CPSC 121
Project 02: Random Walk

Project Overview:

In this project, you will generate a random walk on a grid that starts at the bottom-left corner and ends up at the top-right corner.

- Classes that you will create: `RandomWalk.java, RandomWalkDriver.java`
- Existing interface that you will use: `edu.cwi.randomwalk.RandomWalkInterface (RandomWalk.jar)`
- Existing classes that you will use: `RunUnitTester.java, RunGUITester.java`

Objectives:

1. Write a class that implements methods of an interface, including overloaded constructors and a `toString()`.
2. Write a driver class that uses the above class.
3. Use classes from the java standard library.
4. Use existing classes.
5. Create a list of objects using the `ArrayList` class.

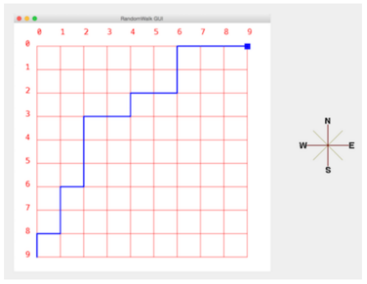
Getting Started

1. Create a new folder called `Project_02` for this project, create a folder called `lib` within that folder, and copy/paste `RandomWalk.jar` into your project
2. Copy/paste the `RunUnitTester.java` and `RunGUITester.java` files into your project directory.
3. You will need to create a new class in your project called `RandomWalk`.
4. You will need to create a new driver class in your project called `RandomWalkDriver`. This will contain the main method.

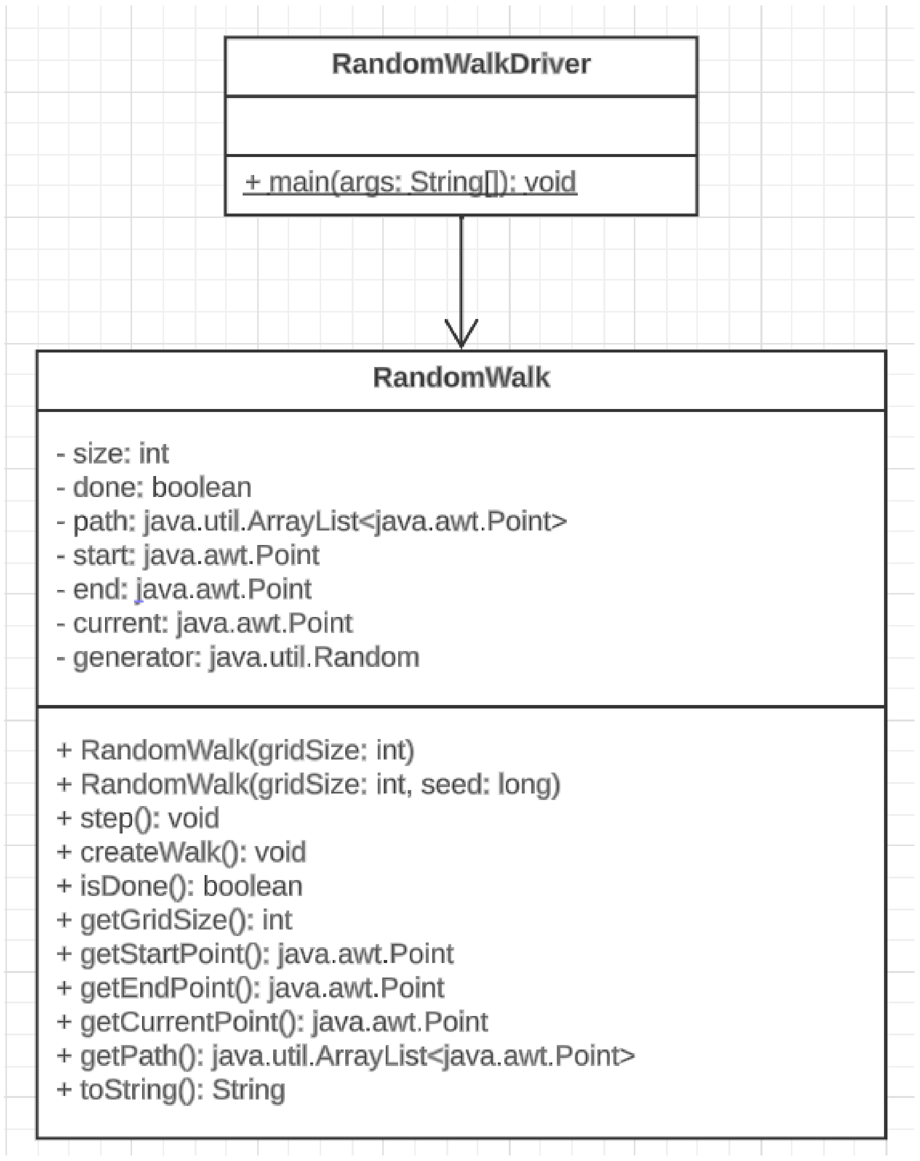
Note – There will be several errors, but these are expected until you finish implementing your `RandomWalk.java` class.

