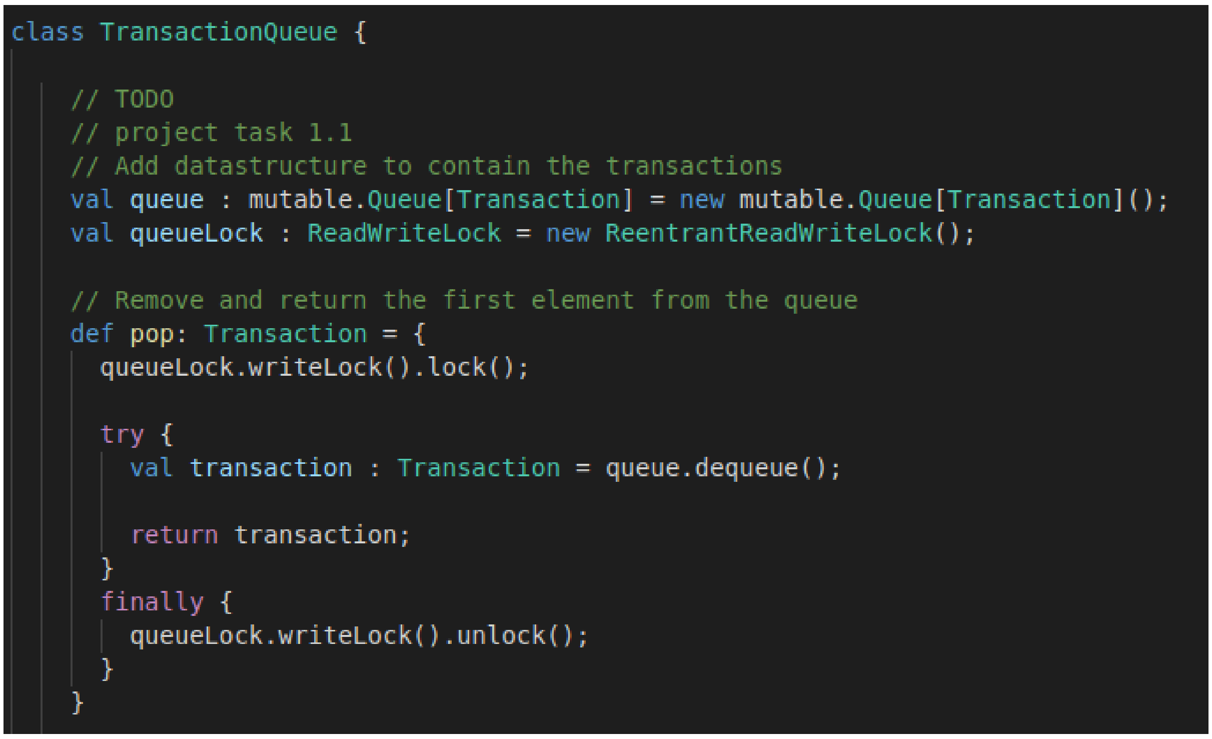
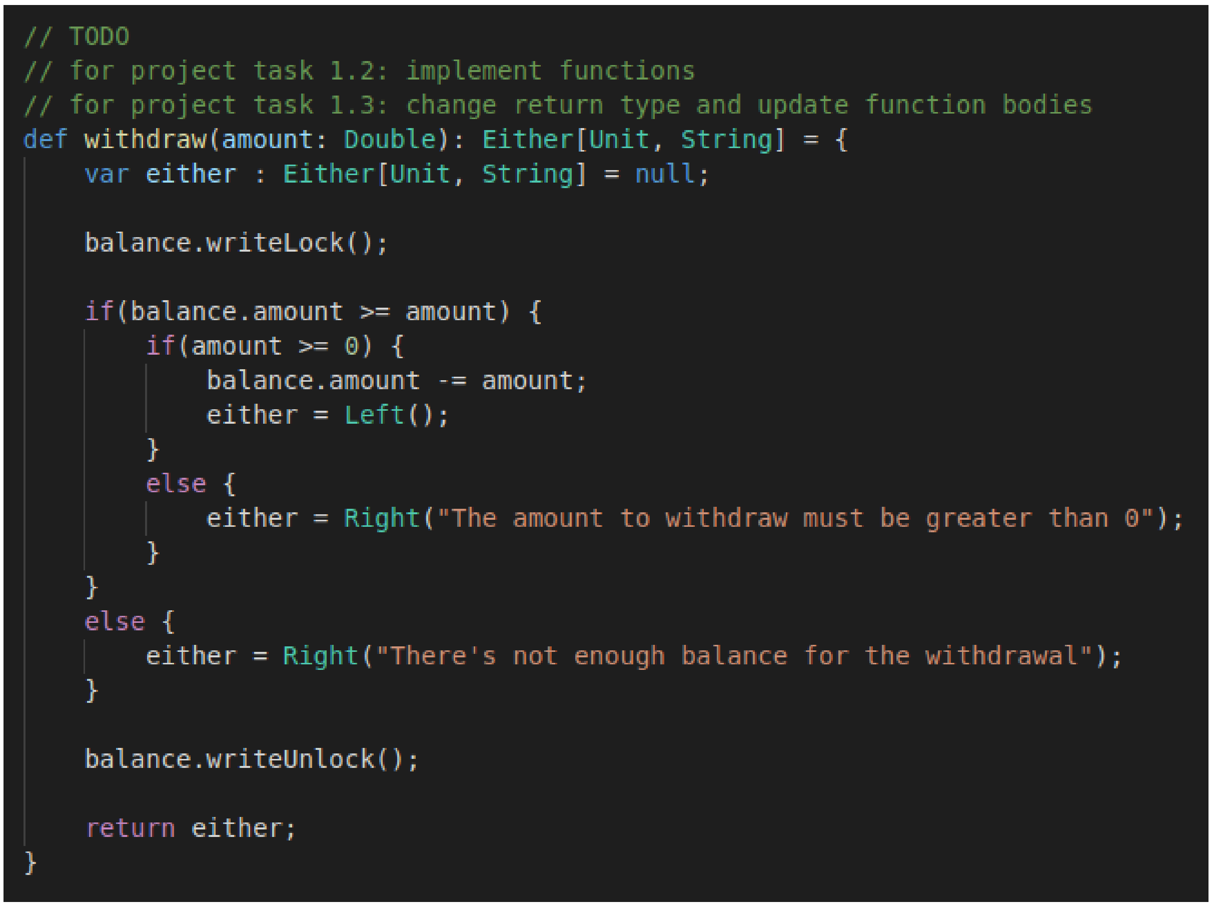
Task 1.1

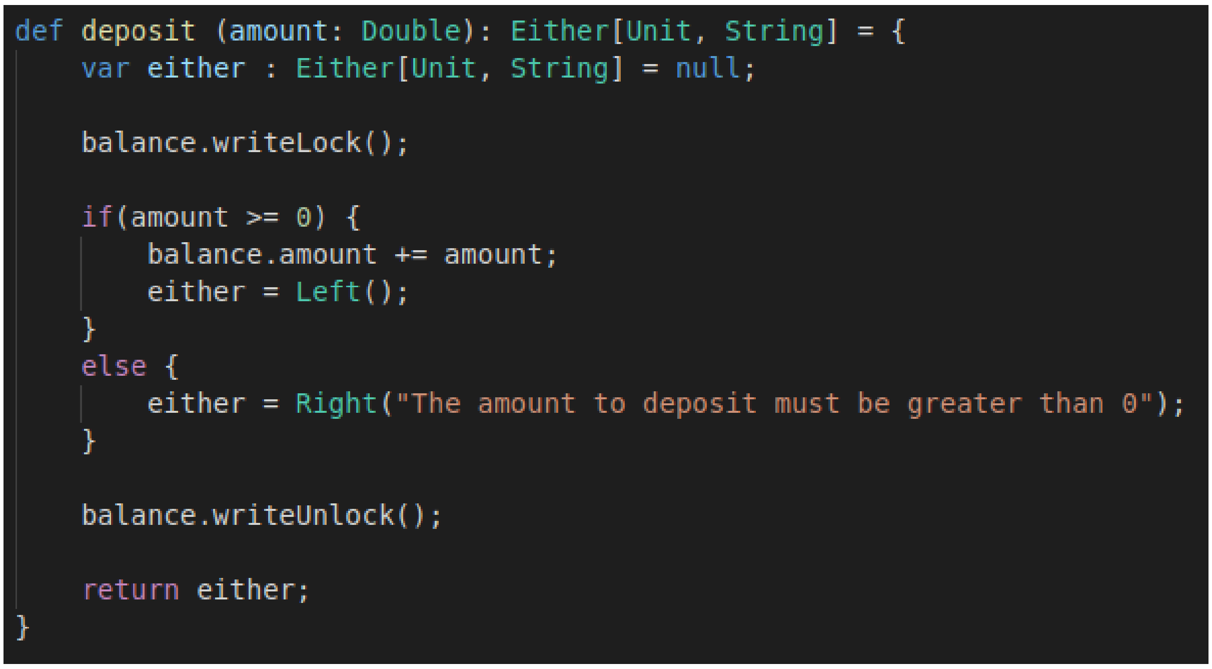


```
class TransactionQueue {

    // TODO
    // project task 1.1
    // Add datastructure to contain the transactions
    val queue : mutable.Queue[Transaction] = new mutable.Queue[Transaction]();
    val queueLock : ReadWriteLock = new ReentrantReadWriteLock();

    // Remove and return the first element from the queue
    def pop: Transaction = {
      queueLock.writeLock().lock();

      try {
        val transaction : Transaction = queue.dequeue();

        return transaction;
      }
      finally {
        queueLock.writeLock().unlock();
      }
    }
```

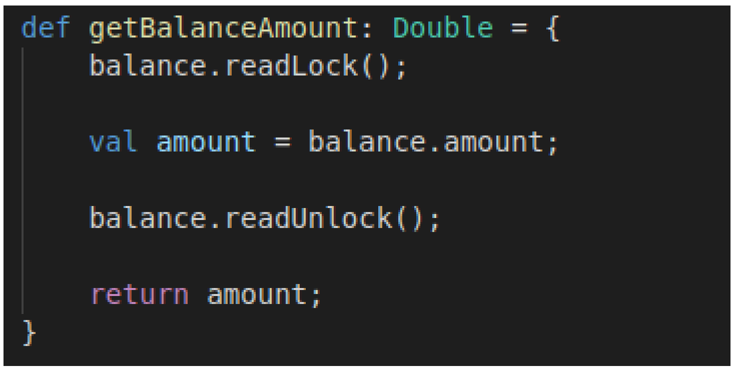
Task 1.2.



```
// 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] = {
    var either : Either[Unit, String] = null;

    balance.writeLock();

    if(balance.amount >= amount) {
