Lecture 2
Functional Programming in Scala

21.3.2023

Iflaah Salman

LUT University

Scala as Functional Language

- Functions are like instances or values and can be:
  - Assigned to variables [**val**, **var**]: can hold a reference to a function that makes them part of the type system in Scala.
  - Passed as parameters to functions.
  - Returned as results of functions.
  - Written as function literals.
  - they can be evaluated which results in the function being executed.

Defining Scala Functions

$$(\texttt{parameter list}) \Rightarrow \{\texttt{func body}\}$$

$$(\texttt{x : Int}) \Rightarrow \{\texttt{x} * \texttt{x}\}$$

It takes a single parameter of type Int and returns an Int.
The signature is
(Int): Int

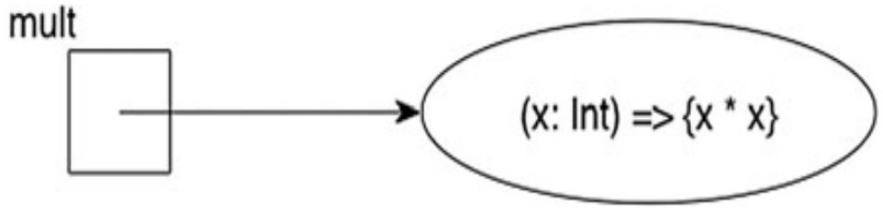
```
object FuncApp1 extends App {

  val mult = (x: Int) => {x * x}

  println(mult(2))
  println(mult(4))

}
```

```
4
16
```

Assigning the function to a local variable or to a property.
Now we can invoke the function via that variable or property.

LUT University

Defining Scala Functions

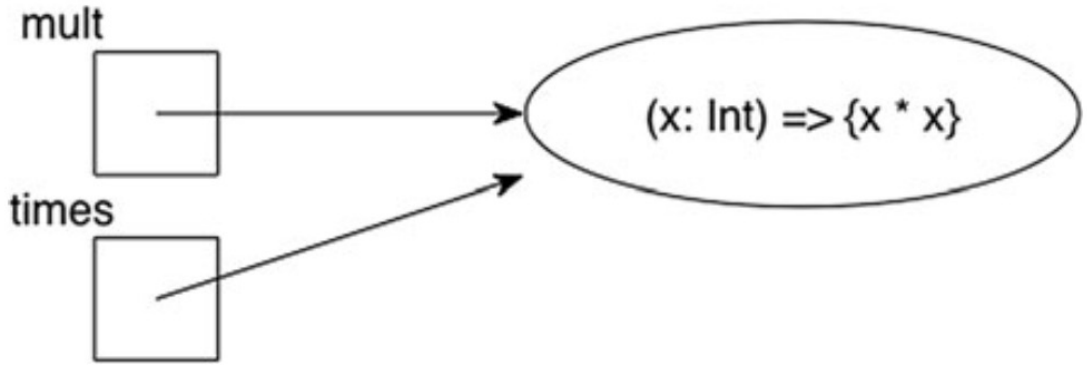


mult holds a reference to a function.

Function is an entity within the language rather than just some code written within an object (as is the case for methods).
Therefore, we can also assign the function referenced by ***mult*** to another variable.

```
val mult = (x: Int) => {x * x}
val times = mult
```

We could now have:




```
println(mult(2))
println(times(2))
```

The results of both statements are the same!
As an effect, ***mult*** and ***times*** are aliases for the same functionality.

LUT University

```
object FunctionLiteralApp extends App {

  // Functional literals can be assigned to variables
  // Note last value assigned is returned as result of
  // function - return type inferred
  var increase = (x: Int) => x + 1
  var y = 1
  println("Initial value of y: " + y)
  y = increase(y)
  println("Increased y: " + y)
  // Can also assign to another identifier
  val aaa = increase
  println("increase using aaa: " + aaa(2))
  // Because increase is a var you can re-assign it
  increase = (x: Int) => x + 99
  y = increase(y)
  println("2nd Increased y: " + y)
  // Can also fundamental change what it is
  // Note can't assign to the parameter X they are vals
  increase = (x: Int) => {
    println("\tIncreasing x")
    println("\tby a fixed amount")
    x + 1
  }
  y = increase(y)
  println("3rd Increased y: " + y)

}
```

Defining Scala Functions!

increase is a **var** we can reassign to it.
Thus we can change the function referenced by ***increase.***

