Carleton University
Department of Systems and Computer Engineering
SYSC 2004 - Object-Oriented Software Development - Winter 2015

Lab 6 - Class Inheritance

Objective

The objective of this lab is for you to develop a simple inheritance hierarchy of classes that model different types of bank accounts.

Attendance/Demo

To receive credit for this lab, you must demonstrate your work. **Also, you must submit your lab work to cuLearn**. (Instructions are provided in the *Wrap Up* section at the end of this handout.)

When you have finished all the exercises, call a TA, who will grade the code you wrote. For those who don't finish early, a TA will ask you to demonstrate whatever code you've completed, starting about 30 minutes before the end of the lab period. **Any unfinished exercises should be treated as "homework"; complete these on your own time, before Lab 7.**

References

Objects First with Java, Chapters 8 and 9.

Getting Started

Step 1: Create a new folder named Lab 6.

Step 2: Download accounts.zip from cuLearn and move it to your Lab 6 folder.

Step 3: Extract the *accounts* project from the zip file into a project folder named accounts.

Step 4: Open the accounts folder. Double-click on the BlueJ project file, which is named package. This will launch BlueJ and open the *accounts* project.

Modelling Bank Accounts

A bank offers its customers two types of accounts:

- A savings account earns interest. The interest is compounded monthly and is calculated on the minimum monthly balance.
- A chequing account earns no interest. It provides three free withdrawals per month. After the three free withdrawals, a $1 transaction fee is charged every time money is withdrawn, until the end of the month.

Four operations are provided for both types of accounts:

- Deposit money;
- Withdraw money;
- Get (return) the current account balance (the amount of money in the account);
- Month-end processing. For a savings account, monthly interest is calculated on the minimum monthly balance and deposited in the account. The minimum monthly balance is then updated to reflect the new account balance. For a chequing account, the withdrawal counter (which keeps track of the number of withdrawals during the month) is zeroed.

Here is an example of how the minimum monthly balance of a savings account is calculated. Suppose that, at the start of the month, the account balance and the minimum balance are $1000. You then withdraw $600. The account balance and minimum balance are now $400.You then deposit $200. The account balance is now $600, but the minimum balance remains $400. You then withdraw $100. The account balance is now $500, but the minimum balance remains $400. On the last day of the month, you withdraw $200. The account balance and minimum balance are reduced to $300. If the monthly interest rate is 1%, $3 of interest (1% of the minimum monthly balance, $300) is deposited during month-end processing, so at the start of the next month the account balance and the minimum balance are $303.

You're going to build an inheritance hierarchy that consists of three classes: `BankAccount`, `SavingsAccount` and `ChequingAccount`. `SavingsAccount` and `ChequingAccount` will be subclasses of `BankAccount`.

Exercise 1 - Implementing the `BankAccount` Class

Step 1: Create a new class named `BankAccount`.

Step 2: Open `BankAccount` in an editor window. Delete the definition of field `x`, the constructor and method `sampleMethod`.

Step 3: `BankAccount` will define the field (instance variable) and all the methods that are common to all savings accounts and chequing accounts:

Step 3(a): All accounts have a balance, so define a field named `balance` that stores a real number. The visibility of this field must be `private`.

Step 3(b): Define the constructor and methods listed here:

- `public BankAccount(double initialBalance)` - the constructor always sets the balance to the specified initial balance.
- `public void deposit(double amount)` - this method increases the balance by the specified amount.
- `public void withdraw(double amount)` - this method decreases the balance by the specified amount. No check is made to see if the account will be overdrawn (i.e., result in

a negative balance).

- `public double getBalance()` - this method returns the current balance.

Step 4: Interactively create a `BankAccount` object with an initial balance of $100 and place it on the Object Bench.

Interactively test the `BankAccount` object's methods. For example, deposit $200 into the account, then verify that the account balance is now $300. Withdraw $100 from the account, then verify that the account balance is now $200. If necessary, correct any bugs in your code and retest the class.

Step 5: Review your class. Make sure that `BankAccount` defines exactly one field (instance variable). Novice Java programmers sometimes define fields when they should instead define variables that are local to methods. If the class has too many fields, you'll need to edit it to fix this, then retest the class.

Exercise 2 - Implementing the `SavingsAccount` Class

Step 1: Create a new class named `SavingsAccount`.

Step 2: Open `SavingsAccount` in an editor window. Modify the class header,

```
public class SavingsAccount
```

so that `SavingsAccount` is a subclass of `BankAccount`. Delete the definition of field `x`, the constructor and method `sampleMethod`.