        if(amount >= 0) {
            balance.amount -= amount;
            either = Left();
        }
        else {
            either = Right("The amount to withdraw must be greater than 0");
        }
    }
    else {
        either = Right("There's not enough balance for the withdrawal");
    }

    balance.writeUnlock();

    return either;
}
```



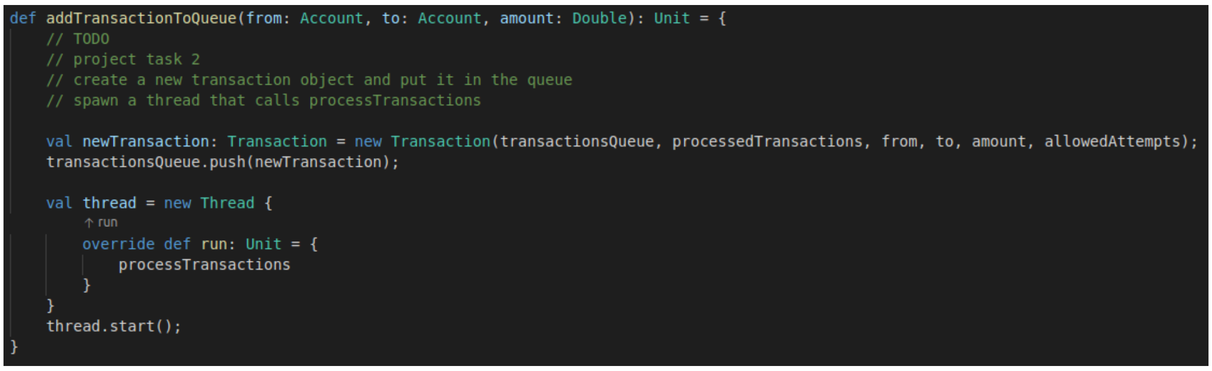
```scala
def deposit (amount: Double): Either[Unit, String] = {
    var either : Either[Unit, String] = null;

    balance.writeLock();

    if(amount >= 0) {
        balance.amount += amount;
        either = Left();
    }
    else {
        either = Right("The amount to deposit must be greater than 0");
    }

    balance.writeUnlock();

    return either;
}
```



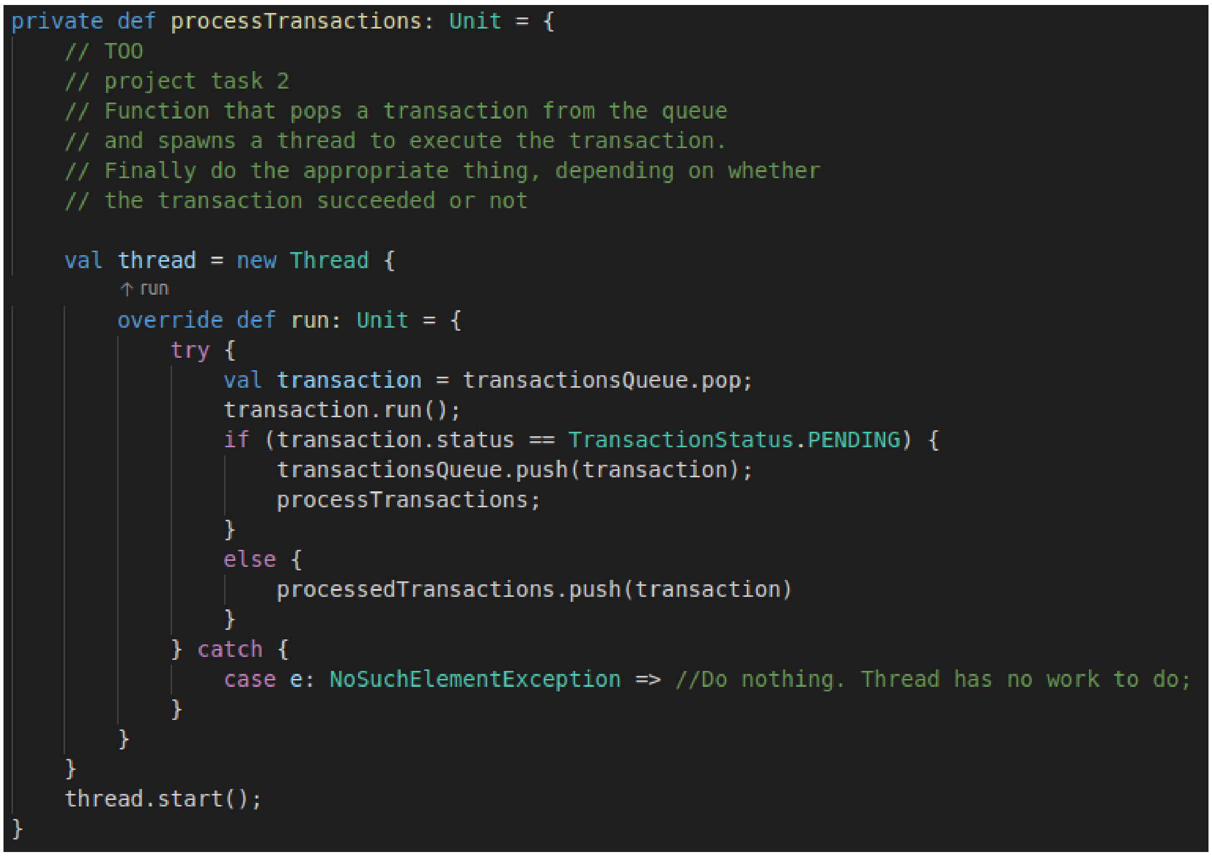
```scala
def getBalanceAmount: Double = {
    balance.readLock();

    val amount = balance.amount;

