Lab 6: Savings & Checking Accounts
Steven K. Andrianoff
Robert Harlan
David Levine
Computer Science Department
St. Bonaventure University
Copyright, 2002

Objective:

This lab introduces inheritance. We will see how to define a derived class (also called a subclass) of an existing class and then examine the relationship between the derived class and the base class (also called the superclass). We will also see how to override inherited methods.

Background:

This lab assumes you have a correct BankAccount class from Lab 5. The BankAccount class should define:

- three instance variables - for the balance, account number, and customer name
- a 3-parameter constructor that initializes the instance variables, and
- these three methods:
  - getBalance() – returns the balance in the account
  - deposit(amount) – increases the balance by amount
  - withdraw(amount) – decreases the balance by amount

Note: For this lab the balance must be a float or a double. Lab 5 had other tasks that are not vital to this lab including a second constructor and protection against negative balances using Exceptions. While your BankAccount class does not have to be perfect, it should implement each of the above correctly. If you used an int for the balance, you must make the requisite changes now. Similarly, your methods must be named as above. (The name of the parameter - amount above - is not important; the names of the methods are.)

In this lab you will define two classes that inherit from (or extend) the BankAccount class.

The SavingsAccount class models a bank account that earns interest. Each SavingsAccount object will have:

- four instance variables:
  - a balance
  - an account number
  - and a customer name, (all inherited from BankAccount), and
  - an interest rate (new to SavingsAccount)
- a 4-parameter constructor that initializes the instance variables, and
- methods:
  - getBalance() (inherited from BankAccount)
  - deposit(amount) (inherited from BankAccount)
  - withdraw(amount) (inherited from BankAccount)
  - addInterest() (new to SavingsAccount) – calculates an interest amount and adds it to the balance
The *CheckingAccount* class models a bank account that charges a fixed fee for each transaction (deposit or withdrawal). Each *CheckingAccount* object will have:

- four instance variables:
  - a balance
  - an account number
  - and a customer name (all inherited from *BankAccount*), and
  - a count of the transactions performed (new to *CheckingAccount*)
- a 3-parameter constructor that initializes all of the instance variables, and
- methods:
  - `getBalance()` (inherited from *BankAccount*)
  - `deposit(amount)` (overrides *BankAccount* method)
  - `withdraw(amount)` (overrides *BankAccount* method)
  - `deductFees()` (new to *CheckingAccount*) – calculates fees, deducts fees from balance, and resets number of transactions.

In addition to these the *CheckingAccount* class will store a constant for the amount of the fee.

**Key idea:** Through the use of the *class inheritance* mechanism both the *SavingsAccount* and *CheckingAccount* classes can be written in such a way that only the new instance variables and new methods need to be defined.

**Preparation:**

Before beginning, make any necessary changes to your Lab 5 (*BankAccount*) and then make a new project called Lab 6 which is a copy of the (possibly changed) Lab 5. **Use this copy only** throughout this week's lab.

**Instructions:**

1. Open your Lab 6 project in Eclipse. If necessary, change the instance variable balance to a double. At this time, make sure that each of your instance variables is declared to be private, not protected. (Yes, there are reasons why protected might be better, but we are exploring those very issues in this lab, so start with private.) Change any of the methods (in the *BankAccount* class and in the *TestCase* that are affected by this change. Double-check that the *BankAccount* class works properly by producing an appropriate green bar.) Next, change the method names. Do so by right-clicking on the inappropriately named method anywhere in the source code and choosing **Refactor: Rename** from the pop-up menu.

2. First, we will write the *SavingsAccount* class. To create the file for this class, go to **File:New** and select **Class**. In this wizard name the new class *SavingsAccount* and in the superclass field type in *BankAccount*. Leave the rest of the fields as they are and select Finish. You should now have a new file named *SavingsAccount.java* with the skeleton of a class named *SavingsAccount* that *extends* *BankAccount*.

   **Within the *SavingsAccount* class**
   - Define a private instance variable to store the interest rate,
   - Define a constructor that takes four parameters: the three parameters used by the *BankAccount* class and a parameter for the interest rate,
   - Define a method `addInterest()` which adds interest to the balance.

   The *SavingsAccount* constructor needs to first invoke the *BankAccount* constructor with the first three parameters provided. The following statement is of the correct form to accomplish this:
   ```java
   super(param1, param2, param3);
   ```
This statement must be the first statement in the SavingsAccount constructor. Following this statement the SavingsAccount constructor needs to initialize the value of the interest rate using the fourth parameter.

The coding of the addInterest() method requires further understanding. A SavingsAccount is a BankAccount so it maintains a balance, however, the balance is a private instance variable in a BankAccount so it is not directly accessible from the body of addInterest(). The only way for the code in the body of addInterest() to see the balance is by using the public getBalance() method of the BankAccount class. For addInterest() to calculate the interest it will need to call getBalance(). For addInterest() to update the balance it will need to call deposit().

Create a JUnit Test Case that will test your SavingsAccount class. In particular, create a SavingsAccount object, and have it perform the appropriate operations. Note that you should be able to invoke any BankAccount method from your SavingsAccount object. Make sure that you test this. As per last week's lab, take a snip-shot of the green bar and paste it into your report for this week.

3. For this step, create a new Java class that will contain a main method. We will use this method to test various aspects of inheritance regarding our classes. (Note that this is different than testing our classes for correctness; we use JUnit for that.)

