6 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 mutable linked lists and methods for mutable linked lists.

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.

6.1 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.

6.2 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!

6.3 Designing Mutable Lists

Start a new project *LinkedLists* and import into it all files in the *Lists.zip* file. You should have three files:

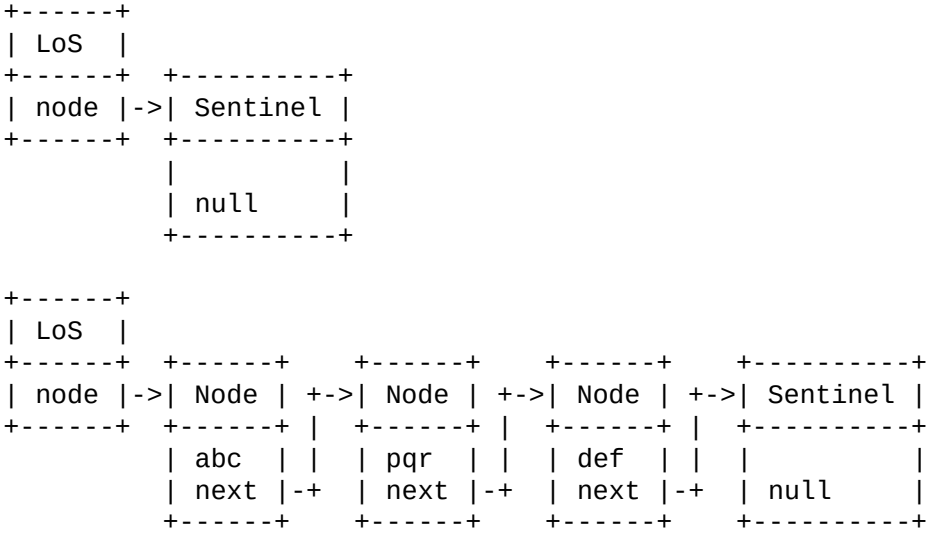
- *Node.java*
- *LoS.java*
- *ExamplesLoS.java*

In the previous example we needed a wrapper class to implement a mutable list of accounts, because we could not mutate an empty list into a nonempty list. This example shows a similar technique, but instead of replacing a large part of the original list with a new list, we will change the structure of the list without modifying anything except the nodes involved in inserting or removing the nodes.

The classes `LoS` and `Node` represent a collection of `String`s organized as a list. Instead of having two different classes for the empty list and for nonempty list, we have a class that represents a node in a list (similar to our *Cons* classes that contained the data and a link to the next item), and a subclass that represents the last node in the list (a sentinel).

The list `LoS` will always contain one `Node`. If the list is empty, the `Node` is the special *sentinel* node.

The following picture illustrates the structure of an empty *linked list* and a *linked list* after we added three `String`s, "def", "pqr", and "abc":



A. Study the code and make sure you understand what is going on. Add an example to each test defined in the `ExamplesLoS` class.

B. Add tests for the methods that are not tested.

C. Design the method `removeNode` for the class `LoS` that removes the node that contains as `data` the given `String`.

D. Design the method `size` that counts the number of nodes in a list (`LoS`), not including the sentinel node.