# Programmeringsspråk TDT4165 - Project Delivery 2

Ola Kristoffer Hoff

20. November 2022

## Project Task 1: Preliminaries

### 1.1 Implementing the TransactionQueue

I did as suggested in the task, I implemented the functions by wrapping the original queue functions in synchronisation-blocks. I also used a standard queue object as suggested.

```
8  class TransactionQueue {
9
10     // TODO
11     // project task 1.1
12     // Add datastructure to contain the transactions
13
14     private val queue = Queue[Transaction]() //A queue of transactions
15
16
17     // Remove and return the first element from the queue
18     def pop: Transaction = {
19       this.synchronized { //Ensure sync on this object
20         queue.dequeue //Return the value first in the queue
21       }
22     }
23
24     // Return whether the queue is empty
25     def isEmpty: Boolean = {
26       this.synchronized {
27         queue.isEmpty
28       }
29     }
30
31     // Add new element to the back of the queue
32     def push(t: Transaction): Unit = {
33       this.synchronized {
34         queue.enqueue(t)
35       }
36     }
37
38     // Return the first element from the queue without removing it
39     def peek: Transaction = {
40       this.synchronized {
41         queue.head
42       }
43     }
44
45     // Return an iterator to allow you to iterate over the queue
46     def iterator: Iterator[Transaction] = {
47       this.synchronized {
48         queue.iterator
49       }
50     }
51 }
```

Figure 1: The implementation of the functions in the *TransactionQueue*.

### 1.2 Account functions

I implemented the basic functionality of the functions in the account.

```scala
  def withdraw(amount: Double): Unit = {
    this.synchronized {
      balance.amount -= amount
    }
  }

  def deposit (amount: Double): Unit = {
    this.synchronized {
      balance.amount += amount
    }
  }

  def getBalanceAmount: Double = {
    this.synchronized {
      balance.amount
    }
  }
```

Figure 2: Functions in *Account.scala* implemented.

### 1.3 Eliminating exceptions

Returns an *Either* with *Right* being a string describing the error, and the *Left* being a *Unit* as before.

```scala
  def withdraw(amount: Double): Either[Unit, String] = {
    this.synchronized {

      if (amount < 0) {
        return Right("Cannot withdraw a negative amount.")
      }

      if (amount > balance.amount) {
        return Right("Cannot withdraw more than the account holds.")
      }

      balance.amount -= amount
      return Left()
    }
  }

  def deposit (amount: Double): Either[Unit, String] = {
    this.synchronized {

      if (amount < 0) {
        return Right("Cannot deposit a negative amount.")
      }

      balance.amount += amount
      return Left()
    }
  }
```

Figure 3: Code for task 1.3 that ensures safe transactions without exceptions.

# Project Task 2: Creating the bank

This is pretty straight forward, I have done exactly as the TODO's stated. This should be clear by the code and the comments in figure 4.

```
6     def addTransactionToQueue(from: Account, to: Account, amount: Double): Unit = {
7       //Create the new transaction object
8       val transaction = new Transaction(transactionsQueue,
9                                         processedTransactions,
10                                        from,
11                                        to,
12                                        amount,
13                                        allowedAttempts)
14
15      //Put the transaction object in the queue
16      transactionsQueue.push(transaction)
17
18      //Spawn a thread that calls the processedTransactions
19      val thread = new Thread {
20        override def run() {
21          processTransactions
22        }
23      }
24
25      //Start the thread
26      thread.start()
27    }
28                                                  // TODO
29                                                  // project task 2
30                                                  // create a new transaction object and
31                                                  // spawn a thread that calls processTra
32
33    private def processTransactions: Unit = {
34      //Pop the transaction from the queue
35      val transaction: Transaction = transactionsQueue.pop
36      //Spawn a thread that executes the transaction
37      val thread = new Thread {
38        override def run() {
39          transaction.run()
40        }
41      }
42
43      //Start the thread
44      thread.start()
45
46      //We sync while checking the value of the transaction status
47      if (transaction synchronized {transaction.status == TransactionStatus.PENDING}) {
48        transactionsQueue.push(transaction) //Still working, add back in queue
49        processTransactions //Call again and check the queue
50      }
51      else { //The transaction is done, either failed or succeeded
52        processedTransactions.push(transaction)
53      }
54    }
```

Figure 4: Code for implementing the functions in the *Bank.scala* file.

## Project Task 3: Actually solving the bank problem

Here is simply check if the result from the withdraw is a left value, a success, if so we try to do the deposit. If this too succeeds we set the status to success. In the other case we must reverse the withdraw if the deposit failed. After this is done, if the status is still pending it means that we did not succeed and we increment out attempt counter and if we have reached the limit, we set the status as failed.

```
63  override def run: Unit = {
64
65      def doTransaction() = {
66          // TODO - project task 3
67          // Extend this method to satisfy requirements.
68          //from withdraw amount
69          //to deposit amount
70          if (from.withdraw(amount).isLeft) { //The withdraw was successful
71            if (to.deposit(amount).isLeft) { //The deposit was successful
72              status = TransactionStatus.SUCCESS
73            }
74            else //Deposit failed, need to reverse withdraw
75            {
76              from.deposit(amount) //This should always work if we just withdrew it.
77            }
78          }
79
80          if (status == TransactionStatus.PENDING) { //If we are still pending, this means we did not succeed
81            attempt += 1 //Need to increment attempt, since it failed
82
83            if (attempt >= allowedAttemps) { //We have reached our maximum number of attempts
84              status = TransactionStatus.FAILED
85            }
86          }
87      }
88
89      // TODO - project task 3
90      // make the code below thread safe
91      if (status == TransactionStatus.PENDING) {
92        //Put the code in a sync block to sync it
93        this.synchronized {
94          doTransaction
95          Thread.sleep(50) // you might want this to make more room for
96                           // new transactions to be added to the queue
97        }
98      }
```

Figure 5: Code for implementing the *run*-function in the *Transaction* class.