Darwin

Due: Friday October 11, 2016

Objectives

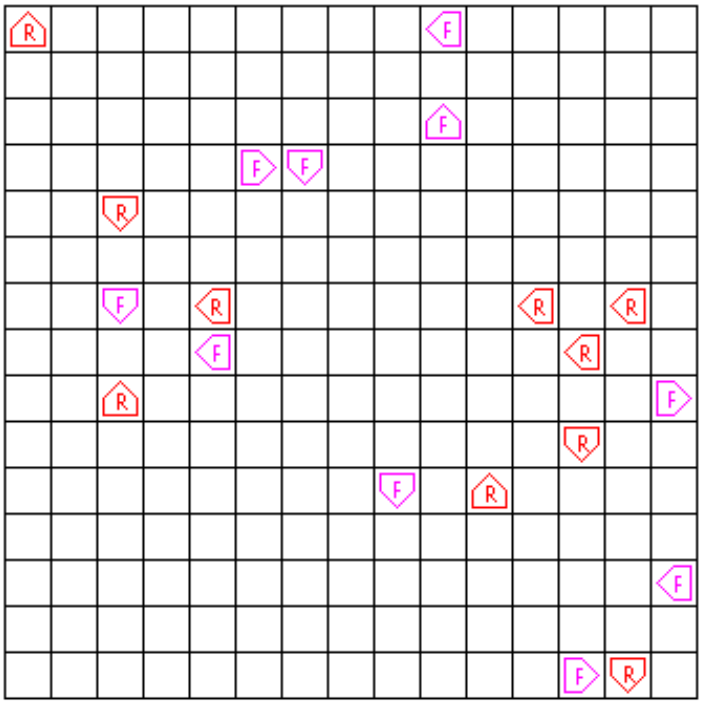
For this assignment, you will:

- Gain practice writing large, multi-class programs.
- Learn the importance of modular decomposition and information hiding. (The entire program is broken down into a series of classes that can be developed and tested independently, without revealing representational details.)
- Gain more practice with `JUnit`.
- Have fun with a problem that is algorithmically interesting in its own right!

This is a larger program than the previous labs, and you will have two weeks to complete it.

Description

In this assignment, your job is to build a simulator for a game called Darwin invented by Nick Parlante. The Darwin program simulates a two-dimensional world divided up into small squares and populated by a number of creatures. Each of the creatures lives in one of the squares, faces in one of the major compass directions (North, East, South, or West) and belongs to a particular species, which determines how that creature behaves. For example, one possible world is shown below:



The sample world on the previous page is populated with twenty creatures, ten of a species called Flytrap and ten of a species called Rover. In each case, the creature is identified in the graphics world with the first letter in its name. The orientation is indicated by the figure surrounding the identifying letter; the creature points in the direction of the arrow. The behavior of each creature–which you can think of as a small robot–is controlled by a program that is particular to each species. Thus, all of the Rovers behave in the same way, as do all of the Flytraps, but the behavior of each species is different from the other.

As the simulation proceeds, every creature gets a turn. On its turn, a creature executes a short piece of its program in which it may look in front of itself to see what's there and then take some action. The possible actions are moving forward, turning left or right, or infecting some other creature standing immediately in front, 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
1	`ifenemy 4`	If there is an enemy ahead, go to step 4
2	`left`	Turn left
3	`go 1`	Go back to step 1
4	`infect`	Infect the adjacent creature
5	`go 1`	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 go instruction that will cause the program to start over again at the beginning of the program.

Programs are executed beginning with the instruction in step 1 and ordinarily continue with each new instruction in sequence, although this order can be changed by certain instructions in the program. Each creature is responsible for remembering the number of the next step to be executed. The instructions that can be part of a Darwin program are listed below:

`hop:` The creature moves forward as long as the square it is facing is empty. If moving forward would put the creature outside the boundaries of the world or would cause it to land on top of another creature, the hop instruction does nothing.

