(48024) Applications Programming Lab Guide - Classes

A brief word before we get started...

What I think is worth your noticing is that this week is more heavy on concepts and theory than previous weeks. It may be true that some students are just better at taking in new concepts and theory than others, perhaps, but it is also true that different students have different study approaches, and some of those might be more effective than others.

So in a week that is particularly heavy on concepts, you may need to adjust your study approach - take more notes for anything you don't understand or are likely to forget and make sure you **use your patterns book and other references** while working on the code. If there is a concept that you have missed, no deep thinking about it will ever produce an answer that you simply don't know, you may actually have to rewatch a video and take down **new notes**.

Reference examples

The main reference example this week is the bank2 example which is included in your skeleton JAR file for this week. This download also includes a bank1 example, but you can ignore it. The bank2 code is the same code that your tutor demonstrated in class.

In the bank2 example, the similarities are as follows:

- a product is like a bank account
- the product's stock is like the bank account's balance (it goes up and down)
- the product's sell method is like the bank account's withdraw method
- the store is like the customer - both have fields which are reference types.

  So if you don't know how to define the fields in Store, take a look at Customer for an example.

Besides the tutor demo, the lecture videos also include worked and explained examples.

Specification #1

Below is a plan similar to what most groups will have arrived at based on guidance from their tutor. If you do not yet have a workable plan for this lab, you may use the one described below.

System structure chart

Classes	Store	Product	CashRegister
Fields	product cashRegister	name stock price	cash
Constructors	Store()	Product(String name, int stock, double price)	CashRegister()
Goals			
menu	use()		
sell	sell()	double sell(int howMany)	add(double money)
restock	restock()	restock(int howMany)	
view stock	viewStock()	toString()	
view cash	viewCash()		toString()

The following sections provide guidance on how to code the fields, constructors, toString functions and goals. You can skip to the part that you are stuck on.

Fields

Build it up from right to left.

- Open up the CashRegister class. According to the above analysis (which you came up with in class), a CashRegister has “cash”. What **type** of thing is cash? It’s a double. So you declare the field as:

  private **double** cash;

  If you did not know that this is the correct way to declare a field, then you probably need to write more in your patterns book for this week’s topic, and you may need to rewatch some of the videos. Without a doubt, the rest of the semester will build heavily on the concepts introduced this week. If there is anything you didn’t quite understand, NOW is the pivotal point in the semester for you to do something about it.

Some students wrote things like this:

private cash;

Or this:

double cash;

If you did not know these were incorrect, then you should write down the correct field declaration pattern. Not every student needs to write this down, but if you don't know it, you should write it down. Write down the things you need to learn and anything you are likely to forget.

- Moving left, open up the Product class. According to the above analysis, a product has a name, stock and price. What **type** of thing is the name? How bout the stock? How about the price? Declare each of these 3 fields.

- Open up the Store class. According to the above analysis, a store has a product and a cash register. What **type** of thing is a product? Answer: it is a **P**roduct (notice the type Product is the classname Product, so it begins with an uppercase letter). The Store also has a cashRegister. What **type** of thing is this? Answer: it is a **C**ashRegister (i.e. it belongs to the CashRegister class with an uppercase C). The fields of class Store a different because they do NOT have primitive types like int or double, etc. They have reference types like Product or CashRegister because they are going to store whole objects, not just individual numbers.

  If you need an example of this, refer to the Customer class of the bank2 demo.

Submit your code to PLATE now and after completing each step so that PLATE can give you marks, AND so that you can check if you are on the right track.

Constructors

Build the constructors from right to left.
- Open the CashRegister class. It has 1 field. Which constructor pattern should apply?
  - Read constructor? No, the initial cash is NOT read from the user.
  - Parameters constructor? No, the initial cash is always zero.
  - Default constructor? Yes, the initial cash is always zero.
    Code the solution by **referring to examples** of this pattern.
- Open the Product class. It has 3 fields. Which constructor pattern should apply?
  - Read constructor? No, the initial product name, stock and price is not read from the user.
  - Default constructor? No, if you set the fields to literals like "Sticky tape", 200 and 2.99, then you have hard coded it to that particular product. What if next

week the store decides to diversify and have multiple products with a different product name, stock and price in each instance?
    - Parameters constructor? Yes, this is more flexible as it allows different field values to be passed in for each instance.
      Code the solution by **referring to examples** of this pattern.
- Open the Store class. It has 2 fields. The fields are reference types, so they will be initialised differently. You need to store a new object into each field. For an example, refer to the Customer class of the bank2 demo. When create a new object, you call the class's constructor. If that constructor has any parameters, you need to look at how many parameters it has, what order they're in, what type each parameter has, and then you will need to pass in actual arguments in the same order and matching the same types. Take a look at the Customer constructor for example. It tries to create a new Account object like this:

  ```
  savingsAccount = new Account("Savings");
  ```

  Why does it pass one string argument "Savings"? Because if you look at the way that the Account constructor was defined, it was defined to require one String argument:

  ```
  public Account(String type) { … }
  ```

  You just need to make sure that the type and number of arguments you pass into the constructor match the type and number of parameters declared for the constructor. Your Product constructor declares 3 parameters (name, stock, price) in a particular order with particular types. So when you try to create a new Product, you must pass in actual argument values with the same types and in the same order. Hint: the first parameter is the product name of type String, so you pass in the actual argument "Sticky tape". We put double quotes only because the type is String. The second and third parameters are not Strings so their arguments should not have double quotes.

