7 Mutating Object State

Goals

Today we *touch the void*. (Go, see the movie, or read the book, to understand how scary the *void* can be.) We will focus on the following four topics:

- Designing methods that change the state of an object
- Designing tests for these methods
- Java Runtime Exceptions
- Designing methods that define equality of two objects

Rather than looking for just one *correct* solution to a problem, we will examine several possible ways of dealing with a problem and try to compare the solutions.

The Problem

We will work with bank accounts: checking, savings, or credit line. The bank has a list of these accounts and the customer may deposit some money or withdraw some money. Checking accounts require that the customer keeps a minimum balance, and so never withdraws all money in the account. Credit line records the balance as the amount currently owed, and it also remembers the maximum the customer can borrow. Customer can withdraw money, if adding the desired amount does not increase the balance owed to be above the maximum limit. When the customer deposits money to the credit line account, it decreases the amount owed by the deposited amount. Customer cannot overpay the debt in the credit line.

7.1 Methods that effect a simple state change

A. Create a Java Project and add following files to it's source directory.

   - *Account.java*
   - *Checking.java*
   - *Savings.java*
   - *Credit.java*
   - *Bank.java*

- *AccountList.java*
- *Examples.java*

B. Make several examples of data for *Checking*, *Savings*, and *Credit* Accounts.

C. Describe to your partner several scenarios of making deposits and withdrawals, to make sure you know when the transaction cannot be completed.

D. Add the method `deposit` to the `abstract class Account` and implement it in all subclasses:

```
//EFFECT: Add the given amount to this account
//Return the new balance
int deposit(int amount);
```

When doing so we encounter several problems:

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

  Answer: Throw a `RuntimeException` changing appropriately the following code:

  ```
  throw new RuntimeException(
            "Balance too low: " + this.balance);
  ```

  Make the message meaningful. You may add to the message some information about the account that caused the problem - the customer name, or the current balance available, or how much more would there need to be in the account for the transaction to go through.

- *Question:* How do we test that the method will throw the expected exception with the expected message?

  Answer: Suppose the method invocation:

  `this.bobAcct.withdraw(1000)`

  throws a `RuntimeException` with the message:

  `"1000 is not available"`.

  The test would then be:

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

The first argument is a `String` that describes what we are testing — it is optional and can be omitted. The second argument defines the `Exception` our method invocation should throw. The third argument is the instance that invokes the method, the fourth argument is the method name, and after that we list as many arguments as the method consumes — all separated by commas. It could be no arguments, or five arguments — it does not matter. For our method that performs the deposit, it will be just the amount we wish to deposit.

- *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 as has an *effect* on the state of the object that invoked, we must test both parts.

  We first define instances of data we wish to use. We also define the method `reset` that initializes the values for the data we expect to work with and may change during the tests. We can then design the test as follows (assuming that the `this.check1` is the instance that should invoke the method:

```
//Tests the deposit methods inside certain accounts.▌
void testDeposit(Tester t){
  reset();
  t.checkExpect(check1.deposit(100), 100);
  t.checkExpect(check1,
    new Checking(0001, 100, "First Checking Account", 0));▌
  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 method is invoked, after the test we reset the data to the original values, so that the test would not affect any other part of the program. Sometimes these two method invocation are divided into two tasks: *setup* and *tear-down*. This is true of the setup actually prepares the data to have some special values before the method is invoked, but afterwards, we want to reset the values to *more normal* state.

There are two tests we have performed. The first one is just like what we have done in the past — we compare the value produced by the method invocation with the expected value. The second test verifies that the state of the object we were modifying did indeed change as expected.

Try the following *incorrect* implementations in the `Checking` class of this method to see why these tests are necessary:

```
//EFFECT: Add the given amount to this account
//Return the new balance
int deposit(int amount){
  return this.balance + amount;
}

//EFFECT: Add the given amount to this account
//Return the new balance
int deposit(int amount){
  this.balance = balance + amount;
  return amount;
}

//EFFECT: Add the given amount to this account
//Return the new balance
int deposit(int amount){
  return 20 + (this.balance = balance + amount);
}

//EFFECT: Add the given amount to this account
//Return the new balance
int deposit(int amount){
  return this.balance = balance + amount;
}
```

Only one of these is correct. Notice the use of the assignment as the return value and as the value used in an arithmetic expression. The result of the assignment is always the value assigned to the identifier on the left-hand side.

Of course, we need to test the method in every class in the union: the `Savings` class as well as the `CreditLine` class.

E. Add the method `withdraw` to the `abstract class Account` and implement it in all subclasses:

```
// EFFECT: Withdraw the given funds from this account
// Return the new balance
int withdraw(int funds);
```

Make sure your tests are defined as carefully as we have done in the previous case.

7.2 Methods that change the state of structured data

The class `Bank` keeps track of all accounts.

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

```
// EFFECT:
// add a new account to the list of accounts kept by this bank
void add(Account acct)
```

Make sure you design your tests carefully.

B. Design the method `deposit` that deposits the given amount to the account with the given name and account number.

Make sure you report any problems, such as no such account, or that the transaction cannot be completed.

Make sure you design your tests carefully.

C. Design the method `withdraw` that withdraws the given amount from the account with the given name and account number.

Make sure you report any problems, such as no such account, or that the transaction cannot be completed.

Make sure you design your tests carefully.

D. Design the method `removeAccount` that will remove the account with the given account id and the given name from the list of accounts in a bank.

```
void removeAccount(int acctNo, String name)
```

Hint: Throw an exception if the account is not found

Follow the Design Recipe!

7.3 Understanding Equality

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

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

B. 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`.

C. 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.

D. 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`.

E. 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.

F. 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.

G. 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)
```