`left:` The creature turns left 90 degrees to face in a new direction.

`right:` The creature turns right 90 degrees.

`infect n:` If the square immediately in front of this creature is occupied by a creature of a different species (an "enemy") that creature is infected to become the same as the infecting species. When a creature is infected, it keeps its position and orientation, but changes its internal species indicator and begins executing the same program as the infecting creature, starting at step n of the program. The number n is optional. If it is missing, the infected creature should start at step 1.

`ifempty n:` If the square in front of the creature is unoccupied, update the next instruction field in the creature so that the program continues from step n. If that square is occupied or outside the world boundary, go on with the next instruction in sequence.

`ifwall n:` If the creature is facing the border of the world (which we imagine as consisting of a huge wall) jump to step n; otherwise, go on with the next instruction in sequence.

`ifsame n:` If the square the creature is facing is occupied by a creature of the same species, jump to step n; otherwise, go on with the next instruction.

`ifenemy n:` If the square the creature is facing is occupied by a creature of an enemy species, jump to step n; otherwise, go on with the next instruction.

`ifrandom n:` In order to make it possible to write some creatures capable of exercising what might be called the rudiments of "free will," this instruction jumps to step n half the time and continues with the next instruction the other half of the time.

`go n:` This instruction always jumps to step n, independent of any condition.

A creature can execute any number of `if` or `go` instructions without relinquishing its turn. The turn ends only when the program executes one of the instructions `hop`, `left`, `right`, or `infect`. On subsequent turns, the program starts up from the point in the program at which it ended its previous turn.

