TDT4165 PROGRAMMING LANGUAGES

Scala Project

Autumn 2025

1 Introduction

Scala is both a functional and object-oriented programming language. This task will introduce you to Scala and familiarize you with some basic concepts through a simple project of a banking system.

Relevant readings:

- the official scala documentation, at https://docs.scala-lang.org/
- the "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**. If you use Linux, you can probably find it as a package in your distribution. Otherwise, it is available 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 the `sbt run` command in the terminal.

2 Assignment Organization and Delivery

This project consists of **two** deliveries (**Part I** and **Part II**), each counting as one individual delivery towards the 5 out of 7 deliveries necessary to get approved for the exam.

2.1 Part I (Delivery 1)

For the first delivery you should refer to tasks in Part I, that is **Task 1: Scala Introduction**, **Task 2: Higher-Order Programming in Scala**, and **Task 3: Concurrency in Scala**. This delivery mainly consists of getting used to using Scala and to shared-state concurrency, to be able to delivery Part II later.

For *Delivery 1*, please deliver a `.zip` file containing the file(s) for these two tasks and the `.pdf` file.

To get this delivery as "Approved" you need to:

- Complete Task 1, Task 2, and Task 3.

2.2 Part II (Delivery 2)

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**. For this task, you have received a partially implemented system written in Scala. Your task is to implement some missing code so that the provided tests will pass. While it is not mandatory to finish Part I in order to deliver Part II, it is highly recommended to follow the sequence, 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.

Please deliver a `.zip` file of the **entire project directory**, including 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.

To get this delivery as "Approved" you need to:

- Implement all the code marked with a "TODO" note;
- Make tests 1–10 pass;
- Make a reasonable attempt at making tests 11 and 12 pass; and
- Answer the questions in Task 3.

Part I

Introduction to Scala

Is your variable variable?

Please note that Scala has two different ways to declare variables[1].

The `var`(iable) keyword declares a *mutable* variable, similar to those found in Java. Those variables can be re-assigned multiple times (i.e., bound to new values).

The `val`(ue) keyword declares an *immutable* "variable". Those variables can be assigned only once (single-assignment), and are somehow similar to those found in Oz and most other languages tailored to functional programming. Note however that they are not dataflow variables as we are used in Oz. In Scala there is no concept of a variable being unbound, and therefore threads to not wait on unbound variables.

Similarly, you find both *mutable* and *immutable* collections in Scala[2], under the `scala.collection.mutable` and `scala.collection.immutable` packages, respectively.

As the Scala documentation mentions: *you should always create a variable with `val`, unless there's a reason you need a mutable variable.*

[1]`https://docs.scala-lang.org/scala3/book/taste-vars-data-types.html`
[2]`https://docs.scala-lang.org/overviews/collections-2.13/overview.html`

Task 1: Scala Introduction

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: Higher-Order Programming in Scala

Scala supports higher-order programming. In this task we will go back to some of the task we developed in Oz and understand how they can be implemented in Scala.

(a) Refer to Assignment 3 of this course, which is about Higher-Order Programming. Implement Task 1 and Task 4 of that assignment in Scala.

(b) Which differences did you find, if any?

Task 3: Concurrency in Scala

One of the goals in the Scala project is to understand how threads work in Scala, and which concurrency primitives are available.

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.

(b) Consider the following Scala code (you find it also in the `ConcurrencyTrobules.scala` file inside the provided .zip):

```
package example

object ConcurrencyTroubles {
  private var value1: Int = 1000
  private var value2: Int = 0
  private var sum: Int = 0

  def moveOneUnit(): Unit = {
    value1 -= 1
    value2 += 1
    if(value1 == 0) {
      value1 = 1000
      value2 = 0
    }
  }
  def updateSum(): Unit = {
    sum = value1 + value2
  }

  def execute(): Unit = {
    while(true) {
      moveOneUnit()
      updateSum()
      Thread.sleep(50)
    }
  }

  // This is the "main" method, the entry point of execution.
  // It could have been placed in a different file.
  def main(args: Array[String]): Unit = {
    for (i <- 1 to 2) {
      val thread = new Thread {
        override def run = execute()
      }
      thread.start()
    }

    while(true) {
      updateSum()
      println(sum + " [" + value1 + " " + value2 + "]")
      Thread.sleep(100)
    }
  }
}
```

Execute this code and observe it running for a few minutes. Answer the following questions:

- What is this code supposed to do?
- Is it working as expected?
- What do you think it is happening?
- Would it be possible to experience the same behavior in Oz? Why or why not?
- Give one of how this behavior could impact a real application.

