

Lecture 3

Higher Order Functions

28.3.2023

Iflaah Salman

Higher Order Function (HOF)

- Functions are values.
- Functions in Scala are part of the type system.
 - just as a parameter can be of type Int, Boolean or Person, **a parameter to a method or a function can be another function.**

**A function that takes a function as a parameter
is referred to as a higher-order function.**

HOFs are functions that do **at least one of the following:**

- Take as a parameter one or more functions.
- Return as output a function

Scala Higher Order Functions

- A function definition defines a function type.
- A type that takes zero or a set of parameters (of given types) and returns a result.

It is a one parameter function that takes an Int and returns an Int— that is its type.

```
(x : Int) => x * 2
```

```
(Parameter type list) => return type
```

Thus a **parameter that expects to be given a reference to a function that takes an int and returns an Int** can be given a reference to any function that takes an Int and returns an Int.

```
(x: Int) => 1
```

```
(y: Int) => y
```

```
(a: Int) => a * 2
```

```
(x: Int) => x * x
```

Scala Higher Order Functions

```
object Processor {  
  val apply = (n: Int, f: Int => Int) => f(n)  
}
```

For example,

```
object HigherOrderFunctionsApp extends App {  
  
  val f1 = (x: Int) => x * x  
  println(Processor.apply(10, f1))  
  
}
```

The output from this application is:

100

The concept of HOF also applies to methods.

```
object Processor {  
  def apply(n: Int, f: Int => Int) = f(n)  
}
```

For example:

```
println(Processor.apply(10, f1))
```

The function **apply** is a higher-order function because its behaviour (and its result) will depend on the behaviour defined by another function—the one passed into it.

Scala HOF!

```
object Processor {  
  val apply = (n: Int, f: Int => Int) => f(n)  
}
```

```
object HigherOrderFunctionsApp extends App {
```

```
  val f1 = (x: Int) => 1  
  val f2 = (y: Int) => y  
  val f3 = (a: Int) => a * 2  
  val f4 = (x: Int) => x * x
```

```
  println(Processor.apply(10, f1))  
  println(Processor.apply(10, f2))  
  println(Processor.apply(10, f3))  
  println(Processor.apply(10, f4))
```

```
}
```

Why to use Higher Order Functions?

The key is:

- We may know that *some function* should be applied to the value 10 but we do not *yet know what it could be*.
- we are *creating a reusable piece of code*
- The piece of code that will apply an appropriate function to the data (we have) when that function is known.

We could have called one of the functions (f1 to f4) directly by passing in the integer to be used.

`f4(10)`

Same result as the following

`Processor.apply(10, f4)`

Higher Order Functions

Scenario

- **Person class** has a **salary** property. **But**, we do not know how to calculate the tax because it is dependent on external factors.
- **calculateTax** method could take an appropriate function that performs that calculation and provides the tax value.
- **class Person** itself does not have a way of calculating the tax
- **taxCalculator** val variable contains a reference to a function.
- The **calculateTax** method is called by passing in the taxCalculator function.

Thus the **class person is a reusable class** that **can have different tax calculation strategies** defined for it.

```
class Person(val salary: Double) {  
    var taxToPay = 0.0
```

```
    def calculateTax(calculator: (Double) => Double) {  
        taxToPay = calculator(salary)  
    }  
}
```

```
object TestPerson extends App {  
    val taxCalculator = (x: Double) => Math.ceil(x * 0.3)  
    val p = new Person(45000.0)  
    p.calculateTax(taxCalculator)  
    println(p.taxToPay)  
}
```

```
TestPerson  
/Library/Java/JavaVirtualMachines/jdk1  
13500.0  
  
Process finished with exit code 0
```

Higher Order Functions in Scala Collections

The **filter** method and the **foreach** method – the two most popular HOFs used with Scala collection classes (**Class List**).

```
def filter(p: (A) => Boolean): List[A]
```

- The parameter to **filter** method is a type of function that takes a single parameter and returns a Boolean. The whole method itself return a new List.
- 'A' is a placeholder for the type held by the List.
- The function p is used to test all the elements in the List. Those that return true are included in the list of values returned.

```
def foreach(f: (A) => Unit): Unit
```

- The foreach method applies a function 'f' to all elements of the list in turn.
- Any result generated by the function 'f' is discarded.