The program for each species is stored in a file in the subfolder named Creatures in the assignment folder. Each file in that folder consists of the species name and color, followed by the steps in the species program, in order. The program ends with an empty line. Comments may appear after the blank line or at the end of each instruction line. For example, the program file for the Flytrap creature looks like this:

```
Flytrap
magenta
ifenemy 4
left
go 1
infect
go 1

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

There are several presupplied creature files:

`Food`: This creature spins in a square but never infects anything. Its only purpose is to serve as food for other creatures. As Nick Parlante explains, "the life of the Food creature is so boring that its only hope in life is to be eaten by something else so that it gets reincarnated as something more interesting."

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

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

`Rover`: This creature walks in straight lines until it is blocked, infecting any enemy creature it sees. If it can't move forward, it turns.

You can also create your own creatures by creating a data file in the format described above.

Classes

Your mission is to write the Darwin simulator and get it running. The program is large enough that it is broken down into a number of separate classes that work together to solve the complete problem.

You are responsible for implementing the `Darwin`, `Species`, `Creature`, and `World` classes. Skeletons of these classes are provided in the starter folder. In addition, we provide you with three helper classes that you should use without modification: `Instruction`, `Position`, and `WorldMap`. Documentation for all of these classes is provided at the end of the handout. Familiarize yourself with the classes before you begin.

`Darwin` class

This class contains the main program, which is responsible for setting up the world, populating it with creatures, and running the main loop of the simulation that gives each creature a turn. The details of these operations are generally handled by the other modules. New creatures should be created in random empty locations, pointing in random directions.

`Species` class

This class represents a species, and provides operations for reading in a species description from a file and for working with the programs that each creature executes.

`Creature` class

Objects of this class represent individual creatures, along with operations for creating new creatures and for taking a turn.

`World` class

This class contains an abstraction for a two-dimensional world, into which you can place the creatures.

`Instruction` class

This simple class represents one instruction out of the Species's instruction set. This class is already implemented for you.

`Position` class

This class represents (x,y) points in the world and constants for compass directions. These are similar to what we used in the Maze solving program. This class is already implemented for you.

`WorldMap` class

This class handles all of the graphics for the simulation. This class is already implemented for you.

Getting Started

Here is a suggested course of action to implement Darwin:

1. Copy the following darwin starter files from `/common/cs/cs062/assignments/assignment06` and create your project.

2. You can double click on `DarwinTest.jar` to run my sample solution. This will give you a chance to see how the program is supposed to behave. It should prompt you to open creature files. Select at least two different creature types. When you are done, click on cancel and the program should commence running.

3. Write the `World` class. This should be straight-forward if you use a 2-dimensional array to represent the world. **Test this class thoroughly before proceeding. Write a JUnit Test Class that verifies that all of the methods work.**

4. Write the `Species` class. The hardest part will be parsing the program file and storing it in the `Species`. Note that the first instruction of a program is at address 1, not 0. **Test this class thoroughly before proceeding. Write a JUnit test class for that class and verify that all of the methods work.** See the hints within the starter file for the `Species` class on how to read lines from the files.

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

6. Begin to implement the simulator in the `Darwin` class. Start by reading a single species and creating one creature of that species. Write a loop that lets the single creature take 10 or 20 turns.

7. Go back to `Creature` and implement more of the `takeOneTurn` method. Test as you go – implement an instruction or two, and verify that a Creature will behave correctly, using your partially written `Darwin` class.

8. Finish up the `Darwin` class. Populate the board with creatures of different species (I suggest about 10 of each kind of creature) and make your main simulation loop iterate over the creatures, giving each a turn. The class should create creatures for the species given as command line arguments to the program when you run it. See `Darwin.java` for more details. Run the simulation for several hundred iterations or so. You can always stop the program by pressing control-C in the terminal window or closing the Darwin Window. Make sure you know how to pass in arguments to the `main` method as the autograder will be passing in command line arguments to run Darwin. Please do not change the folder structure – you should keep all creature files in the Creature folder.

9. Finally, finish testing the implementation by making sure that the creatures interact with each other correctly. Test `ifenemy`, `infect`, etc.

Submitting Your Work

1. You must turn in the complete program as well as corresponding JUnit classes by midnight on Sunday, October 25. Please name your folder as **Assignment06_LastnameFirstname**.

2. You must also turn in a species of your own design in a separate file and name it **Assignment06_LastnameFirstname_species.txt**. It can be as simple or as complex as you like, but must use only the instructions specified above for creatures. We will pit your creatures against each other to watch them battle for survival! Door prizes will be awarded!

Grading

You will be graded based on the following criteria:

criterion	points
JUnit test classes for World and Species	2
World class passes World unit tests	2
Species class passes Species unit tests	4
Creature class correctly implements one turn	3
Darwin class correctly runs the game of Darwin	3
General correctness	3
Appropriate comments (including JavaDoc)	3
Style and formatting	3
Extra Credit	2

Possible Extensions

There are many ways to extend the program to simulate more interesting Species behavior. Here are just a few ideas if you wish to extend Darwin for extra credit:

1. Give creatures better eyesight. Add `if2enemy n`. This instruction checks if there is an enemy two steps in front of a creature. This can help make fly traps much more lethal. You could also add peripheral vision and let creatures see the either side. You can add similar variants for the other tests as well.

2. Give creatures memory. This can be as simple as permitting each creature store a single integer. You can then add the following instructions to the instruction set: `set n` to set a creature's memory; `ifeq v n` to jump to address `n` in the program if a creature's memory contains `v`; and `inc` and `dec` to add and subtract from memory. None of these instructions should end a turn. You can get more coordinated activity by using this type of memory.

3. Give creatures the ability to communicate. Once creatures have memory, let them ask the creature on the square infront of them what is on its mind with the `ifmemeq v n`. This instruction reads the memory of the creature in the square infront of a creature and jumps to `n` if that value is `v`. You can also add `copymem` that copies the value in the memory value of the creature infront of you to your own memory. These instructions permit creatures to form quite successful "phalanx" formations.

4. Make creatures mutate. Perhaps copies of creatures aren't quite the same as originals— when a creature infects another creature, make there be a chance that the infected creature will be a mutation of the infecting creature's species. This will require creating new Species that are close, but not quite exact, copies of an existing Species. Taken to its extreme, you can make species evolve over time to become better and better at surviving in the Darwin world. Come talk to me about this one if you want to try it.

5. Implement any other extension you find interesting.

As usual, name your folder **Assignment06_LastnameFirstname_ExtraCredit** if you attempted any.

