Task 1

a

```
def numberArrayGenerator(): Array[Int] = {
    var result: Array[Int] = new Array[Int](50)

    for(i <- 1 to 50) {
        result(i - 1) = i
    }

    return result
}
```

b

```
def sumElementsInArray(array: Array[Int]): Int = {
    var sum: Int = 0

    for (element <- array) {
        sum += element
    }
    return sum
}
```

c

```
def sumElementsRecursively(array: Array[Int]): Int = {
    array match {
        case Array() => {
            0
```

```
        }

        case Array(_, _*) => {
            return array.head + sumElementsRecursively(array.tail)
        }
    }
}
```

d

```
def nthFibonacci(n: BigInt): BigInt = {
    if (n <= 1) {
        return n
    } else {
        return nthFibonacci(n - 1) + nthFibonacci(n - 2)
    }
}
```

BigInt supports much larger numbers than Int, which is necessary when computing large fibonacci numbers. For instance, fibonacci of 250 would equal 7896325826131730509282738943634332893686268675876375, while the limit for Int is 2147483647

Task 2

a

```
def createThreadWithFunction(callback: () => Unit): Thread = {
  val thread = new Thread(
      new Runnable {
          def run() {
```

```
                callback()
            }
        }
    )

    return thread

}
```

b

```
private var counter: Int = 0

def increaseCounter(): Unit = {
    /this/.synchronized {
        counter += 1
    }
}

def printElement(): Unit = {
    println(counter)
}

def main(args: Array[String]) {
    createThreadWithFunction(increaseCounter).start
    createThreadWithFunction(increaseCounter).start
    createThreadWithFunction(printElement).start
}
```

If two threads tries to read or write to the same variable, these can overlap in execution and the output can differ. This phenomenon is called race condition.

If two people tries to withdraw from the same account at the same time, and the

withdrawals are executed using separate threads, the result can differ because of overlapping execution.

c

```
private var counter: Int = 0
def increaseCounter(): Unit = {
    counter.synchronized {
        counter += 1
    }
}

def printElement(): Unit = {
    println(counter)
}

def main(args: Array[String]) {
    createThreadWithFunction(increaseCounter).start
    createThreadWithFunction(increaseCounter).start
    createThreadWithFunction(printElement).start
}
```

d

A deadlock can occur when two threads tries to access resources that are locked by the other party.

Take for instance two chefs preparing the same meal. One chef needs the knife which is currently in use by the other chef. Simultaneously the other chef needs the cutting board which also is in use. Both chefs refuse to let go of their resource, before they can get the one they need, and the chefs find themselves in a deadlock.

This problem can be avoided if one makes sure that the resources are always accessed in the same order. If both chefs tries to pick up the cutting board first, one of them will have it first, and then the other will wait for it to be available. The chef would not pick up the knife

instead, as the routine order prevents them from it.

```
object A {
    lazy val one = B.two
}

object B {
    lazy val one = A.one
    lazy val two = 30
}
```