TDT4165 Programming Language - Autumn 2021
Alexander Høyby
Bjørn Are Odden

# Scala Project Delivery 2

## Task 1:

Nothing to comment

## Task 2:

Nothing to comment

## Task 3:

Nothing to comment

# Code

## Account

```
import exceptions._

class Account(val bank: Bank, initialBalance: Double) {

    class Balance(var amount: Double) {}

    val balance = new Balance(initialBalance)

    // TODO
    // for project task 1.2: implement functions
    // for project task 1.3: change return type and update function bodies
    def withdraw(amount: Double): Either[Unit, String] = this.synchronized {
      if (amount > balance.amount) {
        Right("Cannot withdraw more than current balance.")
      }
      else if (amount <= 0) {
        Right("Cannot withdraw 0 or negative amount.")
      }
      else {
        this.balance.amount -= amount
        Left()
      }
    }
    def deposit (amount: Double): Either[Unit, String] = this.synchronized {
      if (amount <= 0) {
        Right("Cannot deposit 0 or negative amount")
      }
      else {
        this.balance.amount += amount
        Left()
      }
    }
    def getBalanceAmount: Double = this.synchronized {
      this.balance.amount
    }

    def transferTo(account: Account, amount: Double): Unit = {
        bank addTransactionToQueue (this, account, amount)
    }
}
```

## Bank

```scala
import scala.annotation.tailrec

class Bank(val allowedAttempts: Integer = 3) {

    private val transactionsQueue: TransactionQueue = new TransactionQueue()
    private val processedTransactions: TransactionQueue = new TransactionQueue()

    def addTransactionToQueue(from: Account, to: Account, amount: Double): Unit = {

      // Create a new transaction object
      // Put transaction object in the queue
      transactionsQueue.push(new Transaction(
        transactionsQueue,
        processedTransactions,
        from,
        to,
        amount,
        allowedAttempts))

      // spawn a thread that calls processTransactions
      new Thread(() => processTransactions()).start()

    }

    @tailrec
    private def processTransactions(): Unit = {

      // Pop a transaction from the queue
      val transaction = transactionsQueue.pop

      // Spawns a thread to execute the transaction.
      val transactionThread = new Thread(transaction)
      transactionThread.start()
      transactionThread.join()

      // Finally do the appropriate thing, depending on whether
      // the transaction succeeded or not

      if (transaction.status == TransactionStatus.PENDING) {
        transactionsQueue.push(transaction)
        processTransactions() // retry
      } else { // success or failed
        //processedTransactions.push(transaction)
        processedTransactions.push(transaction)
      }
    }


    def addAccount(initialBalance: Double): Account = {
        new Account(this, initialBalance)
    }

    def getProcessedTransactionsAsList: List[Transaction] = {
        processedTransactions.iterator.toList
    }
}
```

## Transaction

```scala
import exceptions._

import scala.collection.mutable

object TransactionStatus extends Enumeration {
  val SUCCESS, PENDING, FAILED = Value
}

class TransactionQueue {

    // project task 1.1
    // Add datastructure to contain the transactions
    val queue: mutable.Queue[Transaction] = mutable.Queue[Transaction]()

    // Remove and return the first element from the queue
    def pop: Transaction = this synchronized queue.dequeue

    // Return whether the queue is empty
    def isEmpty: Boolean = this synchronized queue.isEmpty

    // Add new element to the back of the queue
    def push(t: Transaction): Unit = this synchronized queue.enqueue(t)

    // Return the first element from the queue without removing it
    def peek: Transaction = this synchronized queue.front

    // Return an iterator to allow you to iterate over the queue
    def iterator: Iterator[Transaction] = this synchronized queue.iterator
}

class Transaction(val transactionsQueue: TransactionQueue,
                  val processedTransactions: TransactionQueue,
                  val from: Account,
                  val to: Account,
                  val amount: Double,
                  val allowedAttempts: Int) extends Runnable {

  var status: TransactionStatus.Value = TransactionStatus.PENDING
  var attempt = 0

  override def run: Unit = {

      def doTransaction(): Unit = {
        attempt += 1
        val withdraw : Either[Unit, String] = from.withdraw(amount)
        if (withdraw.isLeft){
          to.deposit(amount)
```

```
          status = TransactionStatus.SUCCESS
        }
        else if (attempt >= allowedAttempts) {
          status = TransactionStatus.FAILED
        }
      }

      // TODO - project task 3
      // make the code below thread safe

      if (status == TransactionStatus.PENDING) {
        doTransaction()
        Thread.sleep(50) // you might want this to make more room for
        // new transactions to be added to the queue
      }
    }
}
```

Main

```
object Main extends App {

    def thread(body: => Unit): Thread = {
        val t = new Thread {
            override def run() = body
        }
        t.start
        t
    }
}
```