Answer the following questions and in each case paste into your lab report the Java statements you used to test the idea and the results you obtained. Identify any messages as either syntax error messages or run-time exception messages. In either case record the full message. After testing each item, comment out the code in your main method so that there is "proof" that you tested it.

- What happens if you try to print a SavingsAccount object? Is this what you expect, and if not, how could you adjust this?
- What happens if you try to add interest to a BankAccount object?
- Can you assign a SavingsAccount object to a BankAccount variable? If this is possible, what happens if you try to add interest to the BankAccount variable?
- Continuing with the previous question, assign a SavingsAccount object to a BankAccount variable so that you have two different variable names for the one object. What happens if you add interest to the SavingsAccount variable but then ask the BankAccount variable for its balance? Is the result what you expected?
- Can you assign a BankAccount object to a SavingsAccount variable?
- Create a SavingsAccount object. Use the instanceof operator to see if the variable is a SavingsAccount, then see if it is a BankAccount, and then see if it is an Object. Is it?

Print a copy of the source file with the main method and hand it in with your lab.

4. Consider for a moment a SavingsAccount. While it is true that it should manipulate a customer name and an account number exactly as a BankAccount does, the same does not really apply to its balance. In fact, the fundamental difference between the two accounts is that a SavingsAccount periodically adds money (interest) to the balance without the benefit of a deposit. Therefore it is odd that we call that action a deposit and rely on some previously existing mechanism to handle this. It would make more sense to handle this by directly manipulating the balance, but we are unable to do so because that instance variable is private (in BankAccount). Java provides, however, a mechanism for subclasses to directly manipulate instance variables in a superclass if the superclass permits. This is achieved by declaring the variable to be protected (in the superclass). protected variables are like private variables in that they may not be manipulated from outside of the class, but it is permitted for subclasses to manipulate them.
Change the (BankAccount) instance variable for balance to be protected. Test your program, that is, test the creation of a SavingsAccount, make a deposit, add interest to the account, and then display the balance. What happens when you run your JUnit test case now?

Now change the body of the addInterest() method so that it directly manipulates the balance rather than making a call to the deposit() method. Test the program again. Is there any change? When everything works, again include a snip-shot of the green bar.

5. Override the toString() method in the SavingsAccount class. Add code to your JUnit suite to test toString(). And, take the appropriate snip-shot.

Make sure the file SavingsAccount.java is properly documented with javadoc-style comments. Print a copy of the source file SavingsAccount.java and hand it in with your lab.

6. Define a CheckingAccount class in a manner similar to the SavingsAccount (by inheriting from BankAccount).

Within the CheckingAccount class
- Define a private instance variable to store the number of transactions.
- Define a constructor that takes only three parameters: the same three parameters required by the BankAccount constructor. The constructor should call the BankAccount constructor and initialize the new instance variable.
- Define a method to deduct fees from the account.

We will assume the transaction fee is a constant that applies to all CheckingAccount objects. Define a constant that is visible to all CheckingAccount objects but for which there is only one value stored rather than one value per object. To do this define the constant using

```java
private static final double TRANSACTION_FEE = 0.25;
```
(Place this outside of all the methods, near the declarations of the instance variables.)

Since every deposit and withdrawal bumps the number of transactions by 1, the deposit() and withdraw() methods of the BankAccount class need to be overridden. To override the deposit() method add a deposit() method to the CheckingAccount class that has the same signature as BankAccount’s deposit() method. (Recall that the signature of a method includes the return type, the name, and the parameters.) The body of the CheckingAccount deposit() method must do two things: increment the number of transactions and increase the balance. The withdraw() method is similar.

The method to deduct fees from the account should calculate the total fees, subtract the total fees from the balance, and reset the number of transactions to zero.

Add code to your test case to test the CheckingAccount class. To test the class create a CheckingAccount object, perform several transactions (both deposits and withdrawals), deduct the transaction fees, and then display the balance.

Be sure to include a test that verifies that the number of transactions was properly reset when the fees were deducted. Describe (in English sentences, not Java code) in your lab write-up how you tested this. Grab one last snip-shot of the green bar.

Print your JUnit test class.
7. Provide javadoc documentation for the CheckingAccount class. Print a copy of the source file CheckingAccount.java and hand it in with your lab.

With javadoc you can create a collection of inter-linked html files with the class specifications for BankAccount, SavingsAccount, and CheckingAccount. Run the javadoc utility on the project to generate these class specifications. Show the resulting class specifications for the SavingsAccount and CheckingAccount classes to your lab instructor.

Hand in:

The write-up you hand in for this lab should include:

- answers to the questions with a description of the code you used to test the ideas from Step 3
- answers to the questions in Step 4
- the various green bar snip-shots (Steps 2, 4, 5, and 6).
- a description of the test requested in Step 6
- a copy of the source code as requested in Steps 5, 6, and 7
- a cover page

Help Policy:
Help Policy in Effect for This Assignment: Group Project with Limited Collaboration
In particular, you may discuss the assignment and concepts related to the assignment with the following persons, in addition to an instructor in this course: any member of your group; any St. Bonaventure Computer Science instructor; and any student enrolled in CS 132.

You may use the following materials produced by other students: materials produced by members of your group. You may use the following materials produced by other students: NONE.