(48024) Applications Programming Lab Guide - Lists

You should use the tutor’s bank example as a reference while attempting this week’s lab.

Customer and Store are similar because they are both clients containing lists.

- A Customer has a list of Accounts
- A Store has a list of Products

Account and Product are similar because they are both suppliers.

- An account contains data and a toString() function
- A product contains data and a toString() function

Constructors

Product

Code the Product constructor first while referring to the Account constructor. Just as the Account constructor initialises the type field from the parameter with “this.type = type;” your Product constructor should initialise 3 fields from the 3 parameters. ALL 3 fields should be initialised from parameters.

Store

Code the Store constructor while referring to the Customer constructor. Just as the Customer constructor initialises the list of accounts, your Store constructor should initialise the list of products. Remember to create each new Product with 3 parameters. If you forget how to create a new Product, refer back to your solution to last week’s lab.

Note that the list of products is not the only field in Store. There is another field. The constructor should initialise both fields. If you forget how to initialise the other field, look at your Lab 4 solution.

Goals

Use the tutor’s bank example as a reference.
Use your solution to last week’s lab as a reference.

As usual, code the main method and use method first.

View stock

Refer to the bank example, and specifically look at the view method. It has a couple of different versions of the code, with one version commented out. Look at the commented-out version highlighted below in blue:

```
private void view() {
      // View all accounts:
      //
      // for (Account account : accounts)
      //    System.out.println(account);
      //
      // View selected account:
      Account account = account(readType());
      if (account != null)
            System.out.println(account);
      else
            System.out.println("No such account");
}
```

This code loops over each account in the accounts list and shows that account. This works because the account has a toString() function.

You need to write code that loops over each product in the products list and shows that product.

If you get output like this:

```
[Whiteboard Marker - 85 at $1.50, Whiteboard Eraser - .......
```

It means you did this:

```
System.out.println(products);
```

This is wrong because `products` is an ArrayList. The ArrayList class implements its own toString() function that displays all elements on a single line separated by commas and surrounded by a pair of [square brackets]. You don't want that string representation.

Instead, you need to loop over each product in the list and print out each product. You will find an example of this in the bank demo example. Importantly, each time through the loop, the parameter passed into println() should be a single product, not the products list.

Sell

Use last week's solution as a reference:

```
     int number = readNumber();
      if (product.has(number)) {
           double sale = product.sell(number);
           cashRegister.add(sale);
      }
      else
           System.out.println("Not enough stock");
```

Last week's store had one product stored in a field named `product`. The solution above refers to that field. But this week, there is no `product` field, so the above code won't work immediately. This week, there is instead a list of `products`, and you need to ask the user which product they want to sell. The above code will work if you insert some new code above it. You need to:

1. Read the product name from the user.
2. "Look up" that product in the list.
3. Print a message to say that you are selling this product.

To "look up" the product in the list, you need a separate lookup function following the lookup pattern, and you need to call it with the product name from step (1) as a parameter. Again, refer to the bank account example which defines the following lookup function:

```
private Account account(String type) {
     for (Account account : accounts)
          if (account.hasType(type))
               return account;
     return null;
}
```

You need a similar lookup function, but but instead of looking through the list of accounts for a particular account that has the given type, you need to look through the list of products for a particular product that has the given name. Also, just like we use a matches function to test if the account has a matching type, you will need to use a matches function to test if the product has a matching name. In the bank demo, the Account class defines this matches function as follows:

```
public boolean hasType(String type) {
     return type.equals(this.type);
}
```

You need to define a similar matches function in your Product class to test if the product has a matching "name" (rather than type).

Restock

No tips provided

Prune

There is a pattern in the lecture slides to remove all matches from a list.

Advanced goals

As you begin to deal with partial matches, you will find you need a new pattern to replace lookup. Lookup can only find one product. Another pattern in the lecture notes can be used to find multiple matches. This pattern replaces the lookup pattern because whenever you lookup a product by name, there is always the possibility that it may return more than one match. It is suggested that you define a function with the header:

```
private LinkedList<Product> products(String partialName)
```

This function says, “Give me a partial name and I will return you a list of products that match”. There is no need to pass the original list of products as a parameter because your function already has direct access to the `products` field.