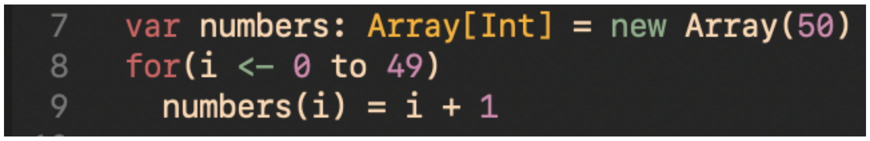
Programmeringsspråk TDT4165 - Project Delivery 1

Ola Kristoffer Hoff

1. November 2022

1 Scala Introduction

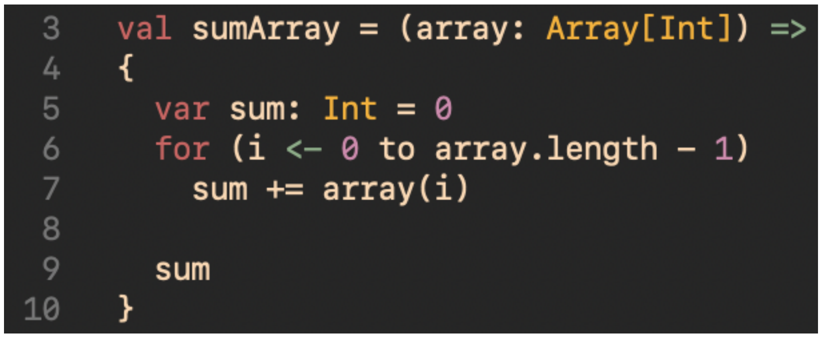
a Generate an array containing the values 1 up to (and including) 50 using a for loop.



```scala
var numbers: Array[Int] = new Array(50)
for(i <- 0 to 49)
  numbers(i) = i + 1
```

Figure 1: Code to generate an array with the number s one to 50, inclusive.

b Create a function that sums the elements in an array of integers using a for loop.

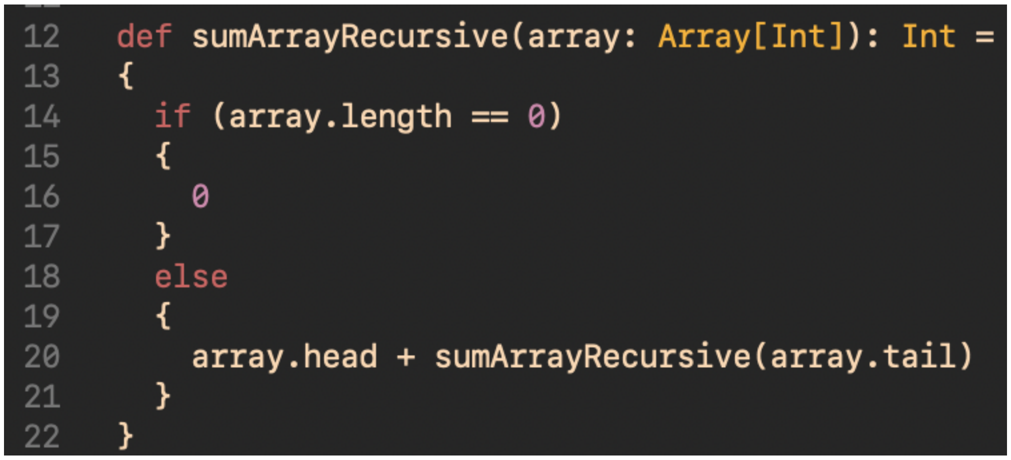


```scala
val sumArray = (array: Array[Int]) =>
{
  var sum: Int = 0
  for (i <- 0 to array.length - 1)
    sum += array(i)

  sum
}
```

Figure 2: Code for function summing an array of integers in a for-loop.

c Create a function that sums the elements in an array of integers using recursion.

The solution here is a method, but methods are just functions bound to objects, so it is still a function.

