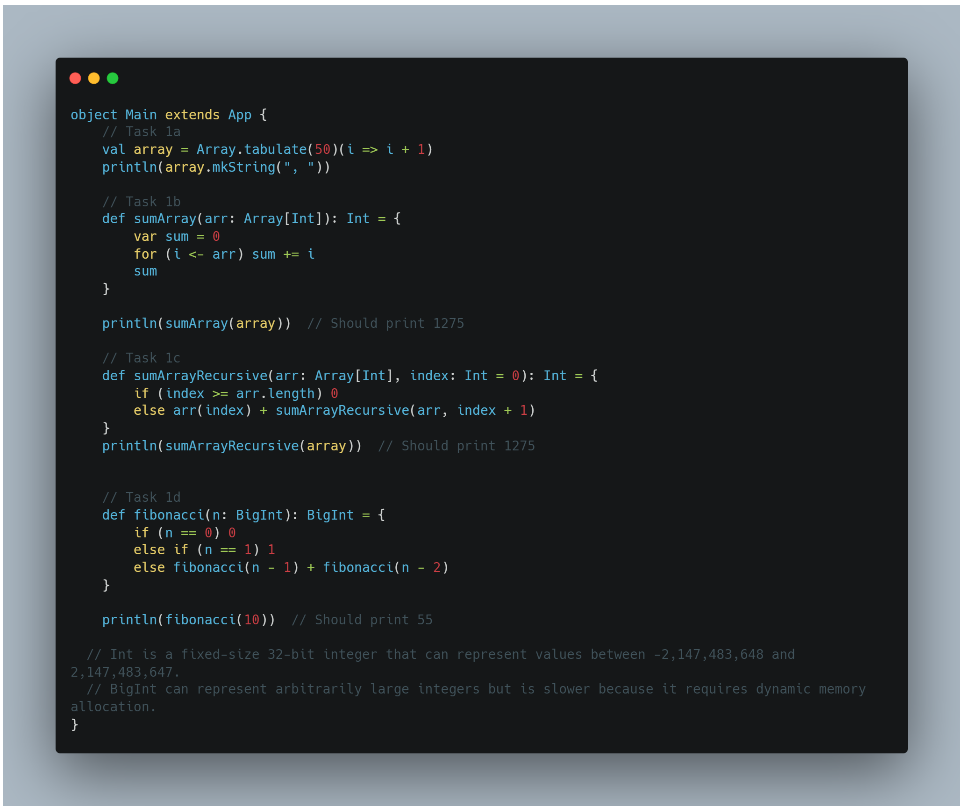
TDT4165 Programming Languages Assignment 4

Martin Hegnum Johannessen

October 8, 2024

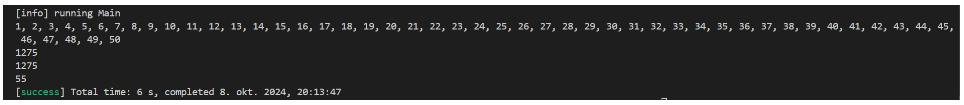
Task: 1 Scala Introduction

The goal of this task is to become familiar with Scala's syntax and core concepts, such as loops, recursion, and handling large numbers using Scala's BigInt. Sub-tasks A, B, C and D has been solved. The code implementation is presented in Figure 1 and the solution is presented in Figure 2.



```scala
object Main extends App {
    // Task 1a
    val array = Array.tabulate(50)(i => i + 1)
    println(array.mkString(", "))

    // Task 1b
    def sumArray(arr: Array[Int]): Int = {
        var sum = 0
        for (i <- arr) sum += i
        sum
    }

    println(sumArray(array))  // Should print 1275

    // Task 1c
    def sumArrayRecursive(arr: Array[Int], index: Int = 0): Int = {
        if (index >= arr.length) 0
        else arr(index) + sumArrayRecursive(arr, index + 1)
    }
    println(sumArrayRecursive(array))  // Should print 1275


    // Task 1d
    def fibonacci(n: BigInt): BigInt = {
        if (n == 0) 0
        else if (n == 1) 1
        else fibonacci(n - 1) + fibonacci(n - 2)
    }

    println(fibonacci(10))  // Should print 55

  // Int is a fixed-size 32-bit integer that can represent values between -2,147,483,648 and
2,147,483,647.
  // BigInt can represent arbitrarily large integers but is slower because it requires dynamic memory
allocation.
}
```

