CS 455 Lab 3: Implementing classes

Spring 2019 [Bono]

Goals and Background

This lab will give you some practice working with a class implementation, a test driver, and dealing with numeric computations. *A note about this lab: Because this lab is a little long for the two-hour period (especially for less experienced students), we decided to make the lab only out of 3 points, even though there are four exercises here: so it's possible to earn 4 points on this 3-point lab, but you should consider it completed if you do the first three exercises. The fourth one will be to give some students an extra challenge (definitely more of a challenge than the bonus problem on lab 2). You cannot get credit for Exercise 4 unless you have already completed exercises 1 - 3.*

This lab is more difficult than the other ones we have had so far; to enable you to get it done in the two hour period we recommend you do some advanced preparation. Suggested preparation:

- Make sure you have done all the assigned readings from the text, especially the sections mentioned below.
- Make sure you understand what the `CashReg` class in Exercise 1 does and how it works (i.e., its implementation). The comments at the start of the file should be helpful for understanding its intended use. The two textbook sections listed below that mention the Cash Register offer a more detailed explanation,
- You may want to even do Exercise 1 ahead of time, or at least the first, part through question 1.1.
- Read over the whole lab so you can start thinking about how to solve the other parts of the lab.

Remember to switch roles with your partner from what they were last time. For more information on lab partnerships, see Lab 2.

Reading and reference material

- Horstmann, Ch. 3, Implementing Classes.
- Horstmann, How To 3.1, Implementing a Class (Simpler Version of Cash Register Example is first presented here)
- Horstmann, Section 4.1.1, Number types (Discusses roundoff errors)
- Horstmann, Section 4.1.2, Constants (Enhancement of Cash Register -- this is basically the version in this lab -- minor differences described in the comments in `CashReg.java` for this lab)
- Horstmann, Section 4.2, Arithmetic
- Horstmann, Section 4.2.5, Converting Floating-Point Numbers to Integers
- Horstmann, How To 4.1, Carrying Out Computations (Vending machine example)
- Horstmann, Section 5.1, `if` Statements

Exercise 1 (1 checkoff point)

For this lab we have provided you with some starter files in Vocareum. There's a separate subdirectory with the starter files for Exercise 4 (called ex4), because that involves different versions of the same classes.

This is a slightly modified version of the cash register class and test program developed in Section 4.2 of the textbook. (We just added one accessor method, some comments, and changed the name of the class slightly.) The two files we will be using are:

- **`CashReg.java`** A cash register class. We added comments at the top of the class file to show how you might use it.

- **`CashRegTester.java`** A test program for the cash register class. This one has a `main` method.

Compile and run the program to see what it does. You can compile and run a multi-file program such as this from the shell using a wild-card on the command line. This will compile all the .java files in the current directory.

```
javac *.java
```

Question 1.1.. The first set of tests has a sequence of two purchases recorded, a payment given, and then change given for that payment. Show a formula that describes how the expected result for the change given was arrived at for this first sequence. Show all purchase amounts and the payment amount in this formula. You will need to look at the code in `CashRegTester.java` to do this.

Following the conventions of output for the other tests already in `CashRegTester.java`, add a new test *before* the other tests in `main` with the following data:

> the customer purchases one item for $4.35, and then pays with a 5 dollar bill, and the change is given.

Question 1.2.. You should have gotten a round-off error in your results. Why do you get such an error? (You don't have to do any computations, just explain the issue.) Hint: this is discussed in section 4.1.1 of the textbook.

We can save the output from a run by redirecting it to a text file instead of sending it to the monitor. This is a feature of the Unix shell. You do it as follows:

```
java CashRegTester > ex1.out
```

Then you can examine the output at your leisure:

```
more ex1.out
```

Save the output of exercise 1 as described above so we can compare it with results in the next exercise.

To get checked off, show the TA your `CashRegTester.java`, `ex1.out` and your answers to the questions.

Exercise 2 (1 checkoff point)

Note: this lab exercise involves more thought and more code than other ones we have done. Because of some confusion about this exercise in past semesters, we have but some key points in **bold** below.

You will have multiple versions of this code floating around for this lab; because the class names are required to correspond to the file names in java programs, it will be much easier to keep track of the various versions if they are each in a separate directory. So, make a subdirectory called `ex2`. Copy both of the files you used for exercise 1 into that subdirectory (i.e., the version of the code after we did exercise 1).

Note: this and the following exercises will only involve making changes to `CashReg.java`; we are done making changes to `CashRegTester.java`.

WARNING: The classes and files will have the same name for every version in this lab so we don't have to keep changing the class names, so make sure you are editing the right version in Vocareum (by looking at the file listing on the left and what folder you are in).

One of the advantages of encapsulation (a.k.a., information hiding) is that you can change the internal representation of a class, and if the interface is unchanged, all of the client code will still work, unmodified.

Also, you will be able to test the new version of the class without modifying the old test program.

In this exercise we are going to make such a change to the internal representation of `CashReg`, without changing the interface at all. In fact, once we are done we should get the exact same results as the old version of `CashReg`, except for the problem from exercise 1.

The current version of the `CashReg` class stores all amounts in *dollars* using `double` variables. Change the implementation so that is uses amounts in *cents* instead (`int`). However, *do not change the interface for the class*; i.e., `recordPurchase` and `giveChange` still take parameters of type `double`, and `getTotal` returns a `double`. And these `double` values still signify some dollar amount (e.g., $2.27). Internally storing everything as cents (`int`s) may make some computations easier and might enable us to fix the rounding error.

