Homework 4

Thomas Lilly

1 Introduction

Now that you have a strong foundation in the Java basics, this week's homework will be about using Object Oriented Programming to model a business's financial transactions. Please read the entire document and **do not forget to put the collaboration statement at the top of your submission!**

2 Problem Description

In order to test systems and new features, businesses will often run simulations to verify they will work as expected. You and your friends think this is a niche market that you would like to break into with a startup. As the first step of creating your new business, you need to create a prototype of your product that will model basic transactions of people buying items from a business. After simulating these transactions you will generate a well formatted report to view the results of the simulation. For simplicity, we will only be simulating one business, but the number of people and number of days will be specified via command line. Without arguments, the program should run with default values which are specified later. *NOTE: If the program only receives 1 of the 2 arguments, it should use the specified value and the remaining default value.*

- Running the program with both command line arguments. A '-p' will denote the number of people and a '-d' will denote the number of days.

```
java MarketSim -p 5 -d 6

Running simulation with 5 people over 6 days...

Simulation Report: Amazon
Execution time: 0.12ms
========================================
Days of simulation:            6
Total Transactions:            21
Total Revenue:                 $4597.14
Number of Items in stock:      1080
Best selling Item:             "Jaybird X2"
========================================
```

- Running the program without command line arguments

```
java MarketSim

Running simulation with 5 people over 31 days...

Simulation Report: Amazon
Execution time: 0.34s
========================================
Days of simulation:            31
Total Transactions:            121
Total Revenue:                 $22511.27
```

```
Number of Items in stock:       787
Best selling Item:              "USB Micro-USB to USB Cable"
==========================================
```

- Running the program with incorrect arguments
  The command `-z` is not known for our program. When someone inputs an incorrect argument we need to inform them of the correct way to use command line arguments, such as calling `-p <numberOfPeople>` where `<numberOfPeople>` is an integer, and then end our program.

```
java MarketSim -z incorrect

Usage: java MarketSim [-p <numberOfPeople>] [-d <numberOfDays>]
```

3 Solution Description

In this model you will create `Person` objects to simulate financial activity over a period of days specified via command line arguments. The simulation will loop for the given number of days and output a report of the simulated activity. The simulation will require four classes: Person, Business, Item, and MarketSim. MarketSim will contain the main method that runs the simulation using the other three classes.

3.1 Item Class

1. Private instance data
   (a) A `name` for each Item
   (b) The number of available items in stock
   (c) The number of times someone has attempted to purchase this `Item`
   (d) The `price` of the `Item`
   (e) Any other instance data you need to complete the required methods throughout the assignment.
2. Constructor
   (a) The item constructor should take in a name, quantity, and price as parameters.
3. Public methods
   (a) Getters and setters for private instance data that makes sense. Think critically about this and what data you need to complete the assignment.
   (b) A method **`increaseNumSold`** that takes in a parameter representing how many items were sold. This method increases the sold number by the appropriate amounts.
   (c) An **`equals`** method. This method should say that two instances of Item are equal if the have the same name and the same price.
   (d) A `toString` method that returns the item's name.

3.2 Person Class

1. Private instance data

   (a) The `name` of the person. Preferably each Person has a slightly different name, but that's not the point of this assignment. For example, in my solution I have each person's name being "personN" where N is a random number. You can do as you like for this - whether it's a predetermined array of names or something weird like I did.

   (b) A variable called `balance` that represents the total amount of money a person has on them. Think critically - what type should this variable be?

2. Constructor

   (a) The Person constructor should have a parameter for the person's name and amount for the starting balance. Both of these values should be stored.

3. Public methods

   (a) Getters for the Person's `name` and `balance`.

   (b) A **`purchase`** method which attempts to purchase an `item` by decreasing the cost of the item from the Person's `balance` and returns a `boolean`. If the amount to be withdrawn would cause the balance to go below zero, then the purchase should not succeed, otherwise it should update `balance` appropriately.

   (c) A **`deposit`** method which should take in a double and add it to the Person's balance. This method should only accept amounts that are positive.

