TDT4165 PROGRAMMING LANGUAGES

Scala Project
Concurrency Paradigms

Fall 2017

Preliminaries

1 Deliverables and Deadline

To submit your solution, upload the files for each task in the format *taskX.zip*, and any other modified/added files to Blackboard before the end of November 9th. Your code should be presented during lab hours before the end of November 16th. For your submission to be approved, 70% test coverage is required for each task.

2 Running the Tests

To run the tests from the command line, `cd` into the taskX-project directory, and run the `sbt test` command. If you have not yet installed `sbt`, visit http://www.scala-sbt.org/release/docs/Setup.html and follow the installation instructions for your OS.

3 Description

Traditional online banking applications are currently experiencing great competition from new players in the market who are offering direct transactions with a few seconds of response time.
Banks are therefore looking at possibilities of changing their traditional method which involves batch transactions at given times of day with hours in between. They must now update their software systems to adapt to the current demand, which means transactions must be handled in real-time.

Your overall task for this project is to implement features of a real-time banking transaction system.

4 Additional information

The tasks are solvable by adding functionality to the framework. Changing the given code isn't necessary, and should be avoided. Additionally, try to use *val* instead of *var* whenever possible.
The project is to be solved in groups. The maximum, and recommended, number of people is 3 per group.

Task 1: Threads

The first task of the project will familiarize you with simple threading and an application thereof.

The file structure of task1-project is presented below.

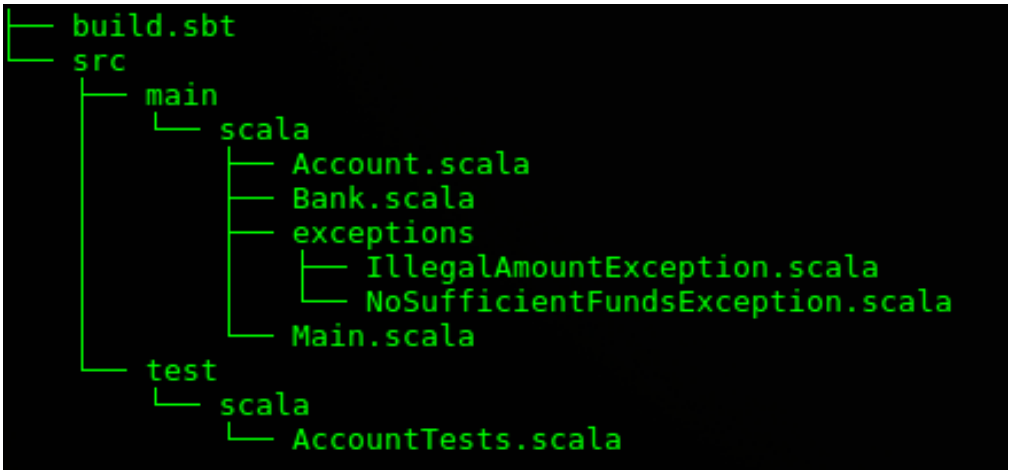

Figure 1: File Structure

For this task, you will be working in Account.scala, Bank.scala, and Main.scala. The main objective is to implement missing functionality and pass all of the tests.

Account In the Account class, one should be able to withdraw (take out) or deposit (insert) a certain amount of money, which will change the total balance in the account. Throw suitable exceptions for invalid states.

Bank In Bank, a method for transactions between accounts should be implemented. Throw suitable exceptions for invalid states.

Main In Main, we have provided an implementation of a thread that you can play around with to familiarize with the concept, and test your code. Use `sbt run` in the command line to run the Main file. Editing this file does not affect the tests.

The implementation of Bank.getUniqueId (shown below) is not thread safe. Why? How would you fix it?

```
private var idCounter: Int = 0

def getUniqueId(): Int = {
    idCounter += 1
    idCounter
}
```

Required Tests

7 of 10 tests need to succeed. Test "Correct balance amount after several withdrawals and deposits" or "Correct balance amounts after several transfers" *must* run successfully.

Task 2: Executors

The file structure of task2-project is is presented below.

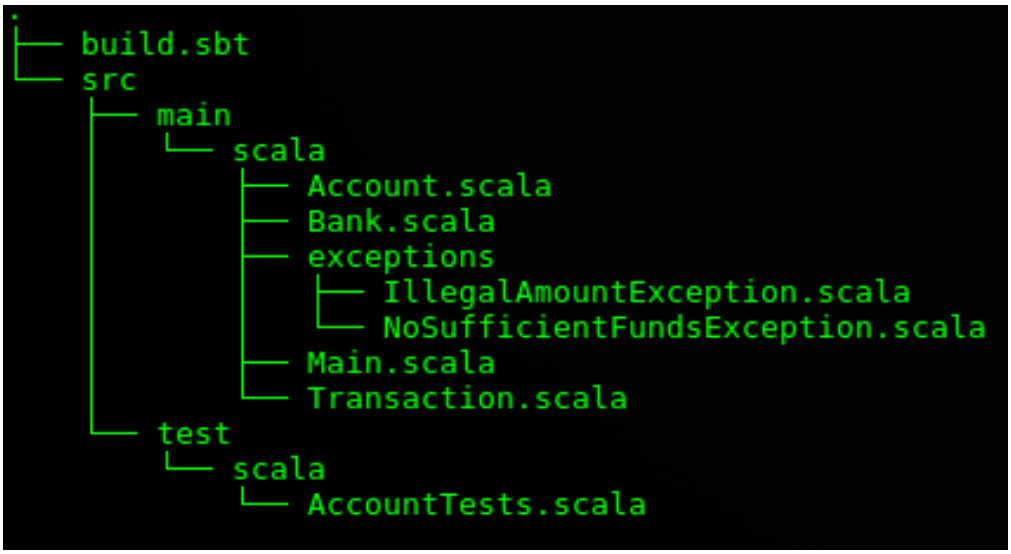

Figure 2: File Structure

In this task, you will implement transactions using the Executor abstraction. New transactions should be added to a concurrent collection TransactionQueue, and from here it should be handled by the executor in Bank. For this task, you will be working in Account.scala, Bank.scala and Transaction.scala. The main objective is to implement missing functionality and pass all of the tests. Account.scala can be filled in from task 1.