Higher Order Functions in Scala Collections

```
object HigherOrderFunctionTests extends App {  
  // Can use as function literals with various Scala  
  // libraries that provide higher-order functions  
  var list = List(1, 2, 3, 4)  
  // Filter is a higher order function that takes the  
  // function to be used to filter the contents of a list  
  var list2 = list.filter((x: Int) => x > 2)  
  println(list2)  
  // Place holder extreme!  
  list.foreach(x => println(x))  
  // Can replace whole parameter list and  
  // expression with underscore  
  list.foreach(println _)  
  // But Scala can imply the underscore so can  
  // just write  
  list.foreach(println)  
}
```

Higher Order Functions

```
def factorial(n: Int): Int = {  
  def go(n: Int, acc: Int): Int =  
    if (n <= 0) acc  
    else go(n-1, n*acc)  
  
  go(n, 1)  
}
```

```
object MyModule {  
  ...  
  private def formatAbs(x: Int) = {  
    val msg = "The absolute value of %d is %d."  
    msg.format(x, abs(x))  
  }  
  
  private def formatFactorial(n: Int) = {  
    val msg = "The factorial of %d is %d."  
    msg.format(n, factorial(n))  
  }  
  
  def main(args: Array[String]): Unit = {  
    println(formatAbs(-42))  
    println(formatFactorial(7))  
  }  
}
```

An inner function, or local definition

```
def abs(n: Int): Int =  
  if (n < 0) -n  
  else n
```

or strings.


Definitions of abs and
factorial go here.

Higher Order Functions

The two functions, **formatAbs** and **formatFactorial**, are almost identical. If we like, we can generalize these to a single function, **formatResult**, which **accepts as an argument the function** to apply to its argument

```
def formatResult(name: String, n: Int, f: Int => Int) = {  
  val msg = "The %s of %d is %d."  
  msg.format(name, n, f(n))  
}
```

**f is required to be
a function from
Int to Int.**




abs **accepts an Int and returns an Int.**
And likewise,

factorial **accepts an Int and returns an Int,**

Both match the `Int => Int` type.

We can therefore pass **abs** or **factorial** as the **f** argument to **formatResult**.



```
scala> formatResult("absolute value", -42, abs)  
res0: String = "The absolute value of -42 is 42."
```

```
scala> formatResult("factorial", 7, factorial)  
res1: String = "The factorial of 7 is 5040."
```

Polymorphic Functions

- Functions that **operate on only one type of data** are called **monomorphic functions**, e.g., **formatResult** method; `Int => Int`
- HOF that works for **any type it's given** is called a **polymorphic function**.
- **Polymorphic functions** are also called **generic functions**.

Monomorphic function:

```
def findFirst(ss: Array[String], key: String): Int = {  
  @annotation.tailrec  
  def loop(n: Int): Int =  
    if (n >= ss.length) -1  
    else if (ss(n) == key) n  
    else loop(n + 1)  
  
  loop(0)  
}
```

Otherwise, increment **n** and keep looking.

If **n** is past the end of the array, return -1, indicating the key doesn't exist in the array.

ss(n) extracts the **n**th element of the array **ss**. If the element at **n** is equal to the key, return **n**, indicating that the element appears in the array at that index.

Start the loop at the first element of the array.

Polymorphic Functions

```
scala> findFirst(Array(7, 9, 13), (x: Int) => x == 9)
res2: Int = 1
```

```
def findFirst[A](as: Array[A], p: A => Boolean): Int = {
  @annotation.tailrec
  def loop(n: Int): Int =
    if (n >= as.length) -1
    else if (p(as(n))) n
    else loop(n + 1)

  loop(0)
}
```

Instead of hardcoding `String`, take a type `A` as a parameter. And instead of hardcoding an equality check for a given key, take a function with which to test each element of the array.

If the function `p` matches the current element, we've found a match and we return its index in the array.