LUT University

Class, Objects, Methods and Functions

- Classes and objects can have both methods and functions defined for them as their members.
- In many cases you can ignore the difference between methods and functions.
  - A method defines behaviour which is tied to the class or object
  - A function defines an operation held by the class or object

```
class Calculator {

  def max(x: Int, y: Int): Int = {
    if (x > y) x else y
  }

  val increment = (x: Int) => x + 1

}

object CalculatorApp extends App {
  val c = new Calculator()
  val a = c.max(2, 3)
  println(a)
  val b = c.increment(3)
  println(b)
}
```

Class, Objects, Methods and Functions

- Calculator that defines both a method ***max*** and a function ***increment***.
- The method **max** is an integral part of the definition of the class Calculator.
- ***increment*** is a read-only **val** property which holds a reference to the function
- Invoking via dot notation.

```
val a = c.max(2, 3)
val b = c.increment(3)
```

This assignment cannot be done for methods because they are not separate entities.

```
val alias = c.increment
println(alias(4))
```

```
class Calculator {

  def max(x: Int, y: Int): Int = {
    if (x > y) x else y
  }

  val increment = (x: Int) => x + 1

}

object CalculatorApp extends App {
  val c = new Calculator()
  val a = c.max(2, 3)
  println(a)
  val b = c.increment(3)
  println(b)
}
```

Class, **Objects**, Methods and Functions

Object

The **object** keyword creates a new singleton type, which is like a class that only has a single named instance.

```
// singleton object
object MySingleton { // the only instance of the MySingleton type
  val mySpecialValue = 53278

  def mySpecialMethod(): Int = 5327

  def apply(x: Int): Int = x + 1
}

MySingleton.mySpecialMethod()
```

```
object Math {

  def max(x: Int, y: Int): Int = {
    if (x > y) x else y
  }

  val increment = (x: Int) => x + 1
}

object MathObjectApp extends App {
  // Can invoke methods
  println(Math.max(2, 3))
  // Can invoke functions
  println(Math.increment(3))
  // Can use named parameters
  println(Math.max(x = 2, y = 3))
  println(Math.max(y = 3, x = 2))
}
```

https://www.youtube.com/watch?v=8V6bMjThNo&list=PLmtsMNDRU0BxryRX4wiwrTZ661xcp6VPM

LUT University

Lifting Methods

Lifting a method means treating a method as if it were a free-standing function.

max on an instance of the class Calculator is assigned to a variable

func of type (Int, Int) => Int

a function that takes two Integer parameters and returns an Integer.

- The method ***max*** is wrapped up in a ***func*** function that would invoke the method.

```
class Calculator {

  def max(x: Int, y: Int): Int = {
    if (x > y) x else y
  }

}

object CalculatorApp extends App {
  val c = new Calculator()
  val func: (Int, Int) => Int = c.max
  println(func(4, 3))
}
```

func holds a reference to a lambda (function literal) that has been created for us by Scala and that references the method ***max***

LUT University

Closure

- A closure (or a lexical closure or function closure) is a reference to a function together with a referencing environment.
- This referencing environment records
  - the context within which the function was originally defined *and*
  - *if necessary* a reference to each of the non-local variables of that function. These non-local or free variables allow the function body to reference variables that are external to the function but which are utilised by that function.

```
object ClosureExamplesApp extends App {

  var more = 100
  val increase = (x: Int) => x + more
  println(increase(10))
  more = 50
  println(increase(10))

}
```

LUT University

Closure!

- it is the current value of **more** that is being used.
- variable **more** is still in scope within the same method as the invocations of the function referenced by ***increase***.

```
object ClosureExamplesApp extends App {

    var more = 100
    val increase = (x: Int) => x + more
    println(increase(10))
    more = 50
    println(increase(10))

}
```

