Due Oct 1 by 11:59pm **Points** 2 **Available** Sep 25 at 9pm - Oct 2 at 11:59pm

This assignment was locked Oct 2 at 11:59pm.

Lab 3 — Protocols, Interfaces and Abstract Classes

Home

Modules

Lab 3: Interfaces & Abstract Classes

Bank Account

1.1 Introduction

The purpose of this lab is to give you practice with the new concepts from this module, such as polymorphism, sub-typing and protocols (known as interfaces in Java). We will also explore some early-stage code reuse via inheritance (using abstract classes).

1.2 What to do

For this lab, you will design and implement the start of a banking solution for a neighborhood bank. There are two types of accounts the bank wants you to implement: one called `SavingsAccount` and one called `CheckingAccount`. Your interface and all classes must be in the `bank` package.

Both accounts can do the following:

- Create a new account by specifying a "starter" amount of money to open it with. The starter amount must be greater than or equal to one cent.
  Do this: **Create a constructor** that takes a single parameter (of type `double`) that represents the "starter amount" for the account. If the amount specified is negative OR the amount is less than one cent ($0.01), throw an `IllegalArgumentException`
- Deposit into their account.
  Do this: **Create a method** called `deposit` that takes a single parameter (of type `double`) that represents the amount deposited into the account. If the amount specified is negative, throw an `IllegalArgumentException`
- Withdraw from their account. If the amount specified is greater than the balance available, this operation fails and returns false.
  Do this: Create a method called `withdraw` that reduces the account balance by the amount specified. Return `true` if the transaction is successful, `false` otherwise. (Hint: test for a variety of cases here, i.e. values > 0, 0 ...etc, what other cases may cause the withdraw transaction to fail)
- Check their balance.
  Do this: Create a method `getBalance` that returns a double (the current account balance)

Non-customer behavior you must implement:

- Bank administrators can **perform monthly maintenance** to assess monthly fees and give a "clean slate" for the subsequent month.
  Do this: **Create a `performMonthlyMaintenance`** method to charge any fees and then reset transaction counters to zero.
- **Do this:** **Create a toString** method that prints the account balance in dollars/cents format (e.g: $10.00). You may want to look up the documentation for the String.format method for this part, or use the DecimalFormat class if you wish.

You are required to use the **`IAccount`** interface as specified below. Both types of accounts implement this interface, so that the bank can access either account through that common protocol. You will need to consider behavior variations as described below.

Behavior variations for a **SavingsAccount** withdraw() method: Savings accounts allow for 6 penalty-free withdrawal transactions per month. Savings accounts allow for an unlimited number of deposits per month.
Rules:

- If the amount specified for the withdrawal is negative, the operation fails. If the number of withdrawals for the month is greater than 6, a transaction penalty of $14 is deducted from the account when monthly maintenance is performed

Behavior variations for a **CheckingAccount** performMonthlyMaintenance() method: A minimum balance of $100 must be maintained throughout the month to avoid fees
Rules:

- If the checking balance falls below $100 at ANY time during the month (before maintenance is performed) an account **maintenance fee of $5 is charged** when the monthly maintenance is performed.

Notes

For each method you write:

- Design the signature of the method.
- Write Javadoc-style comments for that method.
- Write the body for the method.
- Write one or more tests that check that the method works as specified in all cases.

Avoid duplicating code as much as possible. You will likely want to use an abstract superclass and “push” common method code up to that class. Feel free to create private “helper” methods if you need to.

Be sure to use access modifiers, private, *default* (no keyword), protected, and public appropriately.

Include JavaDoc for your classes and constructors as appropriate. You do not need to repeat JavaDoc already existing in a superclass or interface when you override a method. (This is true for the course in general.)

Provide a UML class diagram for your solution.

Here is the IAccount interface. This is the "contract" your solution is agreeing to implement

```
/**
 * This interface represents a bank account. It is the super-type for
 * any other type of traditional financial account a bank might offer
 */

public interface IAccount {
  void deposit(double amount);
  boolean withdraw(double amount);
  double getBalance();
  void performMonthlyMaintenance();
}
```

Lab 3 - rubric

Full Credit - Pass	**Partial Credit - Pass**	**No Credit**
2 pts	**1 pt**	**0 pt**
Bank solution passes all automated tests. Good use of implements and inheritance (extends) for reuse, documented and reasonable number of tests provided.	Bank Solution passes most automated tests. Good use of implements; partial use of inheritance for reuse. Some documentation and/or tests partially implemented	Bank solution passes few tests, problems with OO themes. Missing documentation, missing tests. Or assignment not submitted