Lab 8: Darwin[1]
CSCI 2101 – Fall 2021

Checkpoint due: Sunday, December 5, 11:59 pm (see Section 6)
Species submission due: Wednesday, December 8, 11:59 pm
Complete lab due: Friday, December 10, 11:59 pm
Collaboration Policy: Level 1
Group Policy: Pair-optional (with individual creature submissions)

In this final lab of the semester, you will create a simulator for a world that embodies the survival of the fittest: the world of **Darwin**! Your simulator will allow you to create new creature types that execute customized behavior programs, then watch these creatures compete with each other for world domination. Building your simulator will give you more experience writing large, multi-class programs and will demonstrate the power of modular decomposition and encapsulation.

1 The Darwin World

The world of Darwin is a two-dimensional grid of squares, populated by creatures. 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. For example, consider the 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) in addition to its current position.

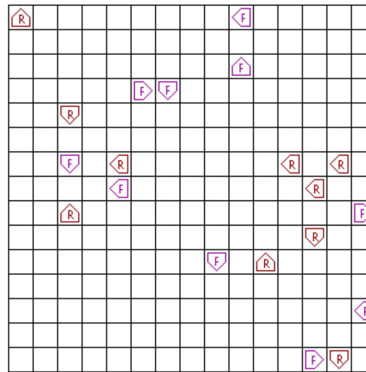

Figure 1: An example Darwin world.

The behavior of each creature is determined by its species. More precisely, we can say that each species defines a particular behavior "program", which applies to creatures of that species. Each creature independently interacts with the world by executing the steps of its behavior program. You can think of each creature as running a separate program, although the actual steps that make up that program will be the same for all creatures of a given species.

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

The simulation proceeds as a series of **rounds**. During each round, every creature gets a **turn**. During a creature's turn, the creature executes one or more **instructions** from its behavior program (described in Section 2), which will ultimately result in one of the following three **actions**:

1. The creature moves forward one space (in the direction of its current orientation).
2. The creature turns left or right 90 degrees.
3. The creature "infects" the creature directly in front of it, transforming the hapless victim into a creature of the infecting species.

In other words, each creature's turn consists of executing one or more instructions that ends in an action instruction. Note that multiple non-action instructions of the behavior program may be executed in a single turn prior to taking an action. 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 each species is world domination via natural selection – every species attempts to infect as many creatures as possible to increase its species population. Note that 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 consists of a series of ordered instructions. The ordering of the instructions is defined by the **address** of each instruction, which counts sequentially starting at zero (e.g., address 2 of the program refers to the third instruction). As an example, the behavior program for the Flytrap species is shown in Table 1.

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

Table 1: The behavior program of the Flytrap species.

As demonstrated in Section 2.2, the species program itself consists only of the instructions (i.e., addresses are implicit, not part of the actual program file). When a creature first starts executing its behavior program, it begins at address 0. Instructions are executed sequentially by address, unless a jump instruction (e.g., `go`, `ifenemy`, etc.) causes a jump to a non-sequential address. Executing an action instruction ends the creature's turn at the current address. When the creature's next turn begins, it picks up right where it left off (i.e., with the address following the last action taken). Note that each creature of the same species is running the same behavior program independently, and thus might be at different places in the behavior program.

As a concrete example, let's summarize the Flytrap behavior program specified above. When a Flytrap takes its first turn, it first executes instruction 0, which is just a **label** instruction marking a particular address. Since a label instruction is not an action, the creature immediately moves onto instruction 1 (in the same turn) and executes that instruction. Instruction 1 checks if an enemy creature is in front of the flytrap. If an enemy is there, the program jumps to the named label ("doinfect", which refers to address 4); if not, the program simply proceeds to the next instruction (address 2). In either case, the program again executes its next instruction (either address 2 or 4), as it has not yet executed an action. This process continues until an action occurs (either the `left` instruction at address 2 or the `infect` instruction at address 5), at which point the flytrap's turn ends. When the flytrap begins its next turn, it will resume at either address 3 or address 6, depending on which action it took in the previous turn.

Note that the instructions of a behavior program resemble a rudimentary programming language, in which the flow of commands executed is determined not by conditionals and loops but instead by simple "jump-to-address" style instructions[2]. In the course of doing this lab, you will essentially be writing an interpreter for this very simple programming language.

2.1 Instruction Listing

Some instructions (such as `left`) consist only of the instruction name, while other instructions (such as `ifenemy`) take the name of a program label as an argument. Executing an action instruction (`left`, `right`, `hop`, or `infect`) always ends the creature's turn, regardless of what effect executing the instruction had (even if the action instruction had no effect). A complete listing of the 11 valid instruction types (including the 4 action instruction types) is given below.

`left` The creature turns left 90 degrees. *Action instruction.*

`right` The creature turns right 90 degrees. *Action instruction.*

`hop` The creature moves forward one square, 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 creature does not move. *Action instruction.*

`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:`)** – the colon is what designates the instruction as a label! The label instruction is the *only* instruction type that includes any non-alphanumeric characters.

