Lab 9: More Object Oriented Design

Let's get started!

Today we'd like you to:

1. **Open Eclipse**. Remember to keep all your projects in the same workspace. Think of a workspace as a shelf, and each project is a binder of stuff on the shelf. If you need some help organizing your workspace, we'd be happy to help!
2. **Create a new Java project**. Call it something like COMP 1510 Lab 9.
3. **Complete the following tasks**. Remember you can right-click the src package in the Eclipse package explorer project to quickly create a new Java class or interface.
4. When you have completed the exercises, **show them to your lab instructor**. Be prepared to answer some questions about your code and the choices you made.
5. **Remember that for full marks your code must be properly indented, fully commented, and free of Checkstyle complaints**. Remember to activate Checkstyle by right-clicking your project in the Package Explorer pane and selecting Checkstyle > Activate Checkstyle.

What will you DO in this lab?

In this lab, you will:

1. recognize when a Java class implements a common Interface from the Java library
2. implement some common interfaces from the Java library
3. model a simple robot that walks in random ways
4. write client programs that use the simple robot in different experiments
5. model a simple telephone with JavaFX
6. examine the static keyword and how it can (and cannot) be used in our code.

Table of Contents

1. Common Java interfaces

We've already encountered some interfaces in our Java studies. Open these webpages, and review the Javadocs for these interfaces:

1. We can implement a JavaFX component's event handler by using a separate class that implements the EventHandler interface (Chapter 4) docs.oracle.com/javase/8/javafx/api/javafx/event/EventHandler.html .
2. The Iterator interface (Chapter 5 and 7) defines the methods an iterator object must possess. Recall from Chapter 5 and lab 8 that an Iterator is an object that lets us iterate through and possible delete the elements in a Collection like the ArrayList docs.oracle.com/javase/8/docs/api/java/util/Iterator.html .
3. The Iterable interface (Chapter 6) works closely with the Iterator interface. Something that implements the Iterable interface can be used with a for-each loop, and can create an Iterator for itself docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html .

INTERESTING FACTOID: a lambda expression can be used anywhere a functional interface can be used. A functional interface is an interface with one method. The EventHandler interface is an example of a functional interface.

4. We've also encountered the Comparable interface in an indirect way. The String class implements the Comparable interface. In fact many classes from the Java library implement the Comparable interface docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html .

   Anything that implements the Comparable interface actually implements the Comparable**<T>** interface where the T is replaced with the class name. We have to specify the kind of object being stored in an ArrayList, like an ArrayList<String>. This is the same thing: we also have to specify what is being compared when we implement the Comparable interface. For example, String implements Comparable<String>.

2. Let's implement Comparable and use it to sort a list of Names

1. Create a package called ca.bcit.comp1510.lab9
2. Create a new class in this package called Name.
3. This Name contains a String for first, middle, and last.
4. This Name is immutable. There are no mutators, only accessors, and the instance variables are final.
5. Validate the parameters in the constructor. Nulls are accepted for middle names, but not first names or last names. No names may be empty Strings or Strings containing nothing but whitespace. Throw an IllegalArgumentException (see slide 30 in Chapter 7) if any parameter is invalid.

6. Name implements Comparable. Add the words **implements Comparable<Name>** to the Name class header. The compiler will remind you that when you implement an Interface, you must implements its methods. Go ahead and do it!
7. Scroll down and you will see a new empty compareTo method. Note its parameter is another Name object. Remember the compareTo contract: Return a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the object specified as the parameter. We want to compare names or sort names alphabetically, so we can use the Strings's compareTo method. A good algorithm could be something like this:
    a. Compare the two last names. If they're not the same, return the value returned when you invoke the compareTo method on the last name instance variable, passing the Name parameter's last name as the parameter.
    b. If the two last names are the same, compare the two middle names. If they're not the same, return the value returned when you invoke the compareTo method on the middle name instance variable, passing the Name parameter's middle name as the parameter.
    c. If the two middle names are the same, the we must compare the two first names. Use the same method described above to compare the Strings.
8. Create a Driver class with a main method.
9. In the main method, create some names, and add them to an ArrayList<Name>.
10. Here's the neat part. Make sure you have some Kleenex because you will surely spill your coffee when you see how amazing this next bit is.
11. The ArrayList class has a sort method. Usually we pass an instance of a class that implements the Comparator interface. We don't have to do this though if we are storing stuff in the ArrayList that implements Comparable.
12. Invoke the sort method on your full ArrayList<Name>, and then print out the results.
13. You may now wipe up your spilled coffee.

3. A simple robot that walks randomly

Let's have some fun. Let's design and implement a simple robot to model a random walk. We will then write two client programs that use it for two different things.

A random walk is basically a sequence of steps in some (possibly enclosed) space where the direction of each step is random. The walk terminates when a maximum number of steps has taken place or when the random walker steps outside of the space.

Random walks can be used to simulate things like the movement of molecules and economic phenomena like stock prices.

We will assume our RandomWalker is walking on a two-dimension square grid. In fact, let's assume our simulations takes place on a square grid with the point (0,0) at the center. The boundary of the square will be a single integer that represents the maximum x- and y-coordinate for the current position on the square (so for a boundary value of 10, both the x-

and y-coordinates can vary from -10 to 1, inclusive). Each step will be one unit up, one unit down, one unit to the left, or one unit to the right. (No diagonal movement.)