Figure 1: Task 1 - Code implementation

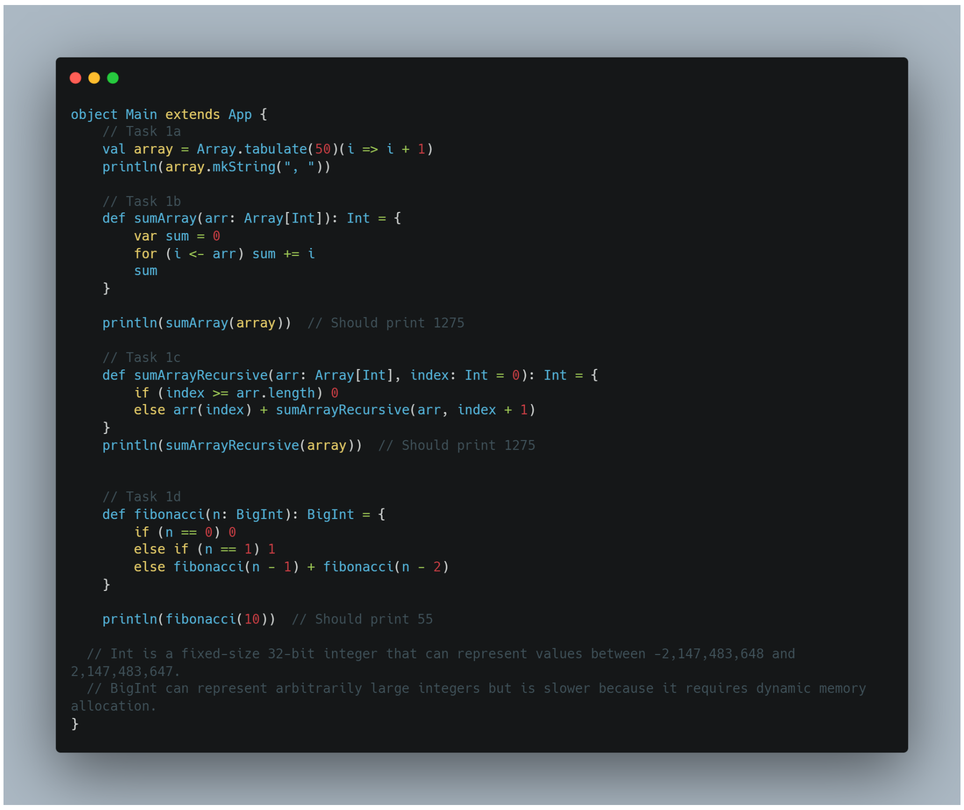


```
[info] running Main
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
 46, 47, 48, 49, 50
1275
1275
55
[success] Total time: 6 s, completed 8. okt. 2024, 20:13:47
```

Figure 2: Task 1 - Compiled code

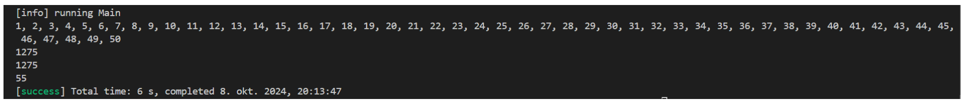
Task: 2 Higher-Order Programming in Scala

The goal of this task is to convert two tasks from Oz into Scala. These tasks focus on functional programming concepts, including recursion and higher-order functions. Below is a illustration of the Scala implementation in Figure 3 and the compiled code in Figure 4.



```scala
object Main extends App {
    // Task 1a
    val array = Array.tabulate(50)(i => i + 1)
    println(array.mkString(", "))

    // Task 1b
    def sumArray(arr: Array[Int]): Int = {
        var sum = 0
        for (i <- arr) sum += i
        sum
    }

    println(sumArray(array))  // Should print 1275

    // Task 1c
    def sumArrayRecursive(arr: Array[Int], index: Int = 0): Int = {
        if (index >= arr.length) 0
        else arr(index) + sumArrayRecursive(arr, index + 1)
    }
    println(sumArrayRecursive(array))  // Should print 1275


    // Task 1d
    def fibonacci(n: BigInt): BigInt = {
        if (n == 0) 0
        else if (n == 1) 1
        else fibonacci(n - 1) + fibonacci(n - 2)
    }

    println(fibonacci(10))  // Should print 55

  // Int is a fixed-size 32-bit integer that can represent values between -2,147,483,648 and
2,147,483,647.
  // BigInt can represent arbitrarily large integers but is slower because it requires dynamic memory
allocation.
}
```