Step 3: Each `SavingsAccount` object needs to store the monthly interest rate and the minimum monthly balance, which is updated by all withdrawals (reread **Modelling Bank Accounts**, above). In class `SavingsAccount`, define two fields named `interestRate` and `minBalance` that store real numbers. The visibility of these fields must be `private`.

Step 4: There's one more thing we need to review before you write the methods in `SavingsAccount`.

- When a Java subclass overrides (redefines) an inherited method, the method in the subclass can call the overridden method, *which has the same name*, but only by using the `super` keyword. For example, if a `withdraw` method in `SavingsAccount` needs to call the `withdraw` method inherited from `BankAccount`, it must use the expression:

  ```
  super.withdraw(amount);
  ```

- A method in a subclass can use an internal method call to call an inherited method, as long as the two methods *have different signatures*. For example, if a `withdraw` method in `SavingsAccount` needs to call the `getBalance` method inherited from `BankAccount`, it should use the expression:

```
getBalance()     // or this.getBalance()
```

The `withdraw` method in `SavingsAccount` shouldn't use the the `super` keyword to call the inherited `getBalanc`e method; i.e., it shouldn't call `getBalance` this way:

```
super.getBalance()   // Don't use super unless it's required!
```

Define incomplete implementation of the constructor and methods listed here. The bodies should be empty - you're defining just enough code so that the class will compile.

- `public SavingsAccount(double initialBalance)`
- `public void setInterestRate(double rate)`
- `public void withdraw(double amount)`
- `public void monthEnd()`

Make sure your class compiles without errors before you move on to Step 5.

Step 5: Complete the implementations of the constructor and methods, to meet these specifications:

- `public SavingsAccount(double initialBalance)` - calls its superclass constructor to set the balance to the specified initial balance. It also initializes the interest rate to 0, and minimum monthly balance to the initial balance.
- `public void setInterestRate(double rate)` - this method sets the monthly interest rate. Parameter `rate` is the monthly interest rate, expressed as a percent (e.g., if the interest rate is 1.2%, pass 1.2 to this method).
- `public void withdraw(double amount)` - this method reduces the amount of money in the account, and if necessary updates the minimum balance. Note that the `withdraw` method in `BankAccount` must be called to reduce the account balance, because methods in `SavingsAccount` cannot access the private `balance` field defined in `BankAccount`. (See the earlier explanation of how a method in a subclass can call an inherited method with the same name.) This method does not need to check if the account will be overdrawn (i.e., result in a negative balance).
- `public void monthEnd()` - this method calculates the monthly interest (remember, it's calculated on the **minimum** monthly balance, not the account balance at the end of the month), deposits this amount into the account, and then updates the minimum balance to equal the current balance.
- Do not define `deposit` and `getBalance` methods in `SavingsAccount`. The `deposit` and `getBalance` methods inherited from `BankAccount` provide the required operations for a savings account, so there is no need to override these methods in `SavingsAccount`.

If we defined these methods in `SavingsAccount`, they would look like this:

```
public void deposit(double amount)
{
    super.deposit(amount);
}

public double getBalance()
{
    return super.getBalance();
}
```

Clearly, these methods simply call the methods that are inherited from `BankAccount`, so there's no point in redefining them in `SavingsAccount`.

Step 6: Interactively create a `SavingsAccount` object with an initial balance of $1000 and place it on the Object Bench. Open an inspector on the object and verify that the constructor initialized the `balance`, `interestRate` and `minBalance` fields to 1000.0, 0.0, and 1000.0, respectively.

Interactively test the `SavingsAccount` class' methods. Set the interest rate to 1.0%. Follow the example described in **Modelling Bank Accounts**, above; i.e., withdraw and deposit the amounts listed in the scenario, and use the inspector to verify that the balance and minimum balance are updated correctly. Finally, call the `monthEnd` method and verify that the account balance and minimum monthly balance are both $303. If necessary, correct any bugs in your code and retest the class.

When testing your `SavingsAccount` object, make sure that you understand why you can call `deposit` and `getBalance` on a `SavingsAccount` object, even though these methods aren't defined in the `SavingsAccount` class.

Step 7: Review your class. Make sure that `SavingsAccount` defines exactly two fields. If the class has too many fields, you'll need to edit it to fix this, then retest the class.

Step 8: Class `SavingsAccountTest` contains a JUnit test suite for class `SavingsAccount`. Right-click on the icon for `SavingsAccountTest` and select Compile. Run the test suite by right-clicking on the class' icon and selecting Test All. A window titled BlueJ: Test Results will open. If your class passes all the tests, a green bar will be displayed, and a green check-mark will appear beside each of the test names. If necessary, edit your classes to correct any errors.