LUT University

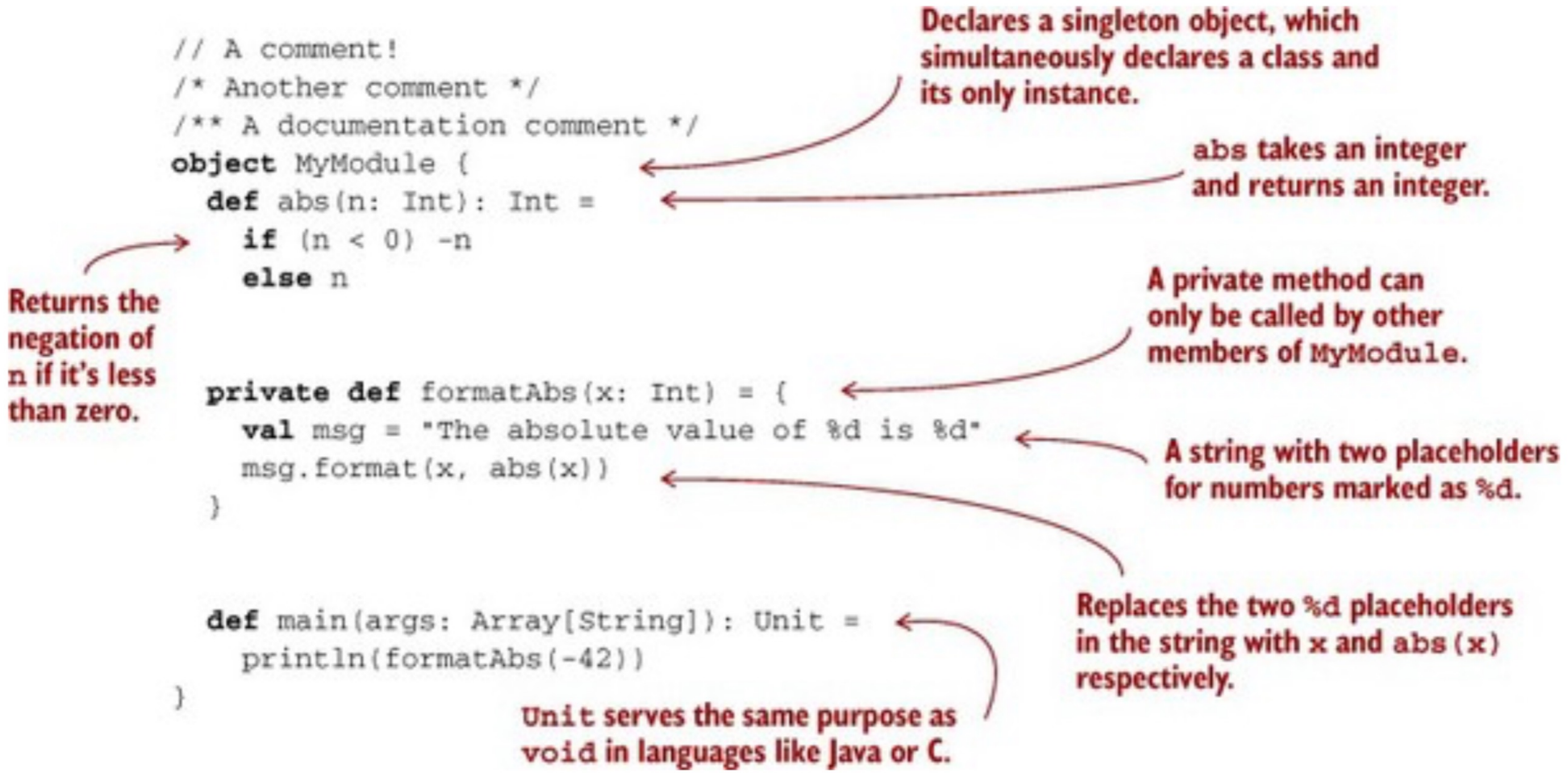
Closure!

- ***resetFunc()*** method has a variable that is local to the method.
- The variable **addition** is used within the method body of a new function definition.
- This new function is then assigned to the property **increment**.
- the second invocation of ***increment*** occurs, back in the ***main*** method after the ***resetFunc()*** method has terminated.
- Normally the variable **addition** would no longer even be in existence, but we have "closure".
- ***increment*** when called the second time in the ***main*** method is the one defined within ***resetFunc()*** and which uses the variable **addition**.

```
object ClosureExamplesApp2 {

  var increment = (x: Int) => x + 1

  def main(args: Array[String]): Unit = {
    println(increment(5))
    resetFunc()
    println(increment(5))
  }

  def resetFunc() {
    // Local variable is bound and stored on the heap
    // as it is used within the function body
    var addition = 50;
    increment = (a: Int) => {
      a + addition

    }
  }

}
```

LUT University

Listing 2.1. A simple Scala program



References

- Chiusano, P., & Bjarnason, R. (2014). *Functional Programming in Scala*. Manning publications.
- Hunt, J. (2018). A Beginner's Guide to Scala, Object Orientation and Functional Programming. In *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer International Publishing. https://doi.org/10.1007/978-3-319-75771-1