

2. Short introduction to Scala

Nelma Moreira & **José Proença**

Concurrent programming (CC3040) 2024/2025

CISTER – U.Porto, Porto, Portugal

<https://fm-dcc.github.io/cp2425>



Motivation

Used by many modern concurrency frameworks

- Syntactic flexibility
- Programming models as Embedded Domain Specific Languages
- Many useful features

Safe language

- Automatic garbage collection
- Automatic bound checks
- No pointer arithmetic
- Static type safety

Java interoperability

- Compiled to Java bytecode
- Can use existing Java libraries
- Good interaction with Java's rich ecosystem
- Chosen by some Java-compatible frameworks

Executing Scala

```
object SquareOf5 extends App {  
  def square(x: Int): Int = x * x  
  val s = square(5)  
  println(s"Result: ␣$s")  
}
```

Call stack vs. object heap

where are values stored?

Concurrent threads:

do not share the call stack,
share the object heap,

```
object SquareOf5 extends App {  
  def square(x: Int): Int = x * x  
  val s = square(5)  
  println(s"Result: ␣$s")  
}
```

Call stack vs. object heap

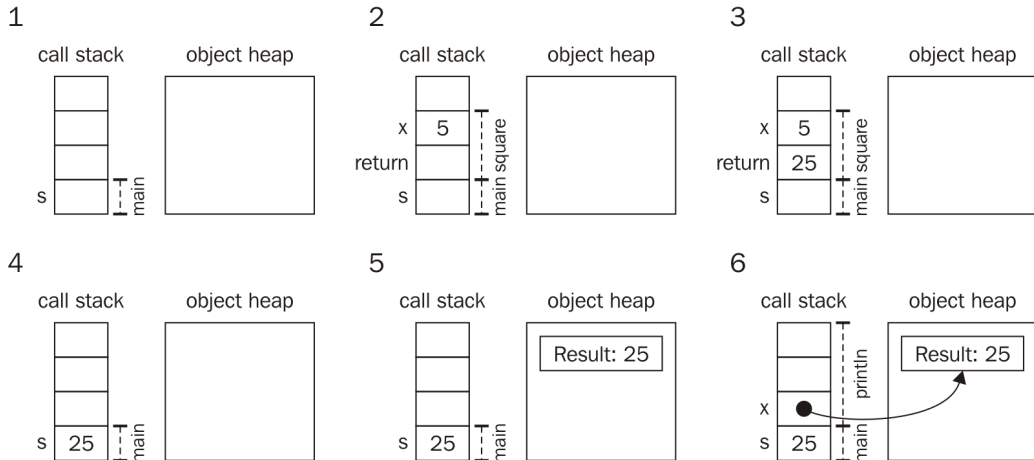
where are values stored?

Concurrent threads:

do not share the call stack,
share the object heap,

Local: SBT (practical lessons)

Online: <https://scastie.scala-lang.org/cIf3BCTQRBybqMcQpYArGA>



[in "Learning Concurrent Programming in Scala", pg. 19]

Scala in a nutshell

```
class Printer(val greeting: String) {  
  def printMessage(): Unit =  
    println(greeting + "!")  
  def printNumber(x: Int): Unit = {  
    println("Number:␣" + x)  
  }  
}
```

One greeting **parameter**
Two **methods**

```
class Printer(val greeting: String) {  
  def printMessage(): Unit =  
    println(greeting + "!")  
  def printNumber(x: Int): Unit = {  
    println("Number:␣" + x)  
  }  
}
```

Using the class

```
val printy = new Printer("Hi")  
           // instantiate  
printy.printMessage() // ?  
printy.printNumber(5) // ?
```

```
object Test {  
  val Pi = 3.14  
}
```

Using the object

```
val x = Test.Pi * 5 * 5  
      // no need to instantiate
```

```
trait Logging {  
  def log(s: String): Unit // just declared  
  def warn(s: String) = log("WARN:␣" + s)  
  def error(s: String) = log("ERROR:␣" + s)  
}  
  
class PrintLogging extends Logging {  
  def log(s: String) = println(s)  
}
```

Using traits

```
val x = new PrintLogging  
val y = new Logging {  
  def log(s:String): Unit =  
    println(s)  
}
```

```
class Pair[A,B](val fst: A, val snd: B)
```

Using Pair

```
val x: Pair[Int,String] = new  
    Pair(4,"a")  
val y = new Pair(2,5) // infer  
    type
```

```
val twice_a: Int=>Int = (x:Int) => x*2
val twice_b = (x:Int) => x*2
val twice_c: Int=>Int = x => x*2
val twice_d: Int=>Int = _ * 2
```

Using lambdas

```
val x = twice_a(4)
```

```
def runTwice(body: =>Unit) = {  
  body  
  body  
}
```

Using Byname

```
runTwice { // prints "Hello" twice  
  println("Hello")  
}
```

```
for (i <- 0 until 10) println(i)
// equivalent to
0.until(10).foreach(i => println(i))

val negatives_a =
  for (i <- 0 until 10) yield -i
val negatives_b =
  (0 until 10).map(i => -1 * i)
```



```
for (i <- 0 until 10) println(i)
// equivalent to
0.until(10).foreach(i => println(i))

val negatives_a =
  for (i <- 0 until 10) yield -i
val negatives_b =
  (0 until 10).map(i => -1 * i)
```

```
val pairs_a =
  for (x <- 0 until 4;
       y <- 0 until 4) yield (x, y)
val pairs_b =
  (0 until 4).flatMap(x =>
    (0 until 4).map(y =>
      (x, y)))
```

Common collections:

Seq[T], List[T], Set[T], Map[K,V]

```
val msgs_a: Seq[String] =  
  Seq("Hello", "world!")  
val msgs_b: List[String] =  
  "Hello" :: "World" :: Nil  
val msgs_c: Set[String] =  
  Set("Hello", "world!")  
val msgs_d: Map[String, Int] =  
  Map("Hello" -> 5, "world!" -> 6)
```

String interpolation:

```
val number = 7  
val msg =  
  s"After $number comes ${number+1}!"
```

```
val successors =  
  Map(1 -> 2, 2 -> 3, 3 -> 4)  
successors.get(5) match {  
  case Some(n) =>  
    println(s"Successor is: $n")  
  case None    =>  
    println("Could not find successor.")  
}
```

```
trait IntOrError  
case class MyInt(i:Int)  
  extends IntOrError  
case class MyError(e:String)  
  extends IntOrError  
  
//...  
def show(ie: IntOrError)  
  ie match {  
    case MyInt(i)    =>  
      println(s"Number: $i")  
    case MyError(e) =>  
      println(s"Error: $e")  
  }  
}
```

```
class Position(val x: Int, val y: Int) {  
  def +(that: Position) =  
    new Position(x + that.x, y + that.y)  
  def *(n: Int) =  
    new Position(x * n, y * n)  
}
```

Using operators

```
val p1 = new Position(3,4)  
val p2 = p1 + p1 * 2 //?
```

File

src/main/scala/cp/lablessons/package.scala

```
package cp

package object lablessons {
  def log(msg: String): Unit =
    println(
      s"${Thread.currentThread.getName}: ␣$msg"
    )
}
```

The log function is used
throughout these lessons

Requires starting with
package cp.lablessons

- Stack and Heap
- Singleton objects
- Traits (similar to Java Interfaces)
- Type parameters
- Lambdas (anonymous functions)
- Byname parameters (lazy)
- “for” expressions
- “for” comprehension
- Scala collections and string interpolation
- Pattern matching
- Operator overloading
- Package objects