Scala project

NTNU

TDT4165 fall 2019

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), and the Scala guest lecture with Amund Murstad from finn.no.

The easiest way to run Scala is to install **sbt** at `http://www.scala-sbt.org/download.html`. Scala builds on top of Java, so you will need to install a Java environment as well.

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

Please deliver a zip file containing two directories: one with the file(s) for the first two tasks and one with the entire project directory with the `build.sbt` file as well as the entire `src/` directory inside. Do not deliver the generated `target` and `project` directories. Figure 1 shows an example directory structure of this.

Evaluation

You will get a score from 0 to 100 on this exercise. This score will later be converted into a final assignment score. Information on how this conversion will be done will be published on BlackBoard. Points for each task and subtask is stated in the exercise. You will be awarded full score for each correct

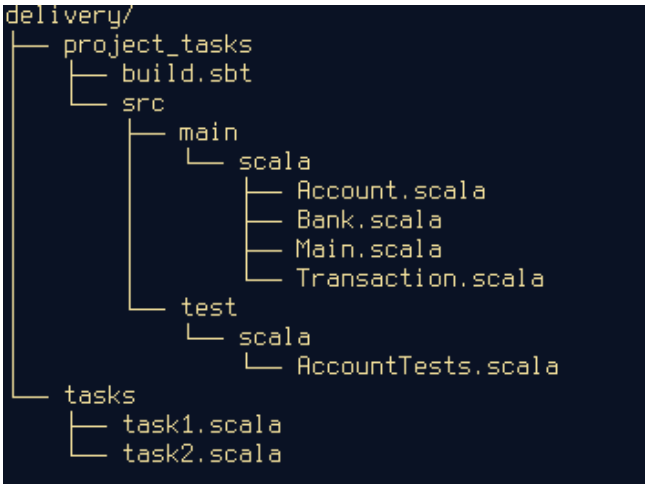

Figure 1: example file structure of your delivery

implementation of each task. Points may be deducted from this score as a percentage of the available points for the given task:

- Code does not run without modifications, but is otherwise correct. (20% deduction)
- The implementation is correct but is overly complex, 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 tasks score for each failed test related to that task.

Task 1: Scala Introduction (16 p)

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

(a) Generate an array containing the values 1 up to (and including) 50 using a for loop. (3p)

(b) Create a function that sums the elements in an array of integers using a for loop. (4p)

(c) Create a function that sums the elements in an array of integers using recursion. (4p)

(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? (5p)

Task 2: Concurrency in Scala (14 p)

One of the most important goals in the Scala project 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. (3p)

(b) Given the following code snippet: (3p)

```
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. (4p)

(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`. (4 p)

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 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

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

1.1 Implementing the TransactionQueue (5p)

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 needs to be implemented in a thread safe manner. Wrapping existing Queue-functions is encouraged.

- Define 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 functions (10p)

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 (5p)

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 withdraw more 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` type 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 function either returns nothing, indicating success, or a string describing the failure".

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

```
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)
}
```

Listing 1: Example using Either to signal errors

Project task 2: Creating the bank (15p)

In the file `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 (30p)

Back in the file `Transaction.scala` there is still work to be done. The `run` function, containing another function and a call to that function 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 handling with the `Either` datatype, you can use pattern matching – otherwise you may have to try/except the exceptions.