`go label` Continues execution of the program from the address indicated by `label` (i.e., jumps to the indicated address).

`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.

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

`ifwall label` If the creature is facing a border of the world, 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).

`infect label` Infects the creature immediately in front of this creature. An infected creature 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 omitted, the infected creature starts at address 0 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 to infect in front of this creature. *Action instruction.*

2.2 Behavior Program Files

A behavior program is written in a plain text file (e.g., `flytrap.txt`). In addition to the instructions of the behavior program, a behavior 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, which must contain the species name and nothing else).

Shown below is the full contents of the program file for the Flytrap species (`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

The following species types are pre-supplied as behavior program files, which will be useful for testing and for getting a sense of what is possible using behavior programs:

`Food`: This creature spins but never hops or infects anything. The life of the Food creature is so boring that its only hope in life is to be eaten by something else, thus reincarnating as something more interesting.

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

`Flytrap`: As previously discussed, this creature spins in one place and infects any enemy creature that happens by.

`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.

Feel free to use these creatures as inspiration (or competition) for your own species as you write new behavior programs!

3 The Darwin Simulator

Your task is to complete a program that can execute rounds of a Darwin simulation. This task 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 simulator components 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 (which refers to an enumeration of a fixed set of possible values) contains all possible instruction types in a species behavior program. Each opcode specifies a particular instruction type (e.g., `hop`, `ifsame`, etc).

`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 instructions of the behavior program on the current creature.

`Darwin`: A wrapper for your final `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.

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 in which 2 species are specified:

```
Enter a species filename: species/Rover.txt
Enter a species color: red
Enter a species filename: species/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 behavior program 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 (i.e., throw one of these exceptions whenever you encounter a program during processing a behavior program file).

4 Implementation Plan

Below is a suggested plan of action for tackling the program. Steps 1 and 2 must be completed by the checkpoint deadline (see Section 6). Don't neglect intermediate testing! This program has a lot of interconnected components, and trying to move on with incomplete prior pieces is not likely to be an effective strategy.

1. (pre-checkpoint) 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.** Note that you will need to create `Creature` objects to test this class – there is a zero-parameter dummy constructor provided in the `Creature` class that you can use for this purpose.

2. (pre-checkpoint) Write the `Species` class. Most of the work here will be parsing the program file and storing it in the `Species` objects. 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 that needs a label does not have one). 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 existing species files) as well as later (when you're writing your own species files).

3. Fill in the basic details of the `Creature` class. Implement enough to create creatures and have them display themselves on the world map. Next, implement `execute` for the simple instructions (`left`, `right`, `go`, `hop`). You will test these instructions in the next step.

4. 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. You should be able to have a creature like `Hop` move across the world.

5. 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. 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 Enumeration Types

We often want to be able to represent one out of some predefined set of possible values. For example, suppose we're writing a program involving playing cards. Each card has some suit (spade, heart, club, or diamond), and we need some way to represent the card suit. One reasonable approach is to represent each value by some arbitrary, unique `int` constant, such as shown below:

```
public static final int SPADE = 0; // values are arbitrary but unique
public static final int HEART = 1;
public static final int CLUB = 2;
public static final int DIAMOND = 3;
...
int myCardSuit = HEART;
```

While this approach is simple, it has a few downsides. The major issue is that we're representing a card suit by an `int`, but it's supposed to be just one of the four specified values, not any arbitrary integer. Since the variable is just an `int`, there's nothing stopping you from assigning some invalid value (e.g., 50) to `myCardSuit`, which wouldn't make sense. Similarly, the `int` type itself doesn't enforce any proper usage; e.g., if you wrote some method that accepted a card suit as a parameter, the parameter type would be `int`, but there would be no way to prevent the caller from handing the method some arbitrary integer that's not `SPADE`, `HEART`, `CLUB`, or `DIAMOND`.

A better way to approach this problem is to define the set of possible values using an `enum` (enumeration) type, which is a special type of class that defines a fixed set of instances of the class (and no further instances can ever be created). For example:

```
public enum Suit {
    SPADE, HEART, CLUB, DIAMOND
}
```

Having defined this `enum` type, we can now use the new type as shown below:

```
Suit myCardSuit = Suit.HEART; // or Suit.SPADE, etc.
```

Using an `enum` has multiple benefits here. One, we now have a new type (`Suit`) that explicitly represents a card suit, which is better than just using an `int` type. Two, the code will now enforce that any `Suit` variable is one of the specified allowed values. Three, the code is cleaner, and doesn't require defining a bunch of constants with arbitrary numeric values. While many `enum` types are just a listing of the possible values (like the above example), you can also give them methods, which can then be called on the `enum` instances (e.g., `myCardSuit.getSuitColor()`).

In this lab, two of the starter classes are `enum` types – `Direction`, which represents a compass direction (`NORTH`, `SOUTH`, `EAST`, or `WEST`), and `Opcode`, which represents an instruction type in a species program (`HOP`, `LEFT`, `IFEMPTY`, etc.). You should not modify these classes, but you will need to use the enum types in the rest of the program. Both of these classes define methods that can be called on the enum instances. For example, if `dir` is a `Direction` (e.g., `Direction.NORTH`), then you could call `dir.right()` to get the direction to the right of it (e.g., `Direction.EAST`).

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. One straightforward way to do this would be via a large block of conditionals, like the following:

