Lab 03: Introduction to Object Oriented Programming

COSC 102 - Spring '19

This lab is an introduction to Object-Oriented Programming (OOP). The goals are for you to

- define and test object code,
- declare, define and use instance variables (aka fields), methods, and constructors and
- understand the difference between object definition code and client code.

Unzip the `lab03` folder, where you will find two directories: `bank_acct` and `point`. At the **jEdit** console in either directory (remember the command `cd $d`, which moves you to the working directory) you can compile all the `.java` files in the directory with: **`javac *.java`**

To run your code, you will run a particular `main` method defined in a compiled `java` source file. In Part 1 for example, you will execute your program by typing in the console:

```
java PointClient
```

since it has a `static main` method.

1 `Point.java` and `PointClient.java`

In the `point` directory there are two files.

- The **`Point.java`** file contains the object definition code for a `Point` object, which is a coordinate location represented by an `x` and `y` value. As-is, this code will compile, however there are syntax and logic errors in the functions you will uncomment in the steps below.
- The **`PointClient.java`** file contains client code which instantiates `Point` objects and calls their instance methods. In the steps below, you will need to add code to `PointClient`'s `main` method to instantiate `Point` objects and evaluate your progress. As-is, this code will compile.

1.1 Warm-up

Follow the steps below to correct and complete the code that defines a point object, `Point.java`, and test it in its client code, `PointClient.java`. Be sure to compile and run often, minimally after each step to check you have done the appropriate modifications. You should not tackle the next step without being sure your program compiles and runs correctly.

1. Begin by reviewing the code in both `Point.java` and `PointClient.java`. Both of these files should compile. Trace the two uncommented lines in `PointClient`'s `main` method, then run the code. Did the output match your expectations? If not, re-read the appropriate `Point.java` code and ask your TA/instructor.
2. Find the commented out function `translate` in `Point.java`. Read the comments to understand the goal of this function, then uncomment the function. As-is, this code has syntax error(s) and will not compile. Correct the error(s) so the function works appropriately. When ready, test your `translate` function, following the comments in `PointClient` labeled for *2.1.2*.
3. In `Point.java`, add a constructor that takes two integers. Test this by instantiating the two `Point` objects `p1` and `p2` described in the client code (line 32). If the client code **doesn't** compile, think about why and tell the lab instructor/TAs the reason.

4. In `Point.java`, you now need to add the parameterless constructor. Compile and test Steps 2 and 3.

5. In `Point.java`, review and uncomment the `distanceFromOrigin` method. There is a logical error here in the computation you need to correct. Use `PointClient.java` to create `Point` instantiations and test your method.

1.2 Instance methods

Next, add the following instance methods to your `Point.java`:

1. ****Before you write any code**** read the details for the two functions outlined below, and then **write a series of test cases** for both functions in your `PointClient.java`. You may comment these tests out until you're ready to run them.

2. `public int quadrant()`

   Returns which quadrant of the *xy*-plane this `Point` object falls in. Quadrant 1 contains all points whose x and y values are both positive. Quadrant 2 contains all points with negative x but positive y. Quadrant 3 contains all points with negative x and y values. Quadrant 4 contains all points with positive x but negative y. If the point lies directly on the x and/or y axis, return 0.

3. `public void flip()`

   Negates and swaps the *xy*-coordinates of the `Point` object. For example, if an object `pt` initially represents the point $(5, -3)$, after a call of `pt.flip();`, the object should represent $(3, -5)$. If the same object initially represents the point $(4, 17)$, after a call to `pt.flip();`, the object should represent $(-17, -4)$.

You will be graded on the quality of your test cases so be sure to cover a variety of scenarios. Do not delete your test cases from your `PointClient` code (though you may comment them out if you like).

2 BankAccount.java

Next, navigate over to the `bank` directory and open the single file, **`BankAccount.java`**. This class will contain both the object code definition for a `BankAccount` and a `main` to test this definition. Currently, each `BankAccount` consists of a name the account is associated with, and a balance. This time, you are responsible for testing your code frequently as you are writing it (**our course rule you need to adhere to!**). Complete the steps applying this good programming practice.

1. Define a `main` method for `BankAccount.java` where you instantiate a default `BankAccount` object. Make sure your code compiles. You should run into one issue here, fix it!

2. Next, implement a constructor for `BankAccount` which accepts a single argument. Think about what this argument needs to be. Hint: we'll say that all new bank accounts start with no money in them – new account holders will need to deposit their starting balance (and we're not in the habit of giving away money!)

3. Add a `toString` method to the `BankAccount` class. Your method should return a `String` which is the account's name and the balance separated by a comma and space. For example, if an account object named `sue` has the name `"Suzanne"` and a balance of 17.25, `sue.toString()` should return: `"Suzanne, $17.25"`

4. Add an instance variable named `withdrawFee` to `BankAccount`. This field represents an amount of money to deduct every time the user withdraws money. The default value is $0.00. Note that this fee is only deducted for withdrawal, but not for deposits.

5. Since you want your newly created `withdrawFee` to be changeable by the client code at any time, write the mutator method `setWithdrawFee`.

6. Add a second constructor for your `BankAccount` which accepts a name as well as an initial value for the withdrawal fee, in that order. Once you have this constructor implemented, take a look at your two constructors, and make sure you have no redundancy between the two *(hint: you can redirect the constructor code execution to another constructor using a special keyword; open your textbook if necessary!)*

7. Update the **`deposit`** method to now return a **`boolean`** indicating whether or not a deposit is successful. If the deposit amount is negative or zero, the deposit is considered unsuccessful – the function would then return `false` and **not** modify the account values.

8. Update the **`withdraw`** method to now return a **`boolean`** indicating whether or not the transaction is successful. Withdrawal transactions are only allowed if the account balance remains positive. Don't forget to take into account the withdrawal fee. As above failed withdrawals should **not** modify the account values.

9. Lastly, add a class method to your `Bank Account`: **`public static BankAccount merge(BankAccount b1, BankAccount b2)`**. This function accepts two `BankAccount` objects as arguments, and merges them into a single account. More specifically, the balance of the merged account will be the sum of the two accounts, and the name on the new account will be a compound of the names on the two merged accounts. For example, the merged account of *Jane Doe* and *John Smith* would have a name of *Jane Doe and John Smith*. Lastly, the withdrawal fee of this new account will be whichever withdrawal fee of the two accounts being merged is **lower**.

   Note that this function **does not modify either argument account**. Instead, it returns a third, brand new `BankAccount` object with all of the merged attributes.

3 Submission

Upload all your completed files to the **Lab 03** submission link on the Moodle page. Your submission should include the following three files:

- `Point.java`
- `PointClient.java`
- `BankAccount.java`

Be sure to submit only your `.java` files and **not** your `.class` files – the `.class` files contain the compiled byte-code and if you submit them we will not be able to grade your work!

This assignment is due for all lab sections on **February 11th** at **10:00PM**.