Mutating Object State and Implementing Equality

6.1 Mutating Object State

Goals

Today we *touch the void*... (sounds creepy right... see the movie, or read the book, to understand how scary the **`void`** can be).

We will focus on the following topics:

- Designing methods that modify (for real) the state of an object
- Designing tests for effectful methods
- Java `RuntimeException`s
- Designing methods that define equality

The Problem

For this Lab we will work with bank accounts again. For our purposes we have *savings accounts*, which must maintain a positive balance, *checking accounts*, which require a minimum balance (not zero), and *credit lines*, which records the balance currently owed and the maximum the customer can borrow. The bank has a list of `Account`s where a customer may deposit or withdraw money. A withdraw from an account cannot reduce the balance below the minimum, and, for credit lines, cause the balance owed to be above the maximum limit. When a customer deposits money to an account, the balance increases, though for a credit line this decreases the amount owed, which cannot drop below zero.

6.1.1 Methods that effect a simple state change

1. Create a Project for this Lab and unzip the files from `Lab6.zip`, from the main lab page.

2. Make examples of `Checking`, `Savings`, and `Credit` accounts. We've started you off with two of them using a different organization than you are used to. We use a `reset()` method to initialize the examples, rather than initializing them in place. Follow the same organization with your examples... more on that later.

3. Discuss several scenarios of making deposits and withdrawals with you partner for each type of account. Make sure you understand when the transaction cannot be completed (i.e., is invalid).

4. Add the method `withdraw` to the `Account` class and implement it in each subclass:

   ```
   // EFFECT: Withdraw the given amount
   // Return the new balance
   abstract int withdraw(int amount);
   ```

   When doing so we encounter a few questions:

   - *Question*: How do we signal that the transaction cannot be completed?

     Answer: **throw** a **new** `RuntimeException` similar to the following:

     ```
     throw new RuntimeException("Over credit limit");
     ```

     Make the message meaningful for your class. You may add some information about the account that caused the problem, the customer name, or the current balance available.

   - *Question*: How do we test that the method throws the expected exception?

     Answer: Suppose the method invocation:

     ```
         this.check1.withdraw(1000)
     ```

     should throw a `RuntimeException` with the message: `"1000 is not available"`. Our test for this exception would then be:

     ```
      t.checkException("Testing withdraw checking",
         new RuntimeException("1000 is not available"),
         this.check1,
         "withdraw",
         1000);
     ```

The first argument is a `String` that describes what we are testing — it is optional and can be left out. The second argument gives the `Exception` our method invocation should throw (the messages must match exactly). The third argument is the instance that invokes the method, the fourth argument is the method name. After that we list as many arguments as the method consumes, separated by commas.

- *Question*: How do we test the correct method behavior when the transaction goes through?

  Answer: We look at the purpose and effect statements. Because the method produces a value as well an *effect* (changes the state of the object), we must test both aspects.

  We first define instances and add them to our `reset` method. We use the `reset` method to initialize our data since the examples may change during each test.

  ```
  // Test the withdraw method(s)
  void testWithdraw(Tester t){
     reset();
     t.checkExpect(check1.withdraw(25), 75);
     t.checkExpect(check1, new Checking(1, 75,
                   "First Checking Account", 20));
     reset();
  }
  ```

  Notice that we use the `reset` method twice. At the start we make sure that the data we use has the correct values *before* the tests are invoked, and *after* the test(s) we reset the data to the original values (since we always need to reset before testing, we can leave the second reset off, though later we may to different tasks before and after testing).

  In this case there are two tests we have to perform: the first is what we have done in the past — comparing the value produced by the method with the expected value. The second test verifies that the *state* of the object did indeed change as expected.

  Try the following *incorrect* implementations of the `withdraw` method in the `Checking` class to see why all this testing is necessary:

```
// Missing Effect...
int withdraw(int amount){
   return this.balance - amount;
}

// Wrong return value...
int withdraw(int amount){
   this.balance = this.balance - amount;
   return amount;
}

// Correct, but no exception!
int withdraw(int amount){
   this.balance = this.balance - amount;
   return this.balance;
}
```

Of course, we need to implement and test the method in each `Account` class: the `Savings` and `Credit` classes as well.

5. Add the method `deposit` to the `Account` class and implement it in all subclasses. Remember, what happens in the `Credit` case when the `balance` would become negative (no more debt)?

```
// EFFECT: Deposit the given funds into this account
// Return the new balance
abstract int deposit(int funds);
```

Make sure your tests are defined carefully as before.