Hints:

- **This is not just changing the types of a few variables, but will involve changing several parts of the class implementation, similar to when we made an enhancement to the `Student` class in lecture.**
- you will want to change the named constants used as well. You should also add a constant called `DOLLAR_VALUE`.
- if you get a compile error about losing precision from doing a computation with an int and a double, you can get it to compile by casting the result to be an int. See section 4.2.5 for how to type cast. Go ahead and do this.

When you run the new version of your code, you can see if it does the right thing by comparing the results to the old version. If you save your output to the file `ex2.out` (exercise 1 showed how to do this), you can compare the results using the Unix `diff` command:

```
diff ex2.out ../ex1.out
```

You should get the same exact output *except* for the test with the rounding error from exercise 1.

Question 2.1.. What results do you get for the test we added in exercise 1. Is this the right answer?

To get checked off, show the TA your code, your working (modulo the issue from question 2.1) test program and the answers to the questions.

Exercise 3 (1 checkoff point)

Like last time, make a subdirectory of your `work` directory called `ex3`. Copy your code from the `ex2` subdirectory into this new subdirectory.

Question 3.1.. Stating what directory you are starting from, show a single Unix command to copy the files as described.

Depending on the changes you made in the last exercise, the first test may have gotten worse results than we got in exercise 1. (If you already have the result that matches the expected result, you may not need to make any changes to your code here.) In this exercise we are going to fix the results.

Fix `recordPurchase` so it records the correct amount for values such as the one given in the first test in `CashRegTester`. (So, just to review, we are talking about the version of the code that stores the necessary values in cents, rather than dollars.) When you run your program, save the results in `ex3.out`.

Hint: the `Math.round` function can be helpful here. It takes a `double` as a parameter (give it a double value). We want to use it such that the resulting rounded value will be the correct `int` for the value in cents.

To get checked off, show the TA your code, your working test program and the answers to the question. This version of the program should get the correct answer for all of the tests.

Since you may not have time to complete the optional Exercise 4, please get Exercises 1 - 3 checked off before proceeding.

Exercise 4 [optional] (1 checkoff point)

Get into the `ex4` directory we provided you. The files in that directory are:

- `Change.java`
- `ChangeTester.java`
- `CashRegTester4.java` (A tester program just for `ex4`)

Copy just your `CashReg.java` file from exercise 3 to `ex4`. You will not be using the tester from the previous exercises here (use `CashRegTester4` instead).

In this exercise we are going to improve the cash register so it can tell the retail salesperson exactly how to give the change (i.e., how many dollars, quarters, dimes, etc.), not just how much the total is for the change to return. This will involve modifying the `CashReg` method `giveChange`. Read this section before jumping into that task: you should follow the steps below for more information about how to make this modification, using a class we already wrote for you to help out.

In other programming languages there is a way to return multiple values back from a function by using reference parameters: Java does not have reference parameters (we will be talking more about this in lecture soon). So if we wanted to change the `giveChange` function so it gives us back various values, one way to do so would involve turning it into several functions that each return one of the values.

Instead, here we're going to keep `giveChange` as a single function, but return multiple values by packaging them into one object.

We wrote a class for you called `Change` to represent some amount of change (i.e., dollars, quarters, dimes, etc.) that might be in your pocket. (Note: its test program is called `ChangeTester`). You will be modifying the `CashReg` class so `giveChange` will return an object of type `Change`. Also, `receivePayment` will now take take a `Change` argument to simplify it (the old version took 5 parameters, one for each denomination of coin plus dollar bills.) The rest of this section goes into more detail on that new interface and steps for incorporating those changes.

You won't have to modify `CashRegTester4` to use the new interface to the `CashReg` class and otherwise make use of the `Change` class, because this version already does that for you (it has the same tests as the old version).

Look at the new `CashRegTester4.java` to see how it has changed. In particular, look at the calls to `receivePayment` and `giveChange` and compare them with the old version.

To start the modifications to `CashReg.java` first just make a stub version of `giveChange` that returns a dummy `Change` value (one with all zeroes is a good option), and a stub for `receivePayment` with the correct interface. Since the `receivePayment` method doesn't return a value, its stub will have an empty method body. The stub functions are:

```
public void receivePayment(Change money) {
    // empty method body
}
```

```
public Change giveChange() {
    return new Change();  // return 0 change object
}
```

Compile and run this new version with the new test program. (This version, of course, will not get the correct results yet.) Save your results of this run into a file called `ex4stub.out` and take a look at them.

Question 4.1. What parts of the results don't match the expected output?

Now, we'll tackle modifying the program so that it gives the correct change. You may assume the cash register has unlimited bills and coinage for making change. Consequently, when someone calls `receivePayment`, you don't have to save the information about what exact coins they used to pay. Modify `receivePayment` so it works correctly with the new parameter.

Now go ahead and modify `giveChange` so that it gives the correct change. When you have it working, save the output in a file called `ex4.out`. Since the output of this test program is somewhat different than in the other versions, you will not be able use `diff` to compare it with results for the other parts of the lab, but you should examine it to make sure it's computing the results correctly.

Hint: The vending machine example described in *How To 4.1* in the textbook may be helpful here.

To get checked off, show the TA your code, your working test program and the answers to the question.

Checkoff for DEN students

When you click the `Submit` button, it will be looking for and compiling (for source code) the files
`README`, `ex1.out`, `CashRegTester.java`, (in your home directory)
`ex2/CashReg.java`, `ex2/ex2.out`,
`ex3/CashReg.java`, and `ex3/ex3.out`.
You do not have to delete the other .java files in those directories. It won't check for the `ex4` files (but we will get them for grading if they are there). As usual, the lab us due by 11:59pm on Sunday Pacific Time at the end of the week for this lab.