Specification

1. Generate a random walk on an ‘m’ by ‘n’ grid that starts at (0, m – 1) (bottom-left-corner) and ends up at (n – 1, 0) (top-right corner).
2. At each point, there are four potential directions in which you can walk: North, East, South, and West. To avoid getting stuck, we will ***only*** walk to the North or East with equal probability. This way, we will always be making progress towards the destination (the north-east corner).
3. See the figure to the right for a sample random walk.



UML Diagram


RandomWalkDriver
+ main(args: String[]): void
RandomWalk
- size: int
- done: boolean
- path: java.util.ArrayList<java.awt.Point>
- start: java.awt.Point
- end: java.awt.Point
- current: java.awt.Point
- generator: java.util.Random
+ RandomWalk(gridSize: int)
+ RandomWalk(gridSize: int, seed: long)
+ step(): void
+ createWalk(): void
+ isDone(): boolean
+ getGridSize(): int
+ getStartPoint(): java.awt.Point
+ getEndPoint(): java.awt.Point
+ getCurrentPoint(): java.awt.Point
+ getPath(): java.util.ArrayList<java.awt.Point>
+ toString(): String

Part 1: Creating the RandomWalk Class

1. If you have not already, create a RandomWalk class from the UML above. Think about the properties this class will have based upon the above UML:
    a. Size of the grid
    b. A random number generator
    c. A Boolean flag named done
    d. An ArrayList of points to represent the path of the random walk
    e. The start point of the walk
    f. The end point of the walk
    g. The current point of the walk
2. Your class must implement the `edu.cwi.randomwalk.RandomWalkInterface`. To implement the interface, you must modify your class header as follows:

```
public class RandomWalk implements RandomWalkInterface {}
```

3. Use the Point class provided in the Java standard library (in the `java.awt` package) to represent each point on the path. For example, the start point can be represented as:

```
start = new Point(0, gridSize - 1);
```

   a. You can access the coordinates of the point using: `start.x` or `start.y`.
4. Use an `ArrayList` of `Point` objects to store the path. The `ArrayList` class is part of the `java.util` package and provides functionality for managing a list of objects. There are several methods available in the `ArrayList` class (e.g. `add`, `remove`, `isEmpty`, `contains`, et cetera). You can construct an `ArrayList` instance for managing your `Point` objects as follows:

```
path = new ArrayList<Point>();
```

   a. Then, you can add new points to your path using the `add()` method of the `ArrayList` class as follows:

```
Point p1 = new Point(x, y);
path.add(p1);
```

5. The following are signatures for the methods you must implement in your `RandomWalk` class and as represented in the UML above. The methods are also defined in the `RandomWalkInterface`. Use the following instructions to flesh out the logic of the constructors and methods.
   a. `public RandomWalk(int gridSize)`
      Initializes the properties and adds the starting point of the walk, but does not create the entire walk. Instantiates a random number generator without a seed.
   b. `public RandomWalk(int gridSize, long seed)`
      Same as the above constructor except that we specify a seed for the random number generator. This is very useful for debugging and testing as the same seed will give the same random number sequence so that we can reproduce a bug! This constructor illustrates the concept of method overloading.
   c. `public void step()`
      Makes the walk go one more step. To add a step to your path, you will add the new Point to your ArrayList. A new step should be added to your path every time the method is called.

      If the step is the final step, set the value of the done instance variable to true to signal that you are done with the random walk.

      It should not take a step if the done variable is set to true.