```
/**
 * A Position is an (x,y) coordinate in the World, much like
 * the Positions for the maze program.
 */
public class Position {
    static public final int NORTH = 0;
    static public final int EAST = 1;
    static public final int SOUTH = 2;
    static public final int WEST = 3;

    /**
     * Create a new position for the given x and y coordinates.
     */
    public Position(int x, int y)

    /**
     * Return the x coordinate for the position.
     */
    public int getX()

    /**
     * Return the y coordinate for the position.
     */
    public int getY()

    /**
     * Return a new position that is in one of the four
     * compass directions from this.
     * @pre direction must be NORTH, SOUTH, EAST, or WEST.
     * @post the Position adjacent to this in the given direction.
     */
    public Position getAdjacent(int direction)
}
```

```
/**
 * This class includes the functions necessary to keep track of the creatures
 * in a two-dimensional world. There are many ways to implement this class but we
 * recommend looking at the Matrix class in Bailey's structure5 package.
 */
public class World<E> {

    /**
     * This creates a new world consisting of width columns
     * and height rows, each of which is numbered beginning at 0.
     * A newly created world contains no objects.
     */
    public World(int w, int h)

    /**
     * Returns the height of the world.
     */
    public int height()

    /**
     * Returns the width of the world.
     */
    public int width()

     /**
     * Returns whether pos is in the world or not.
     * @post returns true if pos is an (x,y) location within the bounds of
     * the board.
     */
    boolean inRange(Position pos)

    /**
     * Set a position on the board to contain c.
     * @pre  pos is in range
     */
    public void set(Position pos, E c)

    /**
     * Return the contents of a position on the board.
     * @pre  pos is in range
     */
    public Object get(Position pos)
}
```

```
/**
 * This class represents one Darwin instruction.  Instructions
 * contain two parts: an opcode and an address.  For instructions
 * that do not perform jumps (hop, left, right, infect), the address
 * part is not used.
 */
public class Instruction {
    public static final int HOP =      1;
    public static final int LEFT =     2;
    public static final int RIGHT =    3;
    public static final int INFECT =   4;
    public static final int IFEMPTY =  5;
    public static final int IFWALL =   6;
    public static final int IFSAME =   7;
    public static final int IFENEMY =  8;
    public static final int IFRANDOM = 9;
    public static final int GO =       10;

    /**
     * Creates a new instruction.  address is the target of
     * the operation, if one is needed.  Otherwise it is not used.
     * @pre 0 < opcode <= GO.
     */
    public Instruction(int opcode, int address)

    /**
     * Returns the opcode
     * @post returns the opcode
     */
    public int getOpcode()

    /**
     * Returns the addrss
     * @post returns the address
     */
    public int getAddress()
}
```

```
/**
 * The individual creatures in the world are all representatives of some
 * species class and share certain common characteristics, such as the species
 * name and the program they execute. Rather than copy this information into
 * each creature, this data can be recorded once as part of the description for
 * a species and then each creature can simply include the appropriate species
 * reference as part of its internal data structure.
 * <p>
 *
 * To encapsulate all of the operations operating on a species within this
 * abstraction, this class provides a constructor that will read a file containing
 * the name of the creature and its program, as described in the earlier part
 * of this assignment. To make the folder structure more manageable, the
 * species files for each creature are stored in a subfolder named Creatures.
 * Thus, creating the Species for the file Hop.txt will cause the program to
 * read in "Creatures/Hop.txt".
 * <p>
 *
 * Note: The instruction addresses start at one, not zero.
 */
public class Species {

/**
 * Create a species for the given file.
 * @pre fileName exists in the Creature subdirectory.
 */
public Species(BufferedReader fileReader)