3.3 Business Class

This class will contain data and methods that will allow people to interact with businesses. There is a provided variable **`businessData`** which will contain the business's current inventory.

1. Private instance data - again, think critically about what these types should be.

   (a) An array of `Items` called `inventory` that represents the items that are available for purchase. The default size of this array should be 10. If the array needs to grow due to the size of input you should double the array's size. Remember that to do this you need to do some array manipulation since you can't just double the size of an array after it has been created. Make sure you don't lose any of the inventory (that is, you copy all of the inventory to the new array)!

   (b) The name of the business

   (c) Total sales during the simulation

   (d) The total number of transactions made during the simulation. A person can buy multiple items in a single transaction.

   (e) You are **NOT** allowed to have a two dimensional array as private instance data for the business.

2. Constructor

(a) The constructor should have a two dimensional String array parameter. An example of this array will be in the provided MarketSim.java file for you to test your code with. The first row contains the name of the business and the second row to the last row contains 3 columns with the item name, price, and quantity in that order. You can assume that this is how arrays of this kind will always be set up and that the businesses will always have something in their inventory.

(b) The constructor should use the two dimensional array to store appropriate information about the business. Consider using some of the methods described below to help accomplish this.

3. Public methods

(a) A getter for the business's `name`.

(b) A **`getBestSellingItem()`** method which returns the best selling `Item` in the `inventory`. A bestselling item is defined as one that has been *attempted to be sold* the most. If there are multiple bestselling items it doesn't matter which one you choose to display as the best seller. Any of the items will do.

(c) A **`sell`** method which takes in a `Person`, an `Item`, and a quantity(amount to sell). This method will handle the logic for completing a single transaction (sale) made by the business. This method should only complete successfully if the `Person` has enough money and there is enough of the `Item` in stock to handle the quantity desired. If either fails then this transaction should fail. If they succeed then the `Person`'s balance,the `Items` in stock, and the `Business`' revenue should update appropriately. Remember in both cases to still update the "attempted times sold" of that item appropriately.

(d) An **`addItem`** method which takes in an `Item` and returns whether it was successfully added. This `Item` is then added to the `inventory`. This method should not allow duplicate `Items` to be added, if a duplicate `Item` is input then it should not be added to the `inventory`. (Duplicate is defined as two `Items` being equal)

(e) A **`String getReport(int days, double execTime)`** method that returns a string containing the report that will be printed in MarketSim. View the provided output on the first page to see what should be included and how to format.

3.4 MarketSim Class

This class will be what's known as a "Driver". This is the class which uses your previously made classes to run the simulation. As shown in the Problem Description, `MarketSim` should take in the '-p' and the '-d' command line arguments which correspond to the number of `Person`(s) in the simulation and how many days(iterations) will be simulated. If the number of people is not specified, the default value should be 5. If the number of days is not specified the default value should be 31. Finally, the program should print the Business's report to the command line before it exits.

- If incorrect command line arguments are specified, you should print the proper usage of the program and exit without printing the Business Report.

- Examples of command line arguments that should work:

  java MarketSim

  java MarketSim -p 4

  java MarketSim -d 22345

  java Marketsim -d 5 -p 3

- Examples of command line arguments that should not work:

  java MarketSim -p

  java MarketSim -d

  java Marketsim -d 5 -p

  java MarketSim -d 4 -p 4 -c

  java MarketSim -zxvf

- Each `Person` should be instantiated with a name and a random integer amount between $50,000 and $100,000 (exclusive) using the `Random` class.

- Each iteration there should be a 75% chance that a `Person` will make a purchase. A single item should be selected randomly from the Business' inventory. After this, the quantity of that item to be purchased should be determined as a random integer between 1 and 5. This functionality should be provided using the `Random` class.

So the flow of your entire simulation should be as follows:

1. Create the correct number of people for the simulation.
2. Run a loop that iterates for the specified number of days (each loop through the iteration = 1 day)
3. In a day, each person has a 75% chance of making a single transaction.
4. If they make a transaction, then they randomly select one of the items from the business' inventory then randomly choose a number 1-5 of how many of that single item to purchase.
5. At the end of the simulation the business' report should be printed to the console.

