**Exceptions exercises**

v. 20171114

For this lesson, we’ll create a simple program, a very basic bookkeeping application:

In the model layer, we put a class, called Account. It should have

* 2 fields: name, and balance.
* A constructor, that takes the name, and the initial balance as parameters
* getName, getBalance, deposit(double), withdraw(double)

This means that name can’t be changed once the account is created, and we can only change the balance by depositing or by withdrawing

1. Throw an IllegalArgumentExcception if one tries to withdraw more money than the balance of the account. Test.
2. Prohibit the creation of accounts without names: Throw an IllegalArgumentException if the constructor is passed a null value for name. Test.
3. Prohibit the creation of accounts with a negative balance. Throw an IllegalArgumentException if the constructor is passed a negativ value for initial balance. Test.
4. Implement your own exception: BalanceUnderrunException that extends IllegalArgumentException. Make the withdraw method throw this instead. Test.

In the model layer, we put a new class: Accounting. **It must be a singleton,** It should have

* An ArrayList of Account objects
* An addAccount(Account) method

1. Implement and throw a DuplicateAccountNameException if the account that you try to add to the Accounting object already exists among the accounts. Test.

The control layer has a single class, called AccountCtrl. Implement methods that let you deoposit, withdraw, and transfer money between accounts:

* depositFunds(String accountName, double amount)
* withdrawFunds(Strging accountName, double amount)
* transferFunds(String accountNameFrom, String accountNameTo, double amount)

1. What happens if you try to withdraw too much money? How do we handle that?
2. What happens, if you try to move money from one account to another, if the source account has less funds than the amount specified? Does the money disappear? Make sure that your transferFunds(…) method doesn’t lose money if the transaction fails due to the lack of funds.

Add a text user interface to the tui layer. You name your class. It should have a menu, where you provide the following options for the user: create account, deposit, withdraw, transfer funds, and show all accounts. If you have made a working TUI library, you may want to use it. If this is too much to implement, replace the text user interface with a simple method that simulates the usage by calling the control layer (a TryMe class).

* Implement the creation of accounts – you may have to add methods to your controller class.
* Implement the transfer of funds between accounts – you may have to add methods to your controller class.

Now, we’ll look at implementing a robust application.

1. Where do we handle all the exceptions that the lower layers may throw?
2. Implement any catch clauses that the system can’t handle/recover from intelligently, and thus needs the users to help it resolve the issue.

Persistence (**After having covered Serializable and disk I/O**)

1. Implement a Serializable version of the project.
2. Add functionality that can save the model layer objects to disk and load them again.