Exercise 3 - Implementing the `ChequingAccount` Class

Step 1: Create a new class named `ChequingAccount`.

Step 2: Open `ChequingAccount` in an editor window. Modify the class header,

```
public class ChequingAccount
```

so that `ChequingAccount` is a subclass of `BankAccount`. Delete the definition of field `x`, the constructor and method `sampleMethod`.

Step 3: Each `ChequingAccount` object needs to count the number of withdrawals, so that the transaction fee can be applied after the free withdrawal limit is reached (reread **Modelling Bank Accounts**, above). In class `ChequingAccount`, define a field named `withdrawals` that stores integers. The visibility of this field must be `private`.

Step 4: Define incomplete implementation of the constructor and methods listed here. The bodies should be empty - you're defining just enough code so that the class will compile.

- `public ChequingAccount(double initialBalance)`
- `public void withdraw(double amount)`
- `public void monthEnd()`

Make sure your class compiles without errors before you move on to Step 5.

Step 5: Complete the implementations of the constructor and methods, to meet these specifications:

- `public ChequingAccount(double initialBalance)` - calls its superclass constructor to set the balance to the specified initial balance. It also initializes the counter that keeps track of the number of withdrawals to 0.
- `public void withdraw(double amount)` - in addition to withdrawing the specified amount of money from the account, this method will update the withdrawal counter. If the number of withdrawals in the month exceeds the number of free withdrawals, the $1 transaction fee is charged when the money is withdrawn. Note that the withdraw method in `BankAccount` must be called to reduce the account balance, because methods in `ChequingAccount` cannot access the private balance field defined in `BankAccount`. This method does not need to check if the account will be overdrawn (i.e., results in a negative balance).
- `public void monthEnd()` - this method simply resets the withdrawal counter to 0.
- Do not define `deposit` and `getBalance` methods in `ChequingAccount`. The `deposit` and `getBalance` methods inherited from `BankAccount` provide the required operations for a chequing account, so there is no need to override these methods in `ChequingAccount`.

Step 6: Interactively create a `ChequingAccount` object with an initial balance of $1000 and place it on the Object Bench. Open an inspector on the object and verify that the constructor initialized the `balance` and `withdrawals` fields to 1000.0 and 0, respectively.

Interactively test the `ChequingAccount` class' methods. Use the inspector to verify that the withdrawal counter is updated each time you make a withdrawal. Verify that when the number of withdrawals in the month exceeds the number of free withdrawals, the $1 transaction fee is charged each time money is withdrawn. Finally, call the monthEnd method and verify that the withdrawal counter is reset to 0. If necessary, correct any bugs in your code and retest the class.

Step 7: Review your class. Make sure that ChequingAccount defines exactly one field. If the class has too many fields, you'll need to edit it to fix this, then retest the class.

Step 8: Class ChequingAccountTest contains a JUnit test suite for class ChequingAccount. Right-click on the icon for ChequingAccountTest and select Compile. Run the test suite by right-clicking on the class' icon and selecting Test All. A window titled BlueJ: Test Results will open. If your class passes all the tests, a green bar will be displayed, and a green check-mark will appear beside each of the test names. If necessary, edit your classes to correct any errors.

Wrap-Up

1. With one of the TAs watching, run the JUnit tests in SavingsAccountTest and ChequingAccountTest. The TA will note how many test cases pass. The TA will review your solutions to the exercises, assign a grade (Satisfactory, Marginal or Unsatisfactory) and have you initial the demo/sign-out sheet.

2. The next thing you'll do is package the project in a *jar* (Java archive) file named accounts.jar. To do this:

   2.1. From the menu bar, select Project > Create Jar File... A dialog box will appear. Click the Include source and Include BlueJ project files check boxes. A check-mark should appear in each box. Do not modify the Main class field.

   2.2. Click Continue. A dialog box will appear, asking you to specify the name for the jar file. Type accounts or select the BlueJ icon named accounts in the list of files. **Do not use any other name for your jar file** (e.g., lab6, my_project, etc.).

   2.3. Click Create. BlueJ will create a file named accounts that has extension .jar. (Note: you don't type this extension when you specify the filename in Step 2.2; instead, it's automatically appended when the jar file is created.) The jar file will contain copies of the Java source code and several other files associated with the project. (The original files in your accounts folder will not be removed).

3. Before you leave the lab, log in to cuLearn and submit accounts.jar. To do this:

   3.1. Click the Submit Lab 6 link. A page containing instructions and your submission status will be displayed. After you've read the instructions, click the Add submission button. A page containing a File submissions box will appear. Drag accounts.jar to the File submissions box. Do not submit another type of file (e.g., a .java file, a RAR file, a .txt file, etc.)