    balance.readUnlock();

    return amount;
}
```

Task 1.3.
Functions `withdraw` and `deposit` have been made thread safe and the errors are now handled with the `Either` datatype.

Task 2

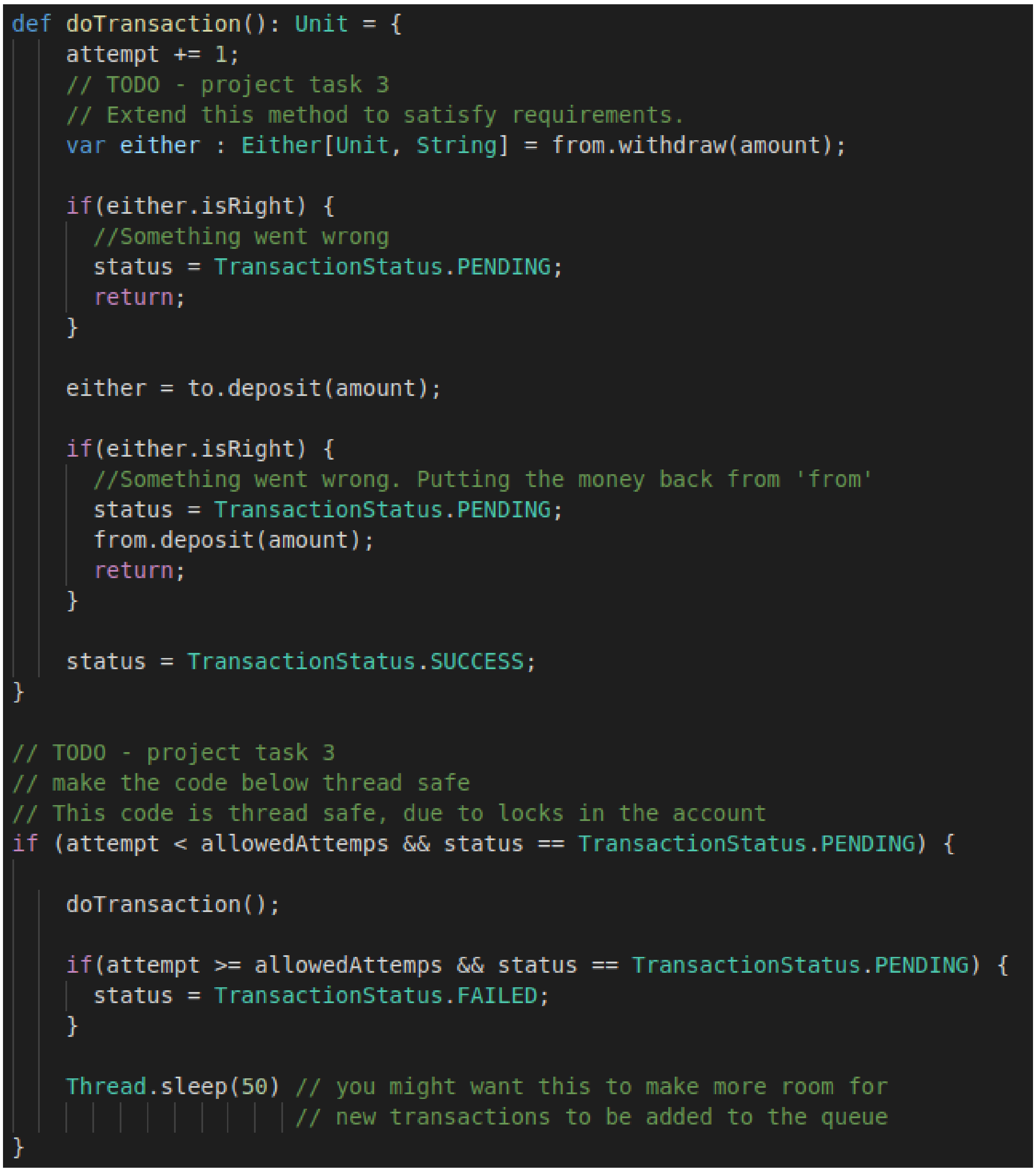


```scala
def addTransactionToQueue(from: Account, to: Account, amount: Double): Unit = {
    // TODO
    // project task 2
    // create a new transaction object and put it in the queue
    // spawn a thread that calls processTransactions

    val newTransaction: Transaction = new Transaction(transactionsQueue, processedTransactions, from, to, amount, allowedAttempts);
    transactionsQueue.push(newTransaction);

    val thread = new Thread {
        override def run: Unit = {
            processTransactions
        }
    }
    thread.start();
}
```



```scala
private def processTransactions: Unit = {
    // TOO
    // project task 2
    // Function that pops a transaction from the queue
