8 Mutating Object State

8.1 Goals

Today, we are attempting to give you hands on practice with four concepts that we have introduced to you over the last two weeks.

- Java Runtime Exceptions
- Constructing and Deconstructing Sets of Data
- Modifying data
- List Item Removal

As you work through today's lab, take care to think of the examples that we have gone over in class, and how they can be extrapolated into the lab. What similarities exist? Where are the differences? How do we take the knowledge we have and solve those differences?

8.2 The Problem

Imagine that today you are tasked by **Northeastern Bank Corp** to create a computer program that serves the following tasks.

- Handles data for *Checking*, *Savings*, and *Credit* Accounts.
- Has the ability to add new customers
- Has the ability to modify *Account* information for a particular customer.
- Has the ability to remove accounts
- Throws a Java RuntimeException whenever an "unacceptable" condition occurs (Too few funds in the account, incorrect account numbers, etc.)

A. Please begin by creating a Java Project, of your choosing, that contains the following files in it's source directory.

   - *Customer.java*
   - *Account.java*

- *Checking.java*
- *Savings.java*
- *Credit.java*
- *Bank.java*
- *AccountList.java*
- *CustomerList.java*
- *Examples.java*

B. Add the method `deposit` to the Abstract class **Account** and implement it in all subclasses:

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

C. 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);
```

D. When you are writing your examples for these methods, make sure you also include examples of classes that extend Account and how they can utilize these methods. *Hint, how does **Credit** differ from the other classes that extend **Account**?*

E. Next, let's deal with the **Customer** class and the **ILoA** class hierarchy that represents a list of accounts. Each customer has a name and a list of accounts he/she holds. Complete the class definition for the class `Customer`.

F. Add the method `add` to **ILoA** and implement it as needed. It should add the given account to our list of accounts.

```
ILoA add(Account acct)
```

G. Design the method `addAccount` for the **Customer** class. It should add the given account to the customer's list of accounts.

```
void addAccount(Account acct)
```

H. Now, let's follow the same methodology and set up the **Bank** class. A **Bank** has a list of all accounts. We already know how to add an account to a list of accounts. However, we also want to be able to remove an account from a list of accounts.

   Design the method `removeAccount` that will remove the account with the given account id from the list of accounts.

```
ILoA removeAccount(int acctNo)
```

   Hint: Throw an exception if the account is not found

I. The bank has one list of accounts for every branch. These are the accounts created in the branch. Each branch then has additional information about its customers (in the `CustomerList`). The branches share the account lists — and so the central list of lists of accounts has one account list for each branch. If we want to make changes in the account list of one branch, we cannot produce a new list of accounts. To solve this problem we build a *wrapper* class for account lists called `AccountList`. Its only data (for now) is an instance of `ILoA`.

   Complete the definition of the class `AccountList` and make examples of its data.

J. Design the method that changes the `AccountList` by adding to it the given account.

```
void addAccount(Account acct)
```

K. Design the method that changes the `AccountList` by removing from it the account with the given account number.

```
void removeAccount(int acctNo)
```

   Follow the Design Recipe!