6.1.2 Methods that change the state of structured data

The `Bank` class keeps track of all accounts.

1. Design the method `openAcct` to `Bank` that allow the customer to open a new account in the bank.

```
// EFFECT: Add a new account to this Bank
void add(Account acct){ ... }
```

Make sure you design your tests carefully.

2. Design the method `deposit` that deposits the given amount to the account with the given name and account number. Make sure you

take exception to any problems, e.g., no such account, or a transaction that cannot be completed.

Make sure to design your tests carefully.

3. Design the method `withdraw` that withdraws the given amount from the account with the given account number. Make sure you *take exception* to any problems, e.g., no such account, or a transaction that cannot be completed.

   Make sure to design your tests carefully.

4. Design the method `removeAccount` that will remove the account with the given account number from the list of accounts in this `Bank`.

   ```
   // EFFECT: Remove the given account from this Bank
   void removeAccount(int acctNo){ ... }
   ```

   Hint: Throw an exception if the account is not found, and ***follow the Design Recipe!***

6.2 Understanding Equality

Note: This material is covered in pages 321 - 330 in the textbook. Read it carefully.

Our object is to define a method that will determine whether a given `Account` is the same as this `account`. We may need such a method to find a desired account within the `Bank`.

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 we want to make sure that all the customer's data matches what we have on file exactly, including balances, interest rates, etc.

6.2.1 Implementing `same`

We will design a `same` method similar to that described in the second part of the lecture, *Equality by safe casting*. The relevant examples can be found in the lecture notes. You may want to look at the code there as you work through this problem.

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

2. Make examples that compare all kinds of accounts: of the same kind (e.g., `Checking` vs. `Checking`) and of different kinds (e.g., `Savings` vs. `Credit`). For the accounts of the same kind you should test both **`true`** and **`false`** cases. Comparing different kinds of accounts should always 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.

4. As in lecture, you need two different helper methods: one that determines whether the given account is a `Checking/Savings/Credit` account, and one that converts this account into the desired type. Design the methods `isChecking`, `isSavings`, and `isCredit`, that determine whether this account is a checking/savings/credit account, respectively.

5. Design the methods `toChecking`, `toSavings`, and `toCredit`, that convert this account into a checking/savings/credit account, respectively. Header and purpose for the checking account case:

   ```
   // Convert this Account into a Checking
   abstract Checking toChecking();
   ```

   In the `Checking` class the body will be just

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

   While the others should throw a `RuntimeException` with the appropriate message.

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

   ```
   // Is the given Account the same as this?
   boolean same(Account that){
      if(that.isChecking()){
         return this.sameChecking(that.toChecking());
      }else{
         return false;
      }
   }
   ```

7. We still need the method `sameChecking` but this only needs to be defined within `Checking`, and can be defined with **`private`** (or **`protected`**) visibility.

8. Complete the design of the `same` methods (including `sameChecking`, `sameSavings`, `sameCredit`) for the other two account classes.

6.2.2 Alternative approaches: bad and good

Bad Option - Incorrect alternative:

The method above can be incorrectly written with two features of the Java language: the **`instanceof`** operator and *casting*. In the `Checking` class this style method would look like the following:

```
// Is the given Account the same as this Checking?
boolean same(Account that){
   if(that instanceof Checking){
      return this.sameChecking((Checking)that);
   }else{
      return false;
   }
}
```

However, this version introduces bugs!

The issue is that any class that later extends `Checking`, say a `PremiumChecking` class, will also be considered a `Checking` instance by the **`instanceof`** operator.

If we implement a similar `same` method in `PremiumChecking`:

```
// Is the given Account the same as this PremiumChecking?
boolean same(Account that){
   if(that instanceof PremiumChecking){
     return this.samePremChecking((PremiumChecking)that);
   }else{
     return false;
   }
}
```

Then now only `PremiumChecking` objects can be the `same` as other `PremiumChecking` instances,

But `Checking` instances can be the `same` as `PremiumChecking` instances!!

These kinds of bugs can cause serious problems. This issue is also illustrated in the example file `BadSame.java` from the lab main page. You can add the file to your project and run the example, but we have included the `tester` output for illustration.

Good Option - A Correct alternative:

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

For our `IShape` hierarchy, the methods were:

```
// Is that Circle the same as this Shape?
public boolean sameCirc(Circ that);
// Is that Rect the same as this Shape?
public boolean sameRect(Rect that);
// Is that Combo the same as this Shape?
public boolean sameCombo(Combo that);
```

This technique is popular in object-oriented languages like Java and is known as *double dispatch* (or *callbacks*).