile doesn't include any raw source code (just the compiled code), you will not be able to view the actual code of the demo simulator classes (even though you can execute them).

Remember that species files will need to be in the specified place *relative to the* `darwin.jar` *file*, so you may need to move or copies your species files to test them with the demo simulator.

8 Evaluation

As usual, your completed program will be graded on correctness, design, and style. Make sure that your program is documented appropriately and is in compliance with the style guide. Lastly, make sure that your name (and the name of your partner, if applicable) is included in all your modified Java files.

9 Submitting Your Program

Submit your program on Blackboard in the usual way (part 1 by the part 1 deadline, and the full program by the part 2 deadline). Remember to create a zip file named with your username(s) and lab number, e.g., `sbowdoin-jbowdoin-lab9.zip`, and upload that file. Also remember to submit your group reports to me by email if working with a partner.