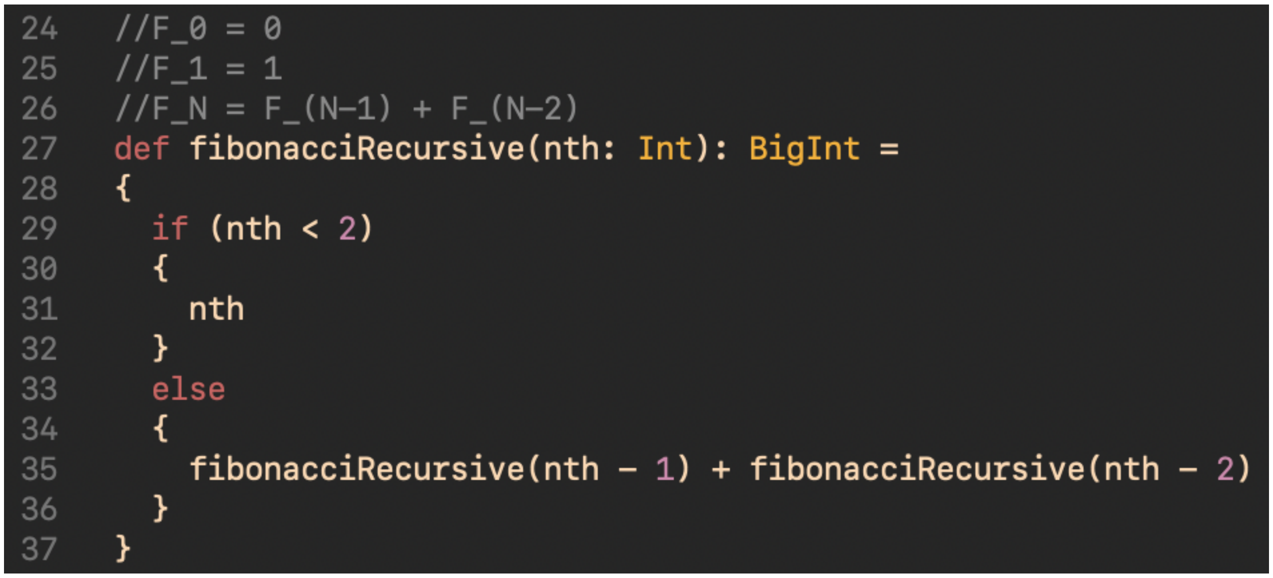


```scala
def sumArrayRecursive(array: Array[Int]): Int =
{
  if (array.length == 0)
  {
    0
  }
  else
  {
    array.head + sumArrayRecursive(array.tail)
  }
}
```

Figure 3: Code for recursive summation of integer array.

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?

The difference between *Int* and *BigInt* in Scala is that *Int* is a primative datatype and *BigInt* is an object. In other words, *Int* has a 32-bit representation, so it can be subject to overflows an other numerical difficulties(i.e. swamping). *BigInt* however is an object which (within reason) can get as big as you need it, it simply expands its internal data size to be able to represent bigger numbers.

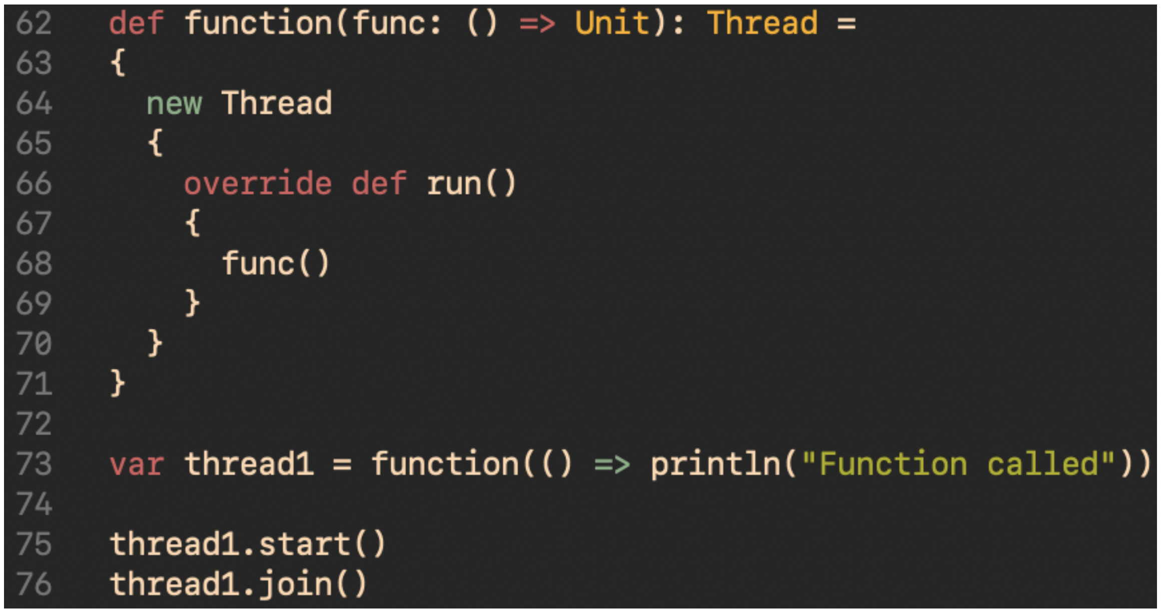


```scala
//F_0 = 0
//F_1 = 1
//F_N = F_(N-1) + F_(N-2)
def fibonacciRecursive(nth: Int): BigInt =
{
  if (nth < 2)
  {
    nth
  }
  else
  {
    fibonacciRecursive(nth - 1) + fibonacciRecursive(nth - 2)
  }
}
```