Submit your code to PLATE now to see if you're on the right track, and also to receive marks for what you have done so far.

If PLATE tells you that something is wrong, do NOT continue onto the next step. If you ignore the fact that you are heading in the wrong direction and you keep going in that direction, you will only create more work for yourself to correct. To make progress, you need to be able to test it at each stage.

toString functions

They are just like the bank2 demo example.

Goals

Implement the goals in the same order that PLATE tests them. Submit to PLATE after each step to see PLATE's next test case.

Menu

It's just like the bank2 demo example.

View stock

It's just like the bank2 demo example.

Sell

This is similar to the withdraw method in the bank2 example. Notice how the goal is spread across classes.

The goal of the sell method is to deduct a number from the product's stock, where that number is read from the user. The plan is also spread across classes:

Store	Product	CashRegister
sell()	double sell(int number) boolean has(int number)	add(double money)
PLAN: 1. Read the number 3. Check if product has number 2. sell that number of the product 3. add money earned to CashRegister	sell PLAN: 1. Deduct number from stock 2. Calculate and return the money earned has PLAN: A boolean function similar to the bank2 example	PLAN: 1. add the money onto the cash field

Notice that the read pattern is implemented in the Store class. Remember that from week 3, all read patterns should be implemented as functions. So you must make a separate read function to read in the number of stock to sell. A read function should be named readX where X is the name of thing being read. So in this case, the read function should be named readNumber. You also have examples of the read pattern in the bank2 demo.

The main difference between the bank2 withdraw goal vs the store's sell goal is that here we suggest that the Product's sell method return the money earned. This is the difficult part, and you are free to skip it for now. Note:

1. You get some marks for deducting the number from the product's stock.
2. You get some marks for adding money to the cash register's cash.

If you find the second step difficult, come back to it at the end.

Restock

Similar to the deposit goal in the bank2 demo example.

View cash

Similar to the view stock goal, but you won't be able to get any marks for it until after you have added the money to the cash register from the sale. So if you skipped that part earlier, go back and complete that part first.

Specification #2

This specification really tests that you are capable of putting together all of the patterns and processes learnt so far. Make sure you use the best coding practices while completing this specification. For example, make sure you use the new versions of both the read loop pattern and the read pattern that were introduced last week. That means you should create a merged read loop and you should call on a separate read function.

To produce the string representation of the lift, you may find it helpful to refer back to the Diamond example in the first video in the study module on methods. That example showed how to perform a loop to repeat a character a certain number of times. However, that example was built using procedures, but this time we are asking you to build it using functions. Functions should not have side effects, so your toString() function should not print anything out.

Instead, you need a way to be able to build up a string without printing it. Instead of doing this:

```
System.out.print(“a”);
System.out.print(“b”);
System.out.print(“c”);
```

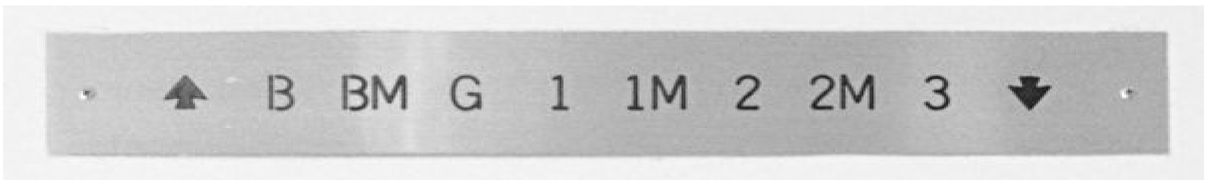
You can do this:

```
String s = “”;
s += “a”;
```

```
s += “b”;
s += “c”;
return s;
```

Some students are unsure of how to interpret the output of this program. As this specification is the basis for Assignment 1, you may find it helpful to also read through the Assignment 1 specification which provides more context to the problem.

In short, the program outputs a lift indicator. You are limited to text output, but the goal is to emulate something like this:



This indicator shows all the levels of the building from left to right, and as the lift moves up and down the lift shaft, different levels will be highlighted on the lift indicator.

In your program, you display the lift indicator using ASCII characters. If the building has 6 levels, then the indicator might look like this: |------|. If the lift is currently on level 1, it will look like this: |#-----|. If the lift is currently on level 4, it will look like this: |---#--|. If the lift is currently on level 6, it will look like this: |-----#|.

Part of the program’s goal is to produce this string representation of the lift indicator. You have been given this code as a starting point:

```
public class Lift {
   private int bottom;
   private int top;
   private int level;
}
```

The bottom and top fields represent the bottom level (e.g. 1) and the top level (e.g. 6) that the lift reaches, while the level field represents the current level that the lift is on. If the lift is currently on level 4, then the number 4 will be stored into the level field.