TDT4165 Programming Languages: Scala Project #1

Zehra Saleh, Yang Ke & Felix R. Bottolfsen

November 7, 2022

Task 1: Scala Introduction

(a)

```scala
// Task 1A
var array: Array[Int] = Array()
for (i <- 1 to 50) array :+= i
```

Figure 1: Screenshot of task 1 (a) code. Produces an array that consists of the numbers from 1 to 50, including 50

(b)

```scala
// Task 1B: Sum of elements in array using for-loop
def sum_using_for(array: Array[Int]): Int = {
  var sum: Int = 0
  for (i <- array) sum += i
  sum
}
```

Figure 2: Screenshot of task 1 (b) code. The function takes in an array of integers and returns the sum of the elements of this array with the use of a for loop

(c)

```scala
// Task 1C: Sum of elements in array using recursion
def sum_using_recursion(array: Array[Int]): Int = {
  if (array.isEmpty) 0
  else array.head + sum_using_recursion(array.tail)
}
```

Figure 3: Screenshot of task 1 (c) code. The function takes in an array of integers and returns the sum of the elements of this array by using recursion

(d)

Task 1(d): What is the difference between Int and BigInt?
Answer: Int is 32-bit long while BigInt is 64-bit long. Therefore BigInt is used when considering integer values that may exceed the supported range by Int.

```scala
// TASK 1D: n-th Fibonacci number using recursion; where F(0) = 0 and F(1) = 1
def nth_fibonacci_using_recursion(n: BigInt): BigInt = {
  if (n == 0 || n == 1) n
  else nth_fibonacci_using_recursion(n-1) + nth_fibonacci_using_recursion(n-2)
}
```

Figure 4: Screenshot of task 1 (d) code. The function takes in a number corresponding to the position in the Fibonacci series, and returns the value at this position

```scala
// TESTING:
println("TASK 1A - The generated array: ")
println("[" + array.mkString(" ") + "]")
println("TASK 1B - For Loop: " + sum_using_for(array))
println("Task 1C - Recursion: " + sum_using_recursion(array))
println("TASK 1D - n-th Fibonacci number:")
println("The 10-th Fibonacci number is " + nth_fibonacci_using_recursion(10))
```

Figure 5: The code used to test tasks 1 (a) - 1 (d). Done to check correct behaviour from the functions created

```
TASK 1A - The generated array:
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50]
TASK 1B - For Loop: 1275
Task 1C - Recursion: 1275
TASK 1D - n-th Fibonacci number:
The 10-th Fibonacci number is 55
```

Figure 6: The generated console output to tasks 1 (a) - 1 (d)

Task 2: Concurrency in Scala

(a)

```
def initializeThread(callback: () => Unit): Thread = {
  return new Thread(new Runnable {
    ↑ run
    def run() = {
      callback()
    }
  })
}
```

Figure 7: Screenshot of task 2 (a) code

(b)

```
// Task2 (b)(c)
class task2BC extends Thread {

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

  // function for threading
  def thread(body: => Unit): Thread = {
    val t = new Thread {
      ↑ run
      override def run() = body
    }
    t.start
    t
  }
  // 3 threads
  val t1 = thread{increaseCounter()}
  val t2 = thread{increaseCounter()}
  val t3 = thread{printCounter()}
}
```

Figure 8: Screenshot of task 2 (b) code

This phenomenon is called "race condition". Race condition can be problematic when generating unique identifier for each thread. The value of the identifier could be duplicated because the threads can read the counter of the same value at the same time.

(c) The function increaseCounter() is changed to be atomic. See figure 9.

(d) A deadlock is when two or more processes are blocked forever since they are all waiting on each other. There are four necessary conditions for a deadlock: mutual exclusion, hold and wait, no preemption and circular

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

Figure 9: Screenshot of task 2 (c) code

wait. By eliminating one of the mentioned conditions, the deadlock will be eliminated

The screenshot of task2d is an example code of how a deadlock can occur using lazy val. Keep in mind that the deadlock will sometimes occur in scala version 2, not 3. In version 3, scala improved the initialisation of lazy val to reduce the risk of deadlocks. The problem in the code below is that Future initialises object A, and this very instance tries to initialise object B internally. Future also tries to initialise object B which can potentially lead to a deadlock in version 2. According to the scala docs, deadlocks can still occur in version 3 by using recursive lazy vals

```scala
object A {
  lazy val initState = 25
  lazy val start = B.initState
}

You, 2 hours ago | 1 author (You)
object B {
  lazy val initState = A.initState
}

def runPotentialDeadlock() = {
  val result = Future.sequence(
    Seq(
      Future {
        A.start
      },
      Future {
        B.initState
      }
    )
  )
  Await.result(result, 20.second)
}

runPotentialDeadlock()
```

Figure 10: Screenshot of task 2 (d) example deadlock