Figure 4: Code for finding n'th Fibonacci number with recursion.

2 Conncurrency in Scala

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.



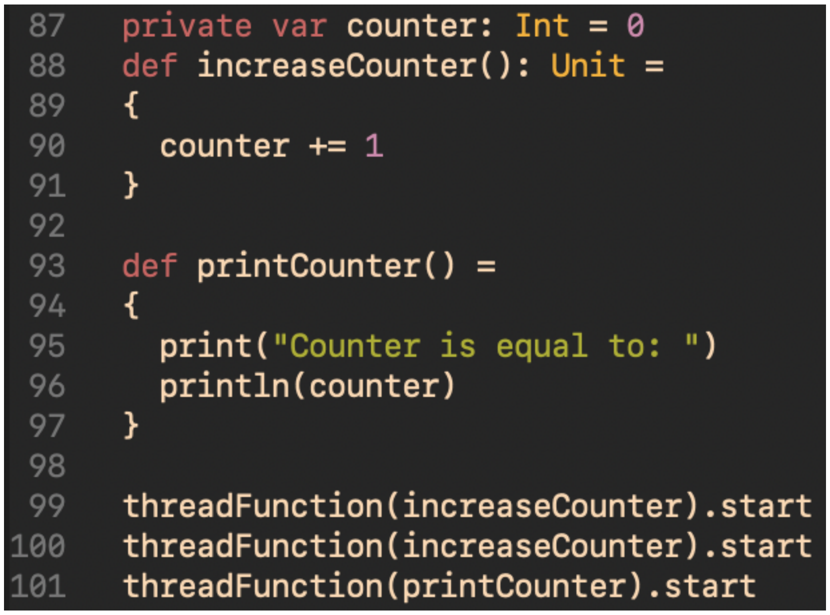
```scala
def function(func: () => Unit): Thread =
{
  new Thread
  {
    override def run()
    {
      func()
    }
  }
}

var thread1 = function(() => println("Function called"))

thread1.start()
thread1.join()
```

Figure 5: Code for function taking function as a parameter and returning a thread calling the parameter function.

b Given the following code snippet. Create a function that prints the current counter variable. Start three threads, two that initialize increaseCounter and one that initialize the print function. Run your program a few times and notice the print output. What is this phenomenon called? Give one example of a situation where it can be problematic.