NOTE: You should consider using a seed for the Random number generator in order to verify the correctness of your program. A seed forces the Random number generator to begin with a certain number, not necessarily the seed, so that consecutive tests will produce the same random numbers.

3.5 Important Notes, Tips, and Tricks

- Use that big beautiful brain of yours! Think critically about what types variables should be and how different methods should be set up. This is a class design homework, so think about the overall design of your classes as you go.

- You are free to create your own instance data and helper methods that might aid you in completing the required methods. Just be sure to appropriately modify your instance data!

- For purchases being made 75% of the time, think about how to make this decision by generating a random double between 0 and 1.
- `System.nanoTime()` is a useful method for finding the execution time of the program.
- You are NOT permitted to used any other data structure in the place of arrays. Additionally, you are NOT permitted to use any class that trivializes this assignment. This includes but is not limited to `Arrays.java`, `ArrayList.java`, etc.
- If your program crashes with correct input, it will be **-25 points**.
- Enhanced for loops can be very useful when working with arrays when you do not need direct access to the the array.
- You will be using `java.util.Random` and some of its various methods. Whenever you want to use the `nextInt()` method, use the method by the same name which takes in a bound: `nextInt(int bound)`. Read the API as to what numbers are generated and think about what number you might want to use as your bound.
- Feel free to consult outside sources for help! The TAs are always here to assist, and a quick Google search for some topic or concept can be beneficial as well.
- For those of you that like to procrastinate, don't. This homework might be a little tricky even though it doesn't sound like it. I'd hate for you to realize that on the day it is due.

4 Checkstyle

Checkstyle counts for this homework. You may be deducted up to 15 points for having Checkstyle errors in your assignment. Each error found by Checkstyle is worth one point. This cap will be raised next assignment. Again, the full style guide for this course that you must adhere to can be found by clicking **here**.

Come to us in office hours or post on Piazza if you have specific questions about what Checkstyle is looking for and how to fix Checkstyle errors.

First, make sure you download the `checkstyle-6.2.2.jar` and `cs1331-checkstyle.xml` from the T-Square assignment page. Then, make sure you put *both* of those files in the same directory (folder) as the `.java` files you want to run Checkstyle on. Finally, to run Checkstyle, type the first line into your terminal while in the directory of your Java files and press enter.

```
$ java -jar checkstyle-6.2.2.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by piping the output of Checkstyle through `wc -l` and subtracting 2 for the two non-error lines printed above (which is how we will deduct points). For example:

```
$ java -jar checkstyle-6.2.2.jar -c cs1331-checkstyle.xml *.java | wc -l
      2
```

Alternatively, if you are on Windows, you can use the following instead:

```
C:\> java -jar checkstyle-6.2.2.jar -c cs1331-checkstyle.xml *.java | findstr /v "Starting
    audit..." | findstr /v "Audit done" | find /c /v "hashcode()"
0
```

5 Turn-in Procedure

Submit your `Business.java, Person.java, Item.java, and MarketSim.java` files on T-Square as an attachment. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double- check that you have submitted and not just saved a draft.

Keep in mind, unless you are told otherwise, non-compiling submissions on ALL homeworks will be an automatic 0. You may not be warned of this in the future, and so you should consider this as your one and only formal warning. This policy applies to (but is not limited to) the following:

1. Forgetting to submit a file
2. One file out of many not compiling, thus causing the entire project not to compile
3. One single missing semi-colon you accidentally removed when fixing Checkstyle stuff
4. Files that compile in an IDE but not in the command line
5. Typos
6. etc...

We know it's a little heavy-handed, but we do this to keep everyone on an even playing field and keep our grading process going smoothly. So please please make sure your code compiles before submitting. We'd hate to see all your hard work go to waste!

6 Verify the Success of Your Submission to T-Square

Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.

3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.

4. Recompile and test those exact files.

5. This helps guard against a few things.

   (a) It helps insure that you turn in the correct files.

   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)

   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is 8PM Thursday. Do not wait until the last minute!