```
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");
}
```

We could alternately write this code 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).

In 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 the `execute` method of the `Creature` class, where you are acting based on `Opcode` values.

Note that every case in the above example ends in a `break` statement, which you've probably only seen before in the context of exiting from a loop. A `break` statement in a switch behaves similarly (immediately jumping to the end of the switch), but importantly, these breaks **aren't implicit**. In other words, if you *don't* include the `break` at the end of a given case, the code will "fall-through" and continue executing subsequent cases. For example, suppose we wrote the previous switch statement, but omitted the break statements:

```
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 we ran the above code with a `val` of 3, the program would jump to case 3 but then fall-through the rest of the cases, resulting in the following output:

```
three
four
something else
```

Fall-through can be useful in some cases (and can't be replicated using regular conditionals), but for most simple situations 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.

Lastly, switch statements can generally only operate on integer types (primarily `ints`, `chars`, and `enums`, all of which are internally just ints), although recent versions of Java also support switches on strings. Some languages (such as Python) do not support switch statements at all.

5.3 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 `String` (e.g., "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 (as 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 (and you needn't understand exactly how it works).

5.4 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.

By default, the computer will execute the simulation loop 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 `pause` (defined in `WorldMap`, and discussed in the next section) in between each simulation round. This added delay will allow you to 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, you will probably end up with a loop 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 `someList` by calling `Collections.shuffle(someList)`.

5.5 Drawing the World

All code necessary to draw the graphical world representation is contained in the supplied `WorldMap` class. Your program will update the graphical map by calling public methods of this class. A summary of the public methods of this class is given below (full details, including parameters, are available in the javadoc):

- `initialize`. Initializes the world map. Should be called exactly once prior to calling any of the other drawing functions.
- `drawCreature`. Draws a given `Creature` at its current position.
- `clearSquare`. Clears a given square of the map.
- `drawMovedCreature`. Re-draws a creature that just moved – essentially a shortcut to calling `drawCreature` and `clearSquare` together, but performs the animation in a smoother way.
- `pause`. Pauses the simulation for a given duration, used to prevent things from finishing too quickly.

Note that all of the above methods are `static`, meaning that you do not call them on an instance of the `WorldMap` class like you would with a regular method. However, since you will be calling these methods from outside the `WorldMap` class itself, the way you call them is slightly different – you invoke them on the class itself rather than an instance of the class. For example:

```
WorldMap.pause(500); // method invoked on class itself, not an instance
```

You may recognize this form of method call from other public static methods you've used in the past, such as `Math.round` or `Integer.parseInt`. You will never actually construct an instance of the `WorldMap` class.

Drawing Management. 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.

6 Logistics and Dates

You will submit your lab in three phases:

- By the Checkpoint deadline, you must submit current 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 Species deadline, you must submit a species file of your own design. It can be as simple or as complex as you like. Everyone should submit their own species file, even if you are working in a group.
- By the final lab deadline, you must submit the entire simulator program (including `World.java`, `Species.java`, `Creature.java`, and `Darwin.java`).

6.1 Tournament

On our last day of class, we will run a tournament in which your submitted species will battle for survival in front of a live audience (you). Fabulous prizes and bragging rights will be awarded! The simulator configuration used in the tournament will consist of a 15x15 grid populated with 10 randomly-placed creatures from each of 4 species. Following a typical tournament setup, we will run a series of initial rounds in which the winners will advance to the finals to determine the ultimate fittest species.

7 Demo Simulator

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

To run the demo simulator from within BlueJ, select "Open ZIP/Jar" from the Project menu, then select `darwin-demo.jar`. This will create and open a new BlueJ project called `darwin-demo`. From here, you can run the `main` method of the `Darwin` class, which will provide the standard text interface as described in Section 3.3. Remember that since the jarfile only includes compiled code, you will not be able to see the code of the complete classes (even though you can execute them).

Alternately, you can run the demo simulator from the command line (on a Mac) by opening the Terminal application (`/Applications/Utilities/Terminal`), navigating to the folder containing `darwin-demo.jar`, and entering `java -jar darwin-demo.jar` (then hitting return).

Either way, executing the jarfile will provide the text interface specified in Section 3.3, prompting you to enter species and colors. The species files will need to be in the specified locations relative to the project directory, so you might need to move or copies your species files to test them with the demo simulator. The test species are included in the `species` directory (within the `darwin-demo` directory), so you can refer to them as, for example, `species/Flytrap.txt`.

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 coding and 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 – one group submission for the checkpoint, individual species submissions, and one group submission for the final deadline. Remember to create a zip file named with your username(s) and lab number, e.g., `sbowdoin-jbowdoin-lab8.zip`, and upload that file. Also remember to submit your individual group reports to me if working with a partner (only at the end of the lab – no group reports necessary for the checkpoint).