   3.2. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status of your file is "Draft (not submitted)". If you're ready to finish submitting the file, jump to Step 3.4. If you instead want to replace or delete your "draft" file submission, follow the instructions in Step 3.3.

3.3. You can replace or delete the file by clicking the Edit my submission button. The page containing the File submissions box will appear.

3.3.1. To overwrite a file you previously submitted with a file having the same name, drag another copy of the file to the File submissions box, then click the Overwrite button when you are told the file exists ("There is already a file called..."). After the icon for the file reappears in the box, click the Save changes button.

3.3.2. To delete a file you previously submitted, click its icon. A dialogue box will appear. Click the Delete button., then click the OK button when you are asked, "Are you sure you want to delete this file?" After the icon for the file disappears, click the Save changes button.

3.4. Once you're sure that you don't want to make any changes to the project you're submitting, click the Submit assignment button. A Submit assignment page will be displayed containing the message, "Are you sure you want to submit your work for grading? You will not be able to make any more changes." Click the Continue button to confirm that you are ready to submit your lab work. This will change the submission status to "Submitted for grading".

Review Questions

Attempt these questions before reading **What You Learned**, below. If your answers to these questions are correct, you should feel confident that you have a good understanding of class inheritance.

1. When you create a `SavingsAccount` object, how many `BankAccount` objects are created? How many `BankAccount` objects are created when you create a `ChequingAccount` object?
2. What are the names of all the fields in a `SavingsAccount` object? What are the names of all the fields in a `ChequingAccount` object?
3. Class `SavingsAccount` inherits field `balance` from class `BankAccount`, so why does the Java compiler report an error if methods in `SavingsAccount` access this field?
4. What are the names of the methods that can be called on a `SavingsAccount` object? What are the names of the methods that can be called on a `ChequingAccount` object?
5. Why do the `SavingsAccount` and `ChequingAccount` classes both override the `withdraw` method they inherit from `BankAccount`?
6. Why do the `SavingsAccount` and `ChequingAccount` classes not override the `deposit` and `getBalance` methods they inherit from `BankAccount`?

7. Why is the `setInterestRate` method defined in class `SavingsAccount` instead of class `BankAccount`?

8. Currently, the `monthEnd` method is defined in the `SavingsAccount` and `ChequingAccount` classes. Should we instead define this method in class `BankAccount` and remove the method definitions from the subclasses?

What You Learned

1. When you create a `SavingsAccount` object, no `BankAccount` object is created. Instead, you have a single object (an instance of class `SavingsAccount`) that inherits one field (`balance`) and three methods (`deposit`, `withdraw`, and `getBalance`) from `BankAccount`.

2. Each `SavingsAccount` object has three fields: `balance` (which is defined in superclass `BankAccount` and inherited from that class), and `interestRate` and `minBalance` (which are defined in class `SavingsAccount`).

3. Inheritance does not imply visibility. Even though field `balance` is inherited by class `SavingsAccount` from class `BankAccount`, methods in class `SavingsAccount` cannot access `balance`, because it has `private` visibility. Only methods in `BankAccount` can access `balance`.

4. We can call five methods on a `SavingsAccount` object: `deposit`, `withdraw`, `monthEnd`, `getBalance` and `setInterestRate`. We can call four methods on a `ChequingAccount` object: `deposit`, `withdraw`, `monthEnd`, and `getBalance`.

5. Method `withdraw` is defined in class `BankAccount`, but is overridden in the subclasses to implement the specialized withdrawal operations provided by savings and chequing accounts. The implementations of `withdraw` in `SavingsAccount` and `ChequingAccount` must call the `withdraw` method inherited from `BankAccount` to update the account balance.

6. Methods `deposit` and `getBalance` are not overridden by the `SavingsAccount` and `ChequingAccount` classes, because the operations they provide are completely appropriate for instances of those subclasses.

7. Method `setInterestRate` is defined in class `SavingsAccount`, not class `BankAccount`, because it doesn't make sense for all other subclasses of `BankAccount` (for example, `ChequingAccount`) to inherit this method - this operation is appropriate for savings accounts, but not chequing accounts, because chequing accounts don't accumulate interest.

8. The month-end operations on savings accounts and chequing account have nothing on common, so we can't define a `monthEnd` method in `BankAccount` that is appropriate for both types of account. Even if the two implementations of `monthEnd` shared common code that could be moved into a `monthEnd` method in `BankAccount`, we would still need to override this method in the subclasses to implement the month-end operations for

the two subclasses (i.e., calculate the interest and update the minimum monthly balance for a savings account, and reset the withdrawal counter for a chequing account).