Laboratory Sheet 4

This Lab Sheet contains material based on Lectures 1 – 8 (up to *15 October 2014*), and contains the submission information for Laboratory 4 (week 5, *20 – 24 October 2014*).

The deadline for submission of the lab exercise is 48 hours after the end of your scheduled laboratory session in week 5 *(20 – 24 October).*

Aims and objectives

- Further practice with object-oriented modeling in Java
- Particular focus on subclasses, abstract classes and overriding methods

Set up

When you download Laboratory4.zip from moodle, please unzip this file. You will obtain a folder Laboratory4, containing a subfolder entitled Submission4_1. Remember that for this Laboratory you will have to switch your Eclipse workspace to the Laboratory4 folder.

In the folder Submission4_1 will be

- a file Transaction.java that contains a skeleton `Transaction` class
- a file TransactionalBankAccount.java that contains a skeleton `TransactionalBankAccount` class
- a file BankAccount.java that is a sample solution file from Laboratory 3
- a file TestTransactionalBankAccount.java that contains JUnit tests for your submission

In Eclipse, you should create a new project entitled Submission4_1; the given files will automatically become part of this project.

Submission material

This exercise builds on the material you submitted for Laboratory 3, so it might be worth referring back to your notes for that session.

Submission exercise

Design a `Transaction` abstract class to represent a transaction on a single bank account. The `Transaction` class should have two `private` instance fields:

- `java.util.Date` date
- `double` amount

and getter methods for both these fields.
It also needs a `public` constructor that takes a `double` amount parameter. The constructor will set `this.amount` to the appropriate value. The constructor will also set `this.date` to the current date and time, using the static library method `java.util.Calendar.getInstance()`[1] and the `Calendar` instance method `getTime()`[2] which returns a `Date` object.

[1] http://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html#getInstance()

[2] http://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html#getTime()

Note that the `Transaction` abstract class has an abstract method with this signature:

```
public abstract boolean apply(TransactionalBankAccount b);
```

Extend the `BankAccount` class that was introduced in Laboratory 3, by creating a `TransactionalBankAccount` subclass that includes details of the most recent transaction. This involves the following:

- add a new `private` instance field `mostRecentTransaction` of type Transaction
- add `public` getter and setter methods for `mostRecentTransaction`
- add two constructors that call the appropriate superclass constructors:
  - `public TransactionalBankAccount()`
  - `public TransactionalBankAccount(String holder, double limit)`
- override the inherited `toString()` method to include the `mostRecentTransaction.toString()` value, as well as calling the parent's (i.e. `BankAccount`) `toString()` method.

Design a `Withdrawal` class and a `Deposit` class, both of which are concrete (i.e. non-`abstract`) subclasses of `Transaction`. Each class should contain an appropriately defined `apply()` method, which operates as follows:

- delegate the bank account updates to `withdraw()` or `deposit()` method associated with the `BankAccount` parameter.
- If the transaction is successfully applied (i.e. no `Exception` is thrown and the `boolean` return value, if any, is `true`), then update the account's `mostRecentTransaction` field via its `public` setter method.
- The `boolean` return value of `apply()` should indicate whether or not the transaction was successful.

Also define a `toString()` method for the `Withdrawal` and `Deposit` classes. The `Withdrawal.toString()` method will return a `String` formatted as follows:

2013-10-03 23:59 WITHDRAWAL £100.00

The `Deposit.toString()` method will return a `String` formatted as follows:

2013-10-03 23:59 DEPOSIT £100.00

You can use the java.text.SimpleDateFormat[3] class to format your Date object as a String.

Use the supplied test class, `TestTransactionalBankAccount` to check that your code conforms to the specification as given above. This JUnit test class creates a small number of accounts and carries out some operations on these accounts, checking their state for consistency. Another way to test your code is to create a new source code file MyTest.java with a `main` method. In this `main` method, instantiate some new `TransactionalBankAccount` objects and invoke various methods on them.

Submission

You should submit your work before the deadline no matter whether the programs are fully working or not.

When you are ready to submit, go to the JOOSE2 moodle site. Click on Laboratory 4 Submission. Click 'Add Submission'. Open Windows Explorer and browse to the folder that

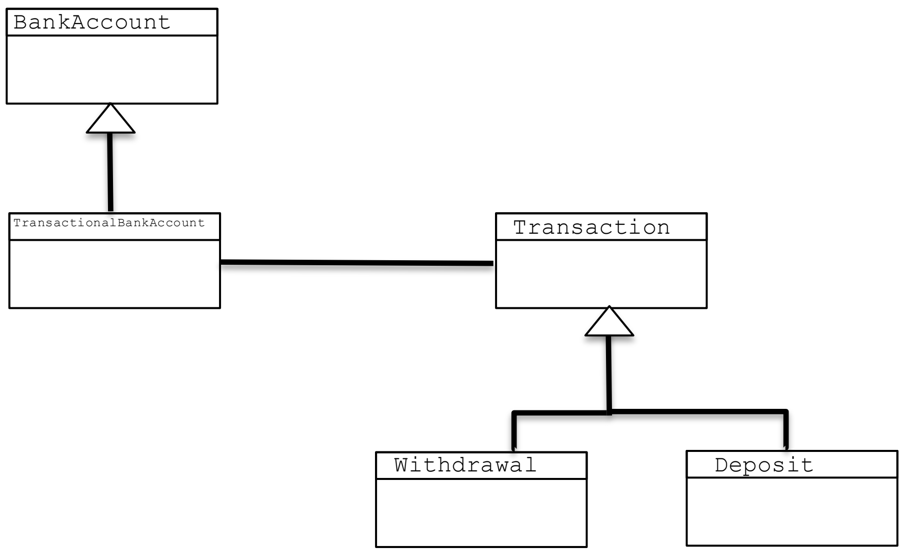
[3] http://www.tutorialspoint.com/java/java_date_time.htm gives examples.

contains your Java source code …\Laboratory4\Submission4_1\ and drag *only* the four Java files Transaction.java, Withdrawal.java, Deposit.java and TransactionalBankAccount.java into the drag-n-drop area on the moodle submission page. **Your markers only want to read your java files, not your class files.** Then click the blue save changes button. Check the four .java files are uploaded to the system. Then click submit assignment and fill in the non-plagiarism declaration. Your tutor will inspect your file and return feedback to you via moodle.

Outline Mark Scheme

A: code passes all tests. Neatly formatted. Appropriate annotations and comments.
B: code passes most tests. Fairly well formatted. Some annotations and comments.
C: code may not compile. Sensible attempt at implementing most methods.
D: code incomplete, but an attempt has been made.

UML Class Diagram (for reference)


BankAccount
TransactionalBankAccount
Transaction
Withdrawal
Deposit