- We're **abstracting over the type of the array** and the function used for searching it.
- we introduce a comma-separated list of type parameters, surrounded by square brackets (here, just a single `[A]`)
- The type parameter list introduces type variables that can be referenced in the rest of the type signature (*similar to parameters in the argument list that are used in the function body*).
- The fact that the same type variable is referenced in both places in the type signature implies that the type must be the same for both arguments

Partially Applied Functions

A Partially Applied function is a function where **one or more of its parameters** have been **applied or bound to a value**, creating a new function with one fewer parameter than the original.

```
val operation = (x: Int, y: Int) => x * y  
println(operation(2, 5))
```

Now we bind the first parameter to 2 so that it will always double the second parameter and store the resulting function reference into a property **double**.

```
val doubleIt = operation(2, _: Int)
```

- **doubleIt** now references a function that takes one parameter an Int.
- The '' underbar indicates a placeholder for future values, and the type that will be used with this placeholder is Int.

```
println(doubleIt(5))
```

This allows for overloading of functions.
overloading relates to functions with the same names but different parameter types.

Partially Applied Functions

It is also possible to have more than one parameter bound and more than one parameter left unbound.

These parameters can be intermixed with any parameter in any position being bound or unbound as required.

```
var sum = (a: Int, b: Int, c: Int) => a + b + c  
println(sum(1, 2, 3))
```

```
val partialSum = sum(1, _: Int, 3)  
println(partialSum(2))
```

The result of executing this listing is:

6

6

Partially applied functions are useful for creating new functions from existing functions.

Currying

- It is an alternative approach to Partially Applied functions.
- A curried function is a function that is applied to multiple argument lists (in contrast a Partially Applied function has one argument list).
- Note that functions can have one or more parameter lists.

```
class Basic {  
  def sum(x: Int, y: Int) = x + y  
}
```

```
object BasicTest extends App {  
  val test = new Basic  
  println(test.sum(2, 3))  
}
```

a single argument list with multiple parameters.

```
package com.jjh.scala.curry
```

```
class CurryTest {  
  def sum(x: Int)(y: Int) = x + y  
}
```

```
object CurryTest extends App {  
  val test = new CurryTest  
  println(test.sum(2)(3))  
}
```

The same function as a multiple argument function, in which each argument list takes a single argument/parameter.

Currying

- This version results in **two function invocations back to back**.
- It is a bit like **chaining two functions together**.
- The **first function** invocation takes a single Int parameter x and **returns** a function **value for the second function**.
- The **second function** takes the Int parameter Y and **applies it to the first function's result**.

```
package com.jjh.scala.curry
```

```
class CurryTest {  
  def sum(x: Int)(y: Int) = x + y  
}
```

```
object CurryTest extends App {  
  val test = new CurryTest  
  println(test.sum(2)(3))  
}
```

Using Curried Functions

These are all variations on the definition of the curried function.

```
import test._  
def plusTwoLL(y:Int):Int = sum(2)(y: Int)  
def plusTwoL(y:Int) = sum(2)(y: Int)  
def plusTwo(y: Int) = sum(2)(y)  
def plusTwoS(y: Int) = sum(2)_  
def plusTwoRS = sum(2)_
```

Partially Applied functions are more flexible in terms of which parameters are applied/bound and which are not yet applied/bound.

Function Literals

```
scala> findFirst(Array(7, 9, 13), (x: Int) => x == 9)
res2: Int = 1
```

When we define a **function literal**, what is actually being defined in **Scala** is an **object** with a **method** called **apply**.

$(a, b) \Rightarrow a < b$ This is **syntactic sugar** for the following object creation:

```
val lessThan = new Function2[Int, Int, Boolean] {
  def apply(a: Int, b: Int) = a < b
}
```

- `Function2[Int, Int, Boolean]` is usually written `(Int, Int) => Boolean`.
- the `Function2` interface (known in Scala as a trait) has an **apply** method.

```
scala> val b = lessThan.apply(10, 20)
b: Boolean = true
```

- `Function2` is just an **ordinary trait (an interface)** provided by the **standard Scala library** to represent **function objects that take two arguments**.
- The library also provides **Function1, Function3, ... Function X**, taking a number of arguments indicated by the name (In Scala **X** can maximum be **22**).

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>