      Note that you will walk North or East with equal probability (use the random number generator). If you are at one of the edges, then you may have only one direction in which you can walk. You can handle this case in multiple ways. For example, you can always pick one of North or East and then if you find you cannot go in that direction, just generate another random choice until you can move.
   d. `public void createWalk()`
      Creates the entire walk in one call by internally using the `step()` method.
   e. `public Boolean isDone()`
      Returns the current value of the done property.
   f. `public int getGridSize()`
      Returns the size of the grid.
   g. `public Point getStartPoint()`
      Returns the starting point of the walk.

h. `public Point getEndPoint()`
   Returns the ending point of the walk.
i. `public Point getCurrentPoint()`
   Returns the current point of the walk.
j. `public ArrayList<Point> getPath()`
   Getter method that returns a copy of the random walk path ArrayList.
k. `public String toString()`
   Returns the path as a nicely formatted string as shown below:

```
[0,4] [0,3] [1,3] [1,2] [2,2] [3,2] [3,1] [4,1] [4,0]
```

Part 2: Testing your RandomWalk Class

1. Write a `RandomWalkDriver` class that does the following:
   a. Implement a main method that deals with the user input as follows:
      i. Prompts the user for the grid size and checks to make sure the user enters a positive integer. If the entered value is invalid, ask the user to enter a different integer. Keep doing this until they enter a valid integer. You can do this with a loop.
      ii. Prompt the user for the random seed value and check to make sure that the user enters a 0 or positive integer. If the entered value is invalid, ask the user to enter a different integer. Keep doing this until they enter a valid integer. You can also do this with a loop.
      iii. Create a `RandomWalk` object with the appropriate constructor. If the seed is zero, then call the constructor with only grid size as an argument. Otherwise call the constructor with two arguments.
      iv. Create the walk using the `createWalk()` method and then print the walk (using the `toString()` method in the `RandomWalk` class).
   b. Hints on testing:
      i. Test for bad input like negative or zero for grid size, or negative random seed to make sure your program catches the error and asks the user to enter values again and again until they get it right.
      ii. Test for boundary cases:
         1. Try small values like 1, 2, 3, 4 for grid size to see if the path generated is correct.
         2. Try random seeds like ‘1234’ and verify that you generate the same path each time.
         3. Try random seed input of zero and verify that you get different random paths each time.
         4. Try a few larger grid sizes like 10, 20, 100 to see if your program stops normally. It is hard to check if the output is correct for large grid sizes but using the GUI program will make it easier.
         5. Use the `RunUnitTester` program to make it easier to debug your program.

Sample Output:

```
java RandomWalkDriver
Enter grid size:6
Enter random seed (0 for no seed):1234
[0,5] [0,4] [1,4] [1,3] [2,3] [2,2] [3,2] [4,2] [4,1] [5,1] [5,0]

For the same grid size and same seed, we should get the same output even though it is random!
java RandomWalkDriver
Enter grid size:6
Enter random seed (0 for no seed):1234
[0,5] [0,4] [1,4] [1,3] [2,3] [2,2] [3,2] [4,2] [4,1] [5,1] [5,0]

With a random seed of 0, we don't supply a seed to Random so the walk will be different each time we run the program
java RandomWalkDriver
Enter grid size:8
Enter random seed (0 for no seed):0
[0,7] [1,7] [1,6] [1,5] [2,5] [2,4] [3,4] [4,4] [4,3] [5,3] [5,2] [6,2] [6,1] [6,0] [7,0]

java RandomWalkDriver
Enter grid size:8
Enter random seed (0 for no seed):0
[0,7] [1,7] [2,7] [2,6] [3,6] [4,6] [4,5] [4,4] [4,3] [5,3] [5,2] [5,1] [5,0] [6,0] [7,0]

Now we show some error checking on the user supplied input
java RandomWalkDriver
Enter grid size:-5
Error: grid size must be positive!
Enter grid size:-6
Error: grid size must be positive!
Enter grid size:6
Enter random seed (0 for no seed):-233
Error: random seed must be >= 0!
Enter random seed (0 for no seed):-455
Error: random seed must be >= 0!
Enter random seed (0 for no seed):1234
[0,5] [0,4] [1,4] [1,3] [2,3] [2,2] [3,2] [4,2] [4,1] [5,1] [5,0]
```

Using the RunUnitTester class for testing methods

When you are done, you can run the `RunUnitTester` to check if your methods meet all the requirements. It will execute several tests on your `RandomWalk` class and provide "PASS" or "FAIL" output with feedback.

You can also use the provided `RunGUITester` class to see the results of your program in an animated form. Note that the `RunGUITester` class expects your `RandomWalk` class to have properly implemented `step()`, `isDone()`, and `getPath()` methods. The image shown at the beginning is a screenshot of the `RunGUITester` program. You don't have to do anything to make the GUI work except implement the methods in the `RandomWalk` class as specified above.

Please note that the `RunGUITester` uses command line arguments so to use it, you must follow the instructions (which we will go over in class) on how to run in VS Code with command line arguments.

Submitting the Lab

Compress the submission files (see below) into a .zip file named: **Project02_*LastName_FirstName*.zip** When you are done and ready to submit, submit the following files in the .zip file:

- `RandomWalk.java`
- `RandomWalkDriver.java`