3 Understanding Equality

Note: This version of the lab follows the Lab 6 from Spring 2011 that deals with the design of tests in the presence of mutation. Use the classes defined in the **Lab6.zip** file to work on this version of the lab.

Note: This material is covered in pages 321 - 330 in the textbook *How to Design Classes*. Read it carefully.

We now want to define a method that will determine whether the given account is the same as the given account. We may need such method to find the desired account in a list of accounts.

Of course, now that we have the abstract class it would be easy to compare just account number and the name on the account. But, maybe, we want to make sure that the customer's data match the data we have on file exactly - including the balances, the interest rates, and the minimum balances - as applicable.

The design of the method `same` is similar to the technique described in the textbook. The relevant classes and examples that were handed out in the class can be found in the file *Coffee.java*. You may want to look at the code there as you work through this problem.

1. Begin by designing the method `same` for the abstract class `Account` .

2. Make examples that compare all kinds of accounts - both of the same kind and of the different kinds. For the accounts of the same kind you need both the expected `true` answer and the expected `false` answer. Comparing any checking account with another savings account must produce `false`.

3. Now that you have sufficient examples, follow with the design of the `same` method in one of the concrete account classes (for example the `Checking` class). Write the template and think of what data and methods are available to us.

4. You will need a helper method that determines whether the given account is a Checking account. So, design the method `isChecking` that determines whether this account is a checking account. You need to design this method for the whole class hierarchy - the abstract class `Account` and all subclasses. Do the same to define the methods `isSavings` and `isCredit`.

5. We are not done. This helps with the first part of the `same` method. We need another helper method that tells Java that our account is of the specific type. Here is the method header and purpose for the checking account case:

```
// produce a checking account from this account
Checking toChecking();
```

   In the class `Checking` the body will be just

```
// produce a checking account from this account
Checking toChecking(){
  return this; }
```

   Of course, we cannot convert other accounts into checking account, and so the method should throw a `RuntimeException` with the appropriate message. We need the same kind of method for every class that extends the `Account` class.

6. Finally, we can define the body of the `same` method in the class `Checking`:

```
// produce a checking account from this account
boolean same(Account that){
  if (that.isChecking()){
    return that.toChecking().sameChecking(this);
  } else {
    return false;
  }
}
```

   That means, we still need the method `sameChecking` but this only needs to be defined within the `Checking` class and can be defined with a `private` visibility.

   Finish this - with appropriate test cases.

7. Finish designing the `same` method for the other two account classes.

Alternative approaches - bad and good

Note 1 - Incorrect alternative:

The method above can be written with two Java language *features*, the `instanceof` operator and *casting* as follows:

```
// produce a checking account from this account
boolean same(Account that){
  if (that instanceof Checking){
    return ((Checking)that).sameChecking(this);
  } else {
    return false;
  }
}
```

However, this version is problematic and not safe.

If the class `PremiumChecking` extends `Checking`, then any object constructed with a `PremiumChecking` constructor will be an instance of `Checking` and the trouble that can result is illustrated in the example *TestSame.java*. You can make a simple project and run the examples, but we include the output from the *tester* for illustration.

Note 2 - A correct alternative:

In the lecture we have introduced another version that also works correctly. It requires us to add a new method to the `abstract class` for each `class` that `extends` the `abstract class`.

Lecture Notes for Lecture 16 from February 18th, 2010 posted on the wiki show this technique for the classes that represent a list of books (`ILoB`, `MtLoB`, and `ConsLoB`.

Here the methods were:

```
// is this list of book the same as the given empty list of books?
public boolean sameMtLoB(MtLob that)

// is this list of book the same as the given nonempty list of books?
public boolean sameConsLoB(ConsLob that)
```