CSC/CPE 203

Lab 1- Java Collections, Control Structure, Classes, Testing, and UML

Due: 9/23

Orientation

This lab is a further introduction to Java. This lab has three parts. Part 1 uses Java control structures as related to some Java collections. Part 2 introduces class structure and definition. Part 3 introduces UML, and the tool we will be using to create UML documents. Those already familiar with Java should finish the lab and then offer to help others.

Objectives

1. To be able to iterate through different Java collections with various control loops (part 1)
2. To be able to write your own class with specified methods (part 2).
3. To be able to use unit testing, including writing your own tests for your code (part 1 and 2)
4. To be able to write a very simple UML diagram of a class using yEd (part 3)

Resources

- Your peers
- Your instructor and TA
- The Java Standard Library - Bookmark this web page, or download a copy. You will be using the Java Standard Library documentation regularly this quarter. You should bookmark this page rather than doing google searches for documentation. A google search will often give you information about the wrong version of Java, inappropriate advice, or bad advice.

Retrieve the files provided for this lab in the PolyLearn

Building and Running Programs

For this lab, you are again required to compile and run Java on the server from the command-line.

To compile multiple Java files, one can run the following command:

```
javac *.java
```

NEW For this lab, we will be using the JUnit library to help us test the correctness of our code. Thus, you will need to tell the compiler where the JUnit library is. There are three ways to do this, with the second being the better.

1. The first includes the library in the "class path" for the given call to the compile. This must be repeated each time you compile the files and when you run the test cases (see below).

```
javac -cp /home/seinakia/www/junit-4.12.jar:/home/seinakia/www/hamcrest-core-1.3.jar:. *.java
```

```
java -cp /home/seinakia/www/junit-4.12.jar:/home/seinakia/www/hamcrest-core-1.3.jar:. org.junit.runner.JUnitCore
TestCases
```

2. Instead, it is much more convenient to configure the system to add the library to your class path each time you log in. Add the following line to your .mybashrc file in your home directory.

```
export CLASSPATH=/home/seinakia/www/junit-4.12.jar:/home/seinakia/www/hamcrest-core-
1.3.jar:${CLASSPATH}:.
```

```
alias javat='java org.junit.runner.JUnitCore'
```

The alias name can be anything you want. Here I used javat, choose whatever name make more sense. After doing so (and either restarting your shell/terminal or logging out and back in or typing "source .mybashrc"), you need only execute the following to compile and run.

```
javac *.java
```

```
javat TestCases
```

3. In .mybashrc file in your home directory do not use alias and recall the JUnitCore anytime you want to run the test.

```
export CLASSPATH=/home/seinakia/www/junit-4.12.jar:/home/seinakia/www/hamcrest-core-
1.3.jar:${CLASSPATH}:.
```

```
javac *.java
```

```
java org.junit.runner.JUnitCore TestCases
```

Part 1: Java Control Constructs and Collection Introduction

In the provided `part1` subdirectory, edit each of the given files to complete the implementations and tests. Comments in each file will direct the edits that you are expected to make. Look up any classes you don't know how to use (such as `Map`) in the Java Standard Library.

It is recommended that you use the order of the unit tests in `TestCases_P1.java` as a guide for the order in which to complete the methods. This order corresponds to the increasing complexity of the methods. A good process to follow would be to read the first group of tests in `TestCases_P1.java`. Edit the corresponding class to make the code pass the tests. *Run* the tests to make sure they now pass. Add the required additional test case. *Run* the tests again. Then move on to the next group.

The code uses a number of classes in the Java Collections Framework. You can tell what they are called by looking at the source. For example, `ExampleMap.java` uses `java.util.LinkedList`, `java.util.List` and java.util.Map. Java does support a "wildcard import" statement ("import java.util.*;") that imports all of the classes in a given package. Most good style guides recommend against using these wildcard imports. Importing each class individually might be more typing, but it makes your code easier to read, and that's almost always much more important than a slight time savings.

(Note For Map: As a Map is not a true collection, there is no direct method for iterating over a map. Instead, we can iterate over a map using its collection views. Any Map's implementation has to provide the following three Collection view methods: keySet(), values(), and entrySet(). More on: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Map.html)

Do NOT google for the documentation of these classes. Instead, get in the habit of reading the official documentation that you bookmarked earlier. To find java.util.List, for example, click on java.util in the Packages pane, and then List in the pane under that. It's OK if you don't understand all of the methods in every class, but you should be able to find what you need fairly quickly.

After each small change, re-run the program. You should get in the habit of compiling and running your program at least once every five or ten minutes. That way, most errors you find will be in something you have done recently, and still remember.

The output for a failed test can be verbose - a stack trace is printed. You can see the specific error at the numbered line above each stack trace. It's generally a good idea to fix the code one file at a time, re-run it after each fix to make sure that the number of errors is decreasing.

Note: You will learn syntax is using in TestCases in the near future. For now, just know that you can list tests to be run using this syntax same in part 2.

Part 2: Classes and Objects

For this part you will implement a simple class (named Point) representing a point in two-dimensional space. Do this by creating a new file named: Point.java and write code for it as specified below. This class must include a single constructor, and support the following operations:

- public Point(double x, double y)
  This is your Constructor.
- public double getX()
  Returns the x-coordinate of this point.
- public double getY()
  Returns the y-coordinate of this point.
- public double getRadius()
  Returns the distance from the origin to the point.
- public double getAngle()
  Returns the angle (in radians) from the x-axis in the in range of *-pi* to *pi*.
- public Point rotate90()
  Returns a newly created Point representing a 90-degree (counterclockwise) rotation of this point about the origin. This should be counterclockwise when pictured with normal

mathematical axes, where x increases to the right, and y increases as you go up. Consider drawing a picture for a point not on an axis as a guide for how you might implement this operation (hint: there is a solution that does not require any sophisticated computations).

Update the given TestCases_P2.java file, in the part2 directory, to verify that your implementation is working. **You will be asked to show a test case for every method you wrote.** You will see some existing test cases that are used to verify some design/style aspects of your implementation, which you do not need to change or modify.

Note that you are not expected to define equality for Point objects at this time; as such, you will need to test the components (x and y) for the expected values. You may write equals() and toString() methods for Point if you know how. (We will discuss these methods in near future.)

Part 3: UML

In this part you will create the UML diagram for your Point, using yEd Graph Editor. See the pdf description of the UML portion of this lab. Store your diagram in a file called point.graphml.

Demo

Demo your work.

Submission

Submit your code for Parts 1 and 2 in the Canvas.

Rubric:	**Points**
1) UML diagram	6
2) Test P1 (16 tests)	16
3) Test P2 (9 tests)	9
4) Extra test cases P1	24
5) Extra test cases P2	15
6) Submission	30