Figure 3: Task 2 - Code implementation

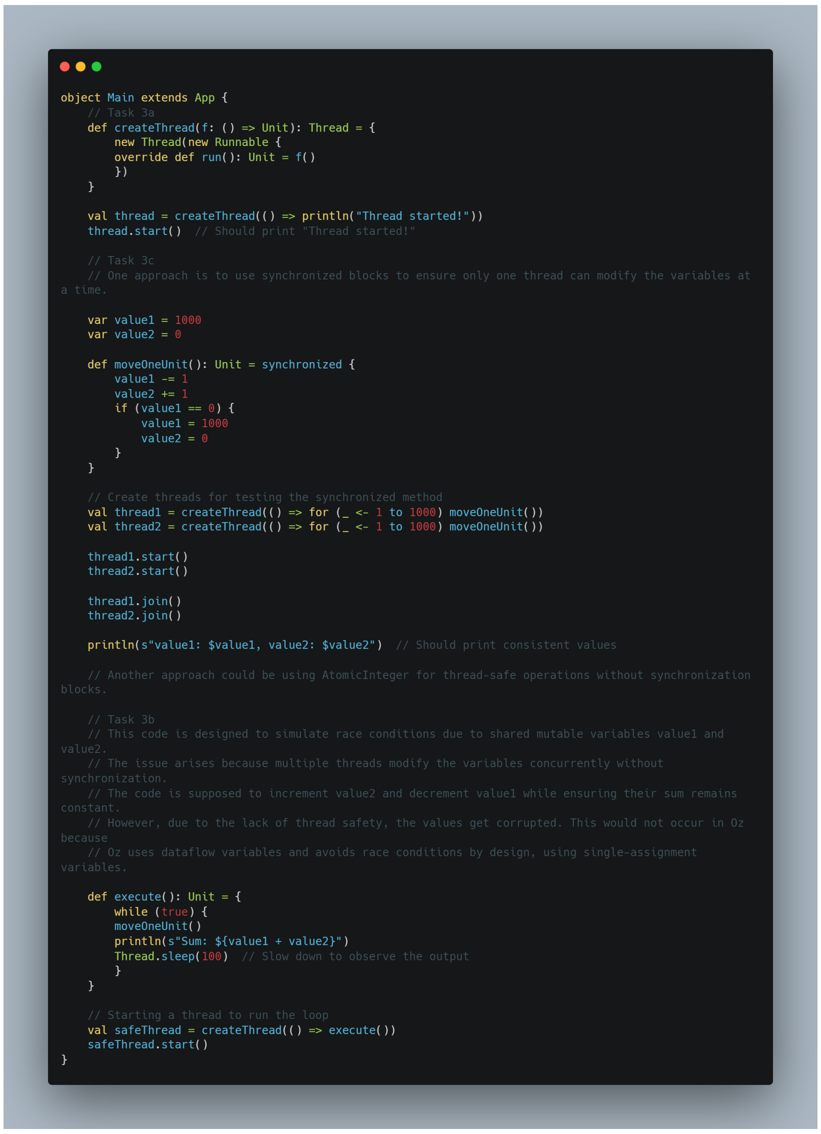


```
[info] running Main
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
 46, 47, 48, 49, 50
1275
1275
55
[success] Total time: 6 s, completed 8. okt. 2024, 20:13:47
```

Figure 4: Task 2 - Compiled code

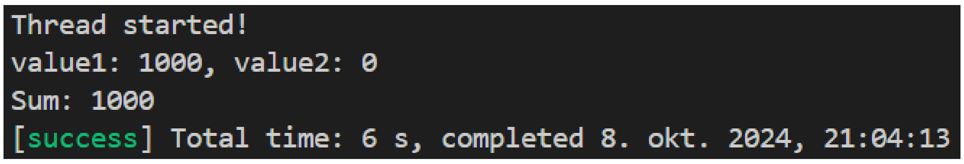
Task: 3 Concurrency in Scala

The goal of this task is to explore the use of threads in Scala to handle concurrency. The task focuses on ensuring thread safety and managing shared state properly, with a demonstration of different techniques like creating custom threads. Figure 5 shows the implemented code and the compiled code is presented in Figure 6.



```scala
object Main extends App {
    // Task 3a
    def createThread(f: () => Unit): Thread = {
        new Thread(new Runnable {
        override def run(): Unit = f()
        })
    }

    val thread = createThread(() => println("Thread started!"))
    thread.start()  // Should print "Thread started!"

    // Task 3c
    // One approach is to use synchronized blocks to ensure only one thread can modify the variables at
a time.

    var value1 = 1000
    var value2 = 0

    def moveOneUnit(): Unit = synchronized {
        value1 -= 1
        value2 += 1
        if (value1 == 0) {
            value1 = 1000
            value2 = 0
        }
    }

    // Create threads for testing the synchronized method
    val thread1 = createThread(() => for (_ <- 1 to 1000) moveOneUnit())
    val thread2 = createThread(() => for (_ <- 1 to 1000) moveOneUnit())

    thread1.start()
    thread2.start()

    thread1.join()
    thread2.join()

    println(s"value1: $value1, value2: $value2")  // Should print consistent values

    // Another approach could be using AtomicInteger for thread-safe operations without synchronization
blocks.

    // Task 3b
    // This code is designed to simulate race conditions due to shared mutable variables value1 and
value2.
    // The issue arises because multiple threads modify the variables concurrently without
synchronization.
    // The code is supposed to increment value2 and decrement value1 while ensuring their sum remains
constant.
    // However, due to the lack of thread safety, the values get corrupted. This would not occur in Oz
because
    // Oz uses dataflow variables and avoids race conditions by design, using single-assignment
variables.

    def execute(): Unit = {
        while (true) {
        moveOneUnit()
        println(s"Sum: ${value1 + value2}")
        Thread.sleep(100)  // Slow down to observe the output
        }
    }

    // Starting a thread to run the loop
    val safeThread = createThread(() => execute())
    safeThread.start()
}
```

Figure 5: Task 3 - Code implementation



```
Thread started!
value1: 1000, value2: 0
Sum: 1000
[success] Total time: 6 s, completed 8. okt. 2024, 21:04:13
```

Figure 6: Task 3 - Compiled code