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

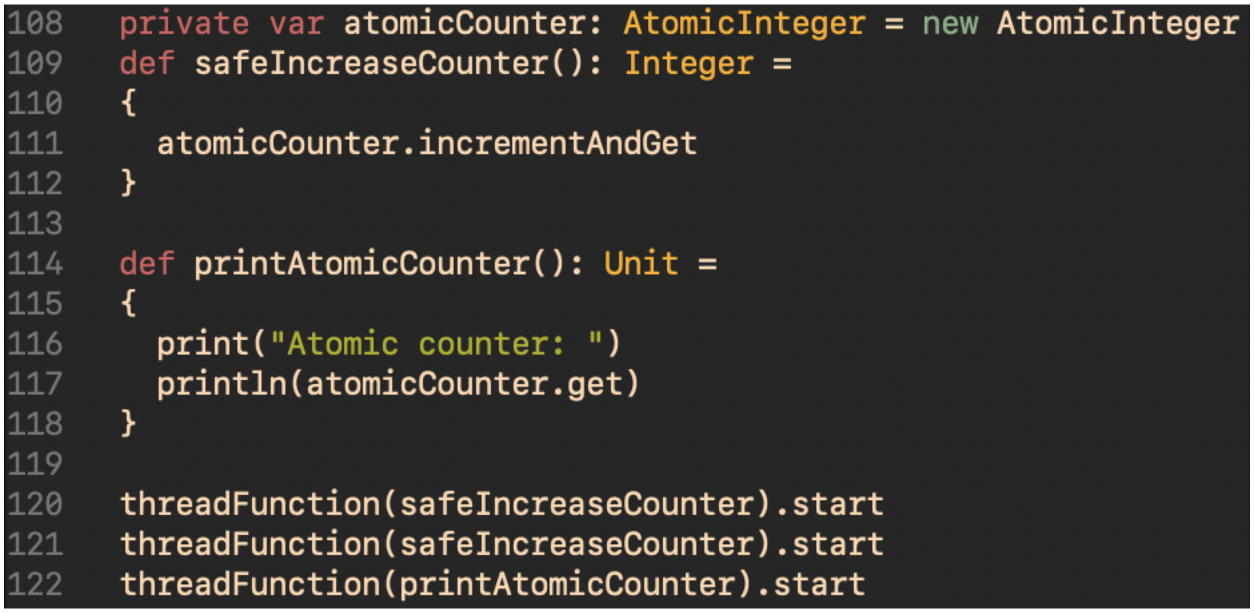
I ran it probably closer to a hundred times, and the only answer I got was: "2". But i suspect that you were fishing for the phenomenon called: *race condtion*. Which is where we have not set up any for op barrier or insurence that the threads will finish in a particular order, and thus they can finish in any order. An example of why this is bad is as follows: imagine that a multiple threads are running and responding to requests from users. If one thread needs to read some data at the same place as another one that is writing data there, that is not good, this is undefined behaviour.



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

def printCounter() =
{
  print("Counter is equal to: ")
  println(counter)
}

threadFunction(increaseCounter).start
threadFunction(increaseCounter).start
threadFunction(printCounter).start
```

Figure 6: Code for function that prints the *counter*-variable in a thread, after two *increaseCounter*-calls.

c Change increaseCounter so that it is thread-safe. Hint: atomicity.

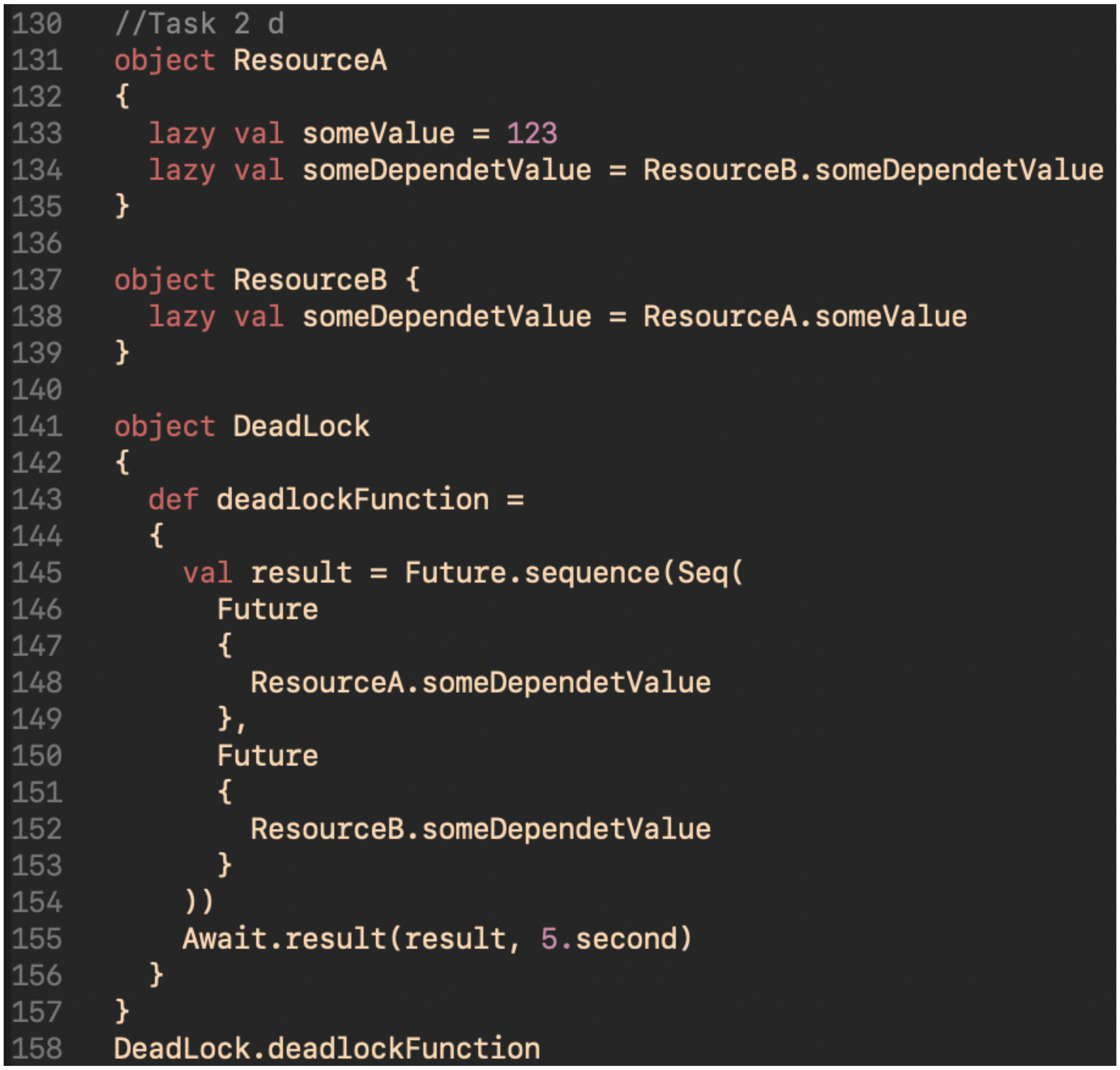