/**
* Return the char for the species
*/
public char getSpeciesChar()

/**
 * Return the name of the species.
 */
public String getName()

/**
 * Return the color of the species.
 */
public String getColor()

/**
 * Return the number of instructions in the program.
 */
public int programSize()

/**
 * Return an instruction from the program.
```

```
 * @pre 1 <= i <= programSize().
 * @post returns instruction i of the program.
 */
public Instruction programStep(int i)

/**
 * Return a String representation of the program.
 */
public String programToString()
```

```
/**
 * This class represents one creature on the board.
 * Each creature must remember its species, position, direction,
 * and the world in which it is living.
 * <p>
 * In addition, the Creature must remember the next instruction
 * out of its program to execute.
 * <p>
 * The creature is also responsible for making itself appear in
 * the WorldMap.  In fact, you should only update the WorldMap from
 * inside the Creature class.
 */
public class Creature {

    /**
     * Create a creature of the given species, with the indicated
     * position and direction.  Note that we also pass in the
     * world-- remember this world, so that you can check what
     * is in front of the creature and to update the board
     * when the creature moves.
     */
    public Creature(Species species, World world, Position pos, int dir)

    /**
     * Return the species of the creature.
     */
    public Species species()

    /**
     * Return the current direction of the creature.
     */
    public int direction()

    /**
     * Return the position of the creature.
     */
    public int position()

    /**
     * Execute steps from the Creature's program until
     * a hop, left, right, or infect instruction is executed.
     */
    public void takeOneTurn()

    /**
     * Return the compass direction that is 90 degrees left of dir.
     */
    public static int leftFrom(int dir)

    /**
```

```
     * Return the compass direction that is 90 degrees right of dir.
     */
    public static int rightFrom(int dir)
}
```

```
/**
 * This class controls the simulation.  The design is entirely up to
 * you.  You should include a main method that takes the array of
 * species file names passed in and populates a world with species of
 * each type.
 * <p>
 * Be sure to call WorldMap.pause every time
 * through the main simulation loop or else the simulation will be too fast
 * and keyboard / mouse input will be slow. For example:
 * <pre>
 *   public void simulate() {
 *       for (int rounds = 0; rounds < numRounds; rounds++) {
 *         giveEachCreatureOneTurn();
 *         WorldMap.pause(100);
 *       }
 *   }
 * </pre>
 */
class Darwin {

    /**
     * The array passed into main will include the arguments that
     * appeared on the command line.  For example, running "java
     * Darwin Hop.txt Rover.txt" will call the main method with s
     * being an array of two strings: "Hop.txt" and "Rover.txt".
     */
    public static void main(String s[])

}
```

```
/**
 * This class exports the methods necessary to display the creatures
 * on the screen.  You should not change this class.
 * You should have the following as the first line of your main:
 *
 *   createWorldMap(x, y);
 *
 * Only the Creature code should need to
 * call the displaySquare method.
 */
public class WorldMap {

    /**
     * Initialized the world map to have size (x,y)
     * and create the Window for it.  This must be called
     * once and only once.
     */
    static public void createWorldMap(int x, int y)

    /**
     * Update the contents of the square indicated by pos.
     * If c is ' ', then the square is cleared, and dir and color are ignored.
     *
     * @pre pos is a valid position
     * @pre <p> c is the character to be put into the square
     * @pre <p> dir is Position.NORTH,Position.SOUTH,Position.EAST,or Position.WEST
     * @pre <p> color is "black", "red", "gray", "dark gray", "pink",
     *                   "orange", "yellow", "green", "magenta", "cyan", "blue"
     */
    static public void displaySquare(Position pos, char c, int dir, String color)

    /**
     * Pause for millis milliseconds.  Call this atleast once
     * in the main simulation loop so that the computer can process
     * mouse / keyboard events.
     *
     */
    static public void pause(int millis)

}
```

Parts of this handout were borrowed from Nick, Eric Roberts, Bob Plummer, and Stephen Freund.