TDT4165 PROGRAMMING LANGUAGES

Scala Project

Fall 2021

Scala is both a functional and object-oriented programming language. This task will introduce you to Scala and familiarize you with some basic concepts and syntax before proceeding to the Scala project.

Relevant readings: "Learn concurrent programming in Scala" PDF that is uploaded together with the project (especially the chapters about threads).
The easiest way to run Scala is to install **sbt** at http://www.scala-sbt.org/
To start a project, create a new folder with a file called `Main.scala` with the following content:

```
object Hello extends App{
    println("Hello World")
}
```

Navigate to the folder you just created and run your code using `sbt run` in the terminal.

Delivery

This project consists of two deliveries, counting towards the 5 out of 7 deliveries necessary to get a grading in this course.

The project will be done in groups of maximum of 3. In BlackBoard, there will be two different kinds of groups; **Prapared** and **Random**.

The **Prepared** groups are for those who wish to choose their own groups, so those who wish to be in the same group can join any of these groups.

The **Random** groups are for those who wish to be put into a group with random members. To join these groups, join a group with available slots, with the lowest number, starting from 1, i.e. if **Random** group 1 is full, group 2 has 1 slot avaiable, and group 3 is emtpy, you join group 2 first.

Each delivery should include a `.pdf` file containing a section for each task. For each task, the PDF should describe the implementation, or include a screenshot of the code, as well as answer any theoretical questions. You can use the template found on BlackBoard, under "Coursework"/"Latex template for PDF's", to generate your PDF file.

Delivery 1 for the project consists of **Task 1: Scala introduction** and **Task 2: Concurrency in Scala**. This delivery consists of getting used to using Scala, to be able to do delivery 2 later.
For delivery 1, please deliver a zip file containing the file(s) for these two tasks, and the `.pdf` file.

Delivery 2 for the project consists of **Project task 1: Preliminaries**, **Project task 2: Creating the bank**, and **Project task 3: Actually solving the bank problem**. This delivery consists of implementing a rigid banking system, with atomized transactions, and error handling. While it is not necessary to finish delivery 1 to finish delivery 2, it is highly recommended, in order to familiarize yourself with using Scala.

For delivery 2, please deliver a zip file of the entire project directory with the `build.sbt` file as well as the entire `src/` directory inside, in addition to the `.pdf` file. Do not deliver the generated `target` and `project` directories.

Figure 1 shows an example directory structure of the entire project.

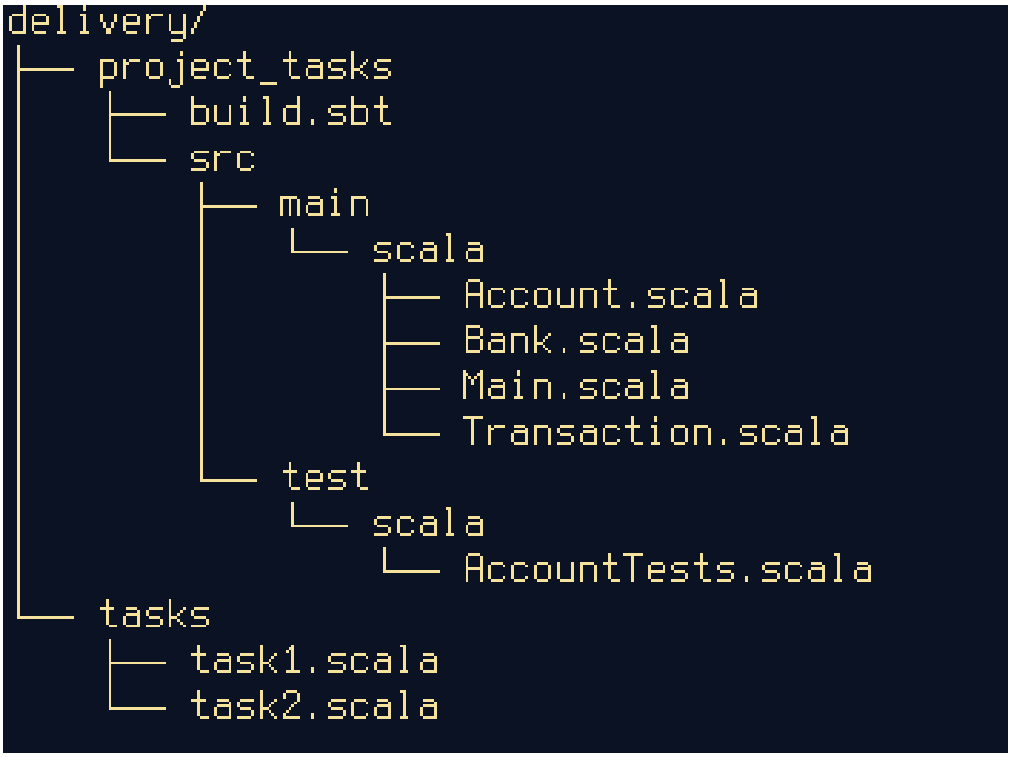

Figure 1: File Structure

Evaluation

Each of the deliveries will have a total of 100 points. To get a delivery as Approved, you need at least 50 points. Points for each task and sub-task is stated in the exercise. You will be awarded full score for each correct implementation of each task. Points may be deduced from this score as a percentage of the available points for the given taks in cases of:

- Code does not run without modifications, but is otherwise correct. (20% deduction)
- The implementation is correct, but is overly complext, long or redundant (20% deduction)
- Unreasonable indentation of code. (20% deduction)
- Functions or variables have names that are not meaningful (20% deduction)

In addition, 1 point will be deducted from each project task score for each failed test related to that task.