```scala
private var atomicCounter: AtomicInteger = new AtomicInteger
def safeIncreaseCounter(): Integer =
{
  atomicCounter.incrementAndGet
}

def printAtomicCounter(): Unit =
{
  print("Atomic counter: ")
  println(atomicCounter.get)
}

threadFunction(safeIncreaseCounter).start
threadFunction(safeIncreaseCounter).start
threadFunction(printAtomicCounter).start
```

Figure 7: Code for safe execution of last task, figure 6, no race conditions.

d One problem you will often meet in concurrency programming is deadlock. What is deadlock, and what can be done to prevent it? Write in Scala an example of a deadlock using lazy val.

A deadlock occurs when we are using locks to thread safe variables. In some situations it might happen that one thread has lock A and waits on lock B, but thread two has lock B and waits on lock A. They are stuck in an infinite lock wait, they in a deadlock. To prevent a deadlock one cannot have circular dependencies. This means that there cannot be a place in code that requires lock A then B (whilst still having A) and a place where the opposite occurs. This can be visualised as a graph, think of the locks as vertices, if one lock A must be acquired to get lock B we can have a directed edge from B to A to symbolise a dependence. If there exists any circular path in the graph (starting in one node, then traversing the graph and ending up in the same node) it is possible to deadlock the program. What I have described above is the essence of a deadlock, but there are four conditions that must be met for a deadlock to occur. These are: circular wait (as described above), hold and wait, mutual exclusion and no preemption. I will quickly describe the last three: Hold and wait: a thread is holding at least one resource (i.e. lock to variable) and waiting to acquire another. Mutual exclusion: only a single thread can use a given resource at a time. No preemption: a resource is given up, freed, by the thread owning it voluntarily (no one can force it to be released, to i.e. escape a deadlock). All these four conditions must be met for a deadlock to occur. So only one of them has to be designed not true to ensure no deadlocks.



```scala
//Task 2 d
object ResourceA
{
  lazy val someValue = 123
  lazy val someDependetValue = ResourceB.someDependetValue
}

object ResourceB {
  lazy val someDependetValue = ResourceA.someValue
}

object DeadLock
{
  def deadlockFunction =
  {
    val result = Future.sequence(Seq(
      Future
      {
        ResourceA.someDependetValue
      },
      Future
      {
        ResourceB.someDependetValue
      }
    ))
    Await.result(result, 5.second)
  }
}
DeadLock.deadlockFunction
```

Figure 8: Code for deadlock example. Inspired by: https://www.baeldung.com/scala/lazy-val