Account The unimplemented functions are similar as in task 1, but they now need to be thread safe.

Bank getUniqueId is now called generateAccountId, and it should be thread safe. (Hint: atomicity.) You will also have to implement continuous processing of new transactions in the transaction queue.

Transaction and TransactionQueue
TransactionQueue should be a concurrent collection in which common queue-operations should be implemented (these are explained in the source code). A Transaction object represents a transaction between two accounts and may succeed or fail, and your task here is to extend the run method. Failed transactions should be retried up to a variable amount of times (allowedAttempts).

Required Tests

9 of the 13 tests need to succeed. Test 7 *must* run successfully.
Note that the test suite can take a few *minutes* to complete.
If the test takes even longer, you may have a deadlock in your code.

Task 3: Actors

The file structure of task3-project is is presented below.

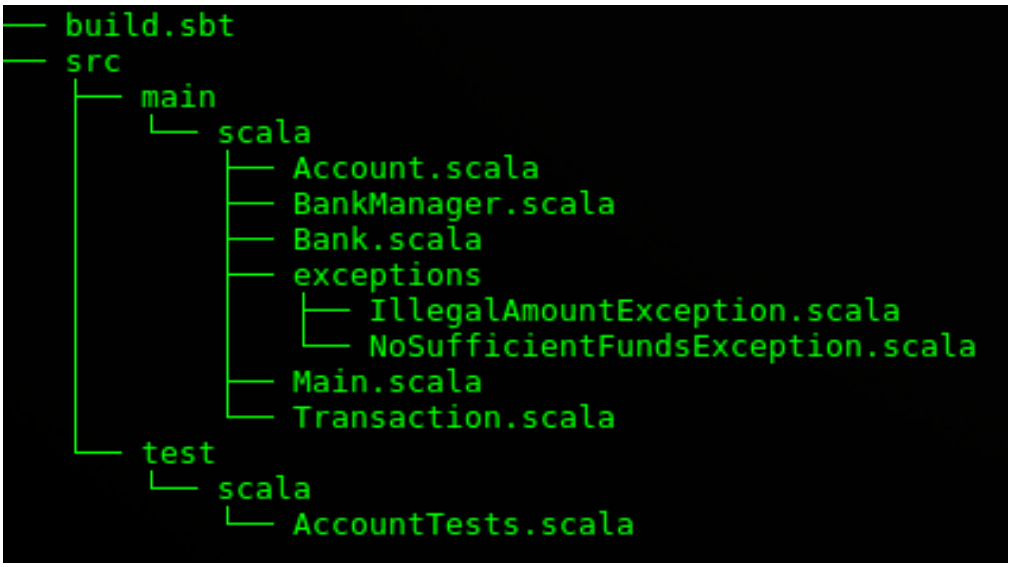

Figure 3: File Structure

In the final task of the Scala project, you will still be working with transactions, but this time you will need to support transactions between accounts in different Bank-instances, using actors. It is highly recommended that you read up on Actors before starting this task, if you are not already familiar with it. You will be working in Bank.scala and Account.scala. Also provided is BankManager.scala, which is the Actor System that you will need when looking up other Banks or Accounts. Below is some important information about what you should implement in this assignment:

1 All Accounts should be initialized with a unique AccountID which is exactly 4 characters long, and each Account belongs to exactly one Bank.

2 All Banks should be initialized with a unique BankID which is exactly 4 characters long.

3 An account number is defined as the BankID concatenated with AccountID (e.g. 40012001: Account with AccountID 2001 belongs to a bank with BankID 4001). Account numbers are represented as Strings in the program, for easier manipulation.

4 A transaction works as follows: the sending Account (A) calls the transferTo-method. transferTo withdraws the correct amount from the A, adds a new Transaction-instance (let's call this t) to a HashMap internal to A, and forwards t to the A's Bank. The Bank should then forward t either to a different Bank or an Account, depending on whether t is internal or not. If t is external and sent to a different Bank, that Bank should forward t to the correct Account.

When the receiving Account (B) has received t, B should process t, and send a TransactionRequestReceipt, saying that t succeeded, back to the A, the same way t was sent (only backwards).

If the transaction somehow fails on the way (e.g., if a Bank or Account does not exist), a TransactionRequestReceipt saying that t failed should be sent back to A from the point of failure.

When A has received a TransactionRequestReceipt, it should update the information about t in the HashMap that the transaction was stored in earlier.

4 A transaction holds the following information:

1. from: String
   The account number of the sending Account.
2. to: String
   The account number of the receiver Account. An internal transaction does not need to include the BankID in the account number (e.g., Account A with account number 40012001 wants to transfer to Account B with account number 40012002, they are in the same Bank and to should be 2002; whereas if a transaction from Account A to Account C with account number 50012002 (different Bank), to should be 50012002).
3. amount: Double
   The amount of money that should be transferred.
4. status: TransactionStatus.Value
   An enum representing the current status of the Transaction (PENDING, SUCCESS, or FAILED).
5. id: String
   A unique ID.
6. receiptReceived: Boolean
   When the sending Account receives the receipt from the receiving Account, this value should be true

Required Tests

13 of 17 tests need to succeed. Two tests should succeed by default.