Note: This code includes an infinite loop; to terminate it you need to terminate it from your IDE, or otherwise use the key combination CTRL+C (or equivalent) if you are working in a terminal.

(c) Modify the code so that it is thread-safe. You can do it in different ways in Scala, can you find at least two different ways to fix the above code?

Part II

The Banking System

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 the 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*$^{\text{TM}}$. 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.

Task 1: Preliminaries

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

1.1 Implementing the TransactionPool

In the file `Transaction.scala` you will find the definition of the `TransactionPool` class. This class needs to be implemented. The class needs a data structure to hold the transactions. There is no requirement on the kind of data structure used, as long as the defined functions are implemented correctly. Wrapping existing collection functions is encouraged.

The functions that are defined also need to be implemented in a thread-safe manner.

- Define a data structure to hold transactions.
- Implement functions of `TransactionPool` in a thread-safe manner.

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

1.2 Account Functions

In the file `Account.scala`, you will find two functions that are not implemented: `withdraw`, and `deposit`.

- `withdraw` removes an amount of money from the account.
- `deposit` inserts an amount of money to the account.

The Account class should be implemented as *immutable*, that is, the state of the object should not be modified, but instead a new object should be returned.

1.3 Eliminating Exceptions

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 make the behavior of the program less predictable; read the section the `Either` datatype in the Appendix (end of this document), and see how it can be used in place of exceptions and program crashes.

- `withdraw` should fail if we withdraw a negative amount or if we request a withdrawal that is larger than the available funds.
- `deposit` should fail if we deposit a negative amount.
- Both should return an `Either` datatype and should not throw exceptions.

Tests 1–6 (and possibly others) should pass now.

Task 2: Completing the Bank

In `Bank.scala` you will find incomplete code that needs to be completed. Further code to be completed or extended is also found in `Transaction.scala`.

- `createAccount` creates a new account and returns its code to the user. The user then interacts with the bank through that code. The account is stored in a local registry of bank accounts, implemented as a key/value map.
- `getAccount` returns the Account corresponding to a given account code. The user can use the object to inspect the balance and other properties. Data integrity is guaranteed by the fact that the object is immutable.
- `transfer` receives data for a new transfer to be made between accounts. It creates a new transaction object and places it in the `transactionQueue`.
- `processTransactions` is called when existing transactions should be processed. It processes the existing transactions concurrently, starting a new thread for each of them. After the transactions are processed, those that succeeded are moved to the the pool of processed transaction pool. Those that failed are restarted, if possible, or otherwise also moved to the pool of processed transactions.
- 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. Once a transaction is completed (either succeeded or used all the avialable attempts), then it should be moved to the `completedTransactions` pool.

Task 3: Explain how the code works

Answer to the following questions:

1. How does the code of the bank system work? Write a short summary of the main ideas of the code, according to your understanding.
2. Which features of the system were the easiest to implement and why?
3. Which features of the system were the most challenging to implement and why?

4. Write a short summary of what you have implemented to make tests pass.

A Appendix

A.1 Notes on the `Either` datatype

The `Either` type[3] is useful to represent the fact that a function may succeed or not; it consists of a `Left` and a `Right` type. For example `Either[String,Unit]` means that the type either holds nothing (`Unit`) or a `String`. We can use this as a return type of a function, to say that "The function either returns *nothing*, indicating success, or a *string* describing the failure".

Convention suggests that Left is used for failure and Right is used for success. However, choosing whether Left of Right means success can also depend on development context. For the sake of automated tests in this project, use Right to indicate success and Left to indicate failure.

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

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

[3]`https://www.scala-lang.org/api/2.13.6/scala/util/Either.html`