1. Create a new class called RandomWalker. It should have the following instance variables:
    i. The x coordinate of the current position
    ii. The y-coordinate of the current position
    iii. The maximum number of steps in the walk
    iv. The current number of steps taken
    v. The boundary of the square the walker inhabits
2. The RandomWalker has an overloaded constructor. There is more than one constructor:
    i. The first version accepts two integers and assigns the first to the maximum number of steps in the walk, the second to the boundary, and sets the other instance variables to zero
    ii. The second version accepts four integers representing the maximum number of stps in the walk, the x-coordinate, the y-coordinate, and the boundary, and assigns these to the corresponding instance variables. The other instance variable should be set to zero.
3. The RandomWalker has a toString that prints out the coordinates and the current step number.
4. Create a driver class called TestWalker which contains a main method.
    i. Declare and instantiate a RandomWalker with maximum steps 10 in a square of boundary size 5 (so a square that's 10 x 10 with the RandomWalker in the middle).
    ii. Ask the user for values for maximum steps and boundary size an create a second RandomWalker using those values.
    iii. Print out the toString information for each RandomWalker object. How does everything look so far?
5. Add a method to RandomWalker called public void takeStep(). This method should simulate taking a single step. Use a Random object to generate a random number with four values, (say 0, 1, 2, 3), and then use a switch statement to change the coordinate(s) of the object. Don't forget to increment the number of steps taken!
6. Modify the TestWalker method so each RandomWalker takes 5 steps. Print out each object after each step so you can see what is happening.
7. Add a method to RandomWalker called public boolean moreSteps() which returns true if the number of steps taken is less than the maximum number of steps.
8. Add a method to RandomWalker called public boolean inBounds() which returns true if the RandomWalker's current coordinates are within its square boundary (include the boundary on the square).
9. Add a method to RandomWalker called public void walk() that accepts no parameters and returns nothing. Its job is to use a loop to simulate a random walk. A RandomWalker should keep taking steps as long as:
    i. The maximum number of steps has not been taken
    ii. The RandomWalker is still in bounds.

10. In TestWalker, create a new RandomWalker with a boundary of 10 and 200 steps. Print it out, ask it to walk, and then print it out again. Run it more than once. Are the results what you expect?
11. It is often helpful to know how far an object is from the origin as it moves. Add an instance variable maximumDistance to the RandomWalker. This should be set to zero in each constructor.
12. Update the takeStep method so that it modifies the maximumDistance instance variable if necessary:
    i. The first stepAdd a private support method to do this. The private support method should be called private int max(int a, int b) and it should return the maximum of a and b.
    ii. Add code to takeStep to update maximumDistance. This can be done in a single statement using the max method. The new value of maximumMethod will be the maximum of either the current value of maximumMethod, or the current distance to the origin. Note that if the current location of the RandomWalker is (-3, 15) the distance to the origin is 15; if the current point is (-10, 7) the distance to the origin is 10. Remember that Math.abs returns the absolute value of a number.
13. Add an accessor method to RandomWalker for the maximumDistance.
14. In TestWalker, add statements to print out the maximum distance in the loop. Does it work correctly?

4. Application: simulating a drunk walker

Let's write a program that simulates a Drunken walker. Your program only needs a main method. The program will simulate a drunk staggering around on some square platform (imagine a square dock in the middle of a lake). The goal of the program is to simulate the walk many times and see how many times the drunk falls off the platform (goes out of bounds). Your **DrunkWalker** program should:

1. Ask the user for the boundary and the number of steps
2. Ask the user for the number of drunks (simulations) to run
3. Use a for loop to instantiate a RandomWalker to represent the drunk and ask it to walk. If it falls out of bounds early, increment a counter.
4. After the loop, print out the total number of tests and the number of falls.

5. Application: Collisions!

This next program will simulate two particles in a space and how many times they collide, i.e., occupy the same coordinates. The goal of the program is to determine and report the number of collisions. Your **Collisions** program should:

1. Contain a main method.

2. Inside the main method, reate two RandomWalker objects. Assume the space they are in is enormous, i.e., use a boundary of 2,000,000, and 100,000 for the maximum number of steps.
3. Start one particle at (-3, 0) and the other at (3, 0).
4. Use a for loop to have each RandomWalker take one step and then determines if they have collided. You should probably make sure your RandomWalker has accessors for its coordinates. To see if they have collided, let's add a static method to our Collisions program called public static boolean samePosition(RandomWalker one, RandomWalker two) that returns true if the two RandomWalker objects passed as parameters are in the same position.
5. The loop in the main method should check if the two particles are in the same position by using the static samePosition method.
6. What is the farthest distance a particle gets with these settings? Other settings?

6. Telephone

It's time for some JavaFX fun. Flex your programming fingers. You must create a JavaFX program that displays the number pad for a phone. It should look like a phone number pad (hint: use a GridPane). There should be a total of 12 buttons for the numbers 0 through 9, and don't forget the buttons for * and #. When a button is pressed, its value should be displayed on a label which is positioned above the number pad. Consider the following:

1. How can you use loops to create the Buttons on the pad?
2. Look for areas of duplicated code. We like to minimize the amount of duplicated code. Do you need a separate listener for each button, or can you use the ActionEvent's getSource method (check out Chapter 5's RedOrBlue.java example). Do we even need to use getSource?
3. Does it make sense to use lambdas here? What about a class that implements the EventHandler?

7. You're done! Show your lab instructor your work.