    // and spawns a thread to execute the transaction.
    // Finally do the appropriate thing, depending on whether
    // the transaction succeeded or not

    val thread = new Thread {
        ↑ run
        override def run: Unit = {
            try {
                val transaction = transactionsQueue.pop;
                transaction.run();
                if (transaction.status == TransactionStatus.PENDING) {
                    transactionsQueue.push(transaction);
                    processTransactions;
                }
                else {
                    processedTransactions.push(transaction)
                }
            } catch {
                case e: NoSuchElementException => //Do nothing. Thread has no work to do;
            }
        }
    }
    thread.start();
}
```

Task 3



```scala
def doTransaction(): Unit = {
    attempt += 1;
    // TODO - project task 3
    // Extend this method to satisfy requirements.
    var either : Either[Unit, String] = from.withdraw(amount);

    if(either.isRight) {
      //Something went wrong
      status = TransactionStatus.PENDING;
      return;
    }

    either = to.deposit(amount);

    if(either.isRight) {
      //Something went wrong. Putting the money back from 'from'
      status = TransactionStatus.PENDING;
      from.deposit(amount);
      return;
    }

    status = TransactionStatus.SUCCESS;
}

// TODO - project task 3
// make the code below thread safe
// This code is thread safe, due to locks in the account
if (attempt < allowedAttemps && status == TransactionStatus.PENDING) {

    doTransaction();

    if(attempt >= allowedAttemps && status == TransactionStatus.PENDING) {
      status = TransactionStatus.FAILED;
    }

    Thread.sleep(50) // you might want this to make more room for
                     // new transactions to be added to the queue
}
```