While there are no individual deliveries for the project, it is expected that each group member contributes to the project, and understands how the delivered project is working, as the knowledge from this project could be relevant later.

Task 1: Scala introduction (53 p)

To read about the basic Scala syntax, feel free to take a look at the Scala documentation at https://docs.scala-lang.org/tour/basics.htmli or read the `Learn Concurrent Programming in Scala.pdf` that was included with this project.

(a) Generate an array containing the values 1 up to (and including 50 using a **for** loop. (10p)

(b) Create a function that sums the elements in an array of integers using a **for** loop. (13p)

(c) Create a function that sums the elements in an array of integers using recursion. (13p)

(d) Create a function to compute the nth Fibonacci number using recursion without using memoization (or other optimizations). Use `BigInt` instead of `Int`. What is the difference between these two data types? (17p)

Task 2: Concurrency in Scala (47 p)

One of the most important goals in the Scala porject is to learn concurrency programming. Here, it is done by using threads.

You can read more about how to program threads in Scala at https://twitter.github.io/scala_school/concurrency.html

(a) Create a function that takes as argument a function and returns a Thread initialized with the input function. Make sure that the returned thread is not started. (10p)

(b) Given the following code snippet: (10p)

```
private var counter: Int = 0
def increaseCounter(): Unit = {
    counter += 1
}
```

Create a function that prints the current `counter` variable. Start three threads, two that initialize `increaseCounter` and one that initialize the print function. Run your program a few times and notice the print output. What is this phenomenon called? Give one example of a situation where it can be problematic.

(c) Change `increaseCounter` so that it is thread-safe. Hint: atomicity. (13p)

(d) One problem you will often meet in concurrency programming is deadlock. What is deadlock, and what can be done to prevent it? Write in Scala an example of a deadlock using `lazy val`. (14p)

Scala project

Introduction

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 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 simplified and scaled down real-time banking transaction system.

The code is commented with TODOs to help you find the correct place to write your code for each task.

Running the tests

The project comes with some tests to help both you and us in evaluating the project. If all tests pass, your implementation is probably correct^{TM}. In the project root directory (the directory with the `build.sbt` file) run `sbt test` to install dependencies and verify that your installation works. The tests should not fail, and not even compile (they will after task 1.3).

There is no meaningful main program in the handout, so running `sbt run` won't do much. Feel free to use it for experimenting.

Project task 1: Preliminaries (28p)

This task will set up a few utility functions that might help you later on.

1.1 Implementing the TransactionQueue (7p)

In the file `Transaction.scala` you'll see a class definition of `TransactionQueue`. This class needs to be implemented. The class needs a datastructure to hold the transactions. A queue is sufficient. The functions that are defined also need to be implemented in a thread-safe manner. Wrapping existing Queue-functions is encouraged.

- Define a datastructure to hold transactions.
- Implement functions of `TransactionQueue` in a thread-safe manner.

You are not required to use these functions later, but they might prove useful.

1.2 Account funcitons (14p)

In the file `Account.scala`, you'll see three functions that aren't implemented: `withdraw`, `deposit` and `getBalanceAmount`.

- `withdraw` removes an amount of money from the account.
- `deposit` inserts an amount of money to the account.
- `getBalanceAmount` returns the amount of funds in the account.

1.3 Eliminating exceptions (7p)

We also want our `deposit` and `withdraw` functions to fail gracefully in case of errors. make sure that illegal transaction amounts are causing the functions to fail. Exceptions are bad - read the section below on the `Either` datatype and see how it can be used instead of exceptions and program crashes.

If a function fails, make sure it is atomic - meaning that no money is lost or transferred in the case of a failure. The functions also need to be made thread safe.

- `withdraw` should fail if we withdraw a negative amount or if we see a withdrawal larger than the available funds.

- `deposit` should fail if we deposit a negative amount.
- Both should be thread safe.
- Both should return an `Either` datatype and not throw exceptions.

Tests 1-6 should pass now.

Notes on the `Either` datatype

The `Either` type is useful to represent whether or not a function succeeded or not. It consists of a `Left` and a `Right` type, for example `Either[Unit,String]` - this means that the type either holds nothing (`Unit`) or a `String`. We use this to say that "The functions either returns nothing, indicating success, or a string describing the failure".

Choosing whether Left of Right means success is usually up to you, but for the sake of automated tests use Left to indicate success and Right to indicate failure.

Below is an example of how `Either` can be used in a function.

```
def wants_a_positive_number(number: Int): Either[Int, String] = {
    if number < 0 return Right("This is not a positive number")
    Left(number)
}
...
val result = wants_a_positive_number(5)
result match{
    case Right(string) => println(string)
    case Left(number) => println(number)
}
```

Project task 2: Creating the bank (21p)

In the `Bank.scala` you'll see two incomplete functions.

- `addTransactionToQueue` creates a new transaction object and places it in the `transactionQueue`. This function should also make the system start processing transactions concurrently.
- `processTransactions` runs through the `transactionQueue` and starts each transaction one at a time. If a transactions' status is pending, push it back to the queue and recursively call `processTransactions`. Otherwise, the transaction has either failed, or succeeded, and should be put in the processed transactions queue.

Project task 3: Actually solving the bank problem (51p)

Back in the file `Transactions.scala` there is still work to be done. The `run` function, containing another function and a call to that functino needs to be finished. The goal of `doTransaction` is to transfer money safely, which means withdrawing money from one account and depositing it to the other account.

Each transaction is allowed to try to complete several times, indicated by the `allowedAttempts` variable. A transactions status is `PENDING` till it has either succeeded or used up all its attempts.

If you implemented error handing with the `Either` datatype, you can use pattern matching - otherwise you may have to `try/except` the exceptions.