Chapter 22
Functional Programming in Scala

22.1 Introduction

This chapter examines the functional programming features of Scala and looks at how they can be used while the next chapter will look at concepts such as Currying and Partially Applied functions.

22.2 Scala as a Functional Language

In Scala functions are *first*-class language constructs just as classes and objects and Value Classes are first-class language types. That is Functions are part of the type system in Scala, a variable can be defined to hold a reference to a function, and functions can be assigned to *vals* and *vars*. They can be passed into methods, constructors and other functions as parameters, etc. They are top-level entities that act as independent units or entities within the language. They can of course be evaluated in which case they are executed and will take in some data as parameters and return a result, etc.

That is, functions are like instances or values and can be:

- Assigned to variables,
- Passed as parameters to functions,
- Returned as results of functions,
- Written as function literals.

In addition, they can be evaluated which results in the function being executed.

Functions should have referential integrity, and this is true of Scala functions just as much as it is true of functions in pure functional languages. Given the same set of inputs, a function should produce the same set of outputs every time. It should thus be possible to replace the function call with the result (at least in theory).

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_22

Thus a Scala function should only transform its inputs into its outputs and should have no (hidden) side effects. That is, it should not change the state of the system nor should it be involved in any state-based behaviour.

22.3 Defining Scala Functions

A function definition can be anonymous as it defines a type and this it is an entity in its own right. For example, the syntax for a function definition is:

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

An example of a function definition (or functional literal) is:

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

This defines a function that takes an Int and returns an Int where the body of the function is defined as x * x. The signature of this function (sand thus its type) is

- (Int): Int

That is, it takes a single parameter of type Int and returns an Int.

Note that this function definition matches the criteria we specified for functions—the function takes in a single Int and returns a value derived purely based on this parameter (it multiplies itself by itself). We could thus replace it by its result.

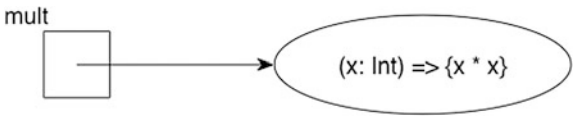
As it stands it is of limited value as the function merely defines a new function. However, we can assign this to a local variable or to a property. Once we have done this we can invoke the function via that variable or property. As a very simple example consider the following same application:

```
object FuncApp1 extends App {

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

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

}
```

The result of running this simple application is:

```
4
16
```

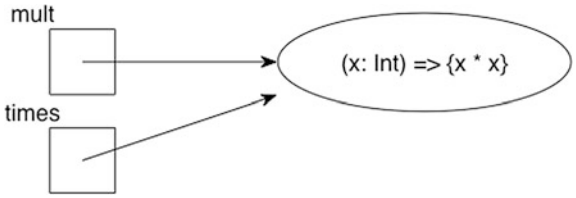
In this example within the object FunctionTest1 we have defined a function (x:Int) => {x * x} and assigned that function to mult. In essence, we have the following:



This illustrates that a function is an entity within the language rather than just some code held within an object (as is the case for methods). This means we can also assign the function referenced by mult to another variable, thus we could write:

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

We could now have:



We can thus access the function via either mult or times.

In the earlier example application, we accessed the function via the val mult, for example,

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

The result of this expression is that the function referenced by mult is evaluated by passing in the value 2. This value is bound to the variable x and the body of the function is executed. Which results in the expression 2 * 2 being processed resulting in 4 being returned by the function. This is then passed to the println method and printed out to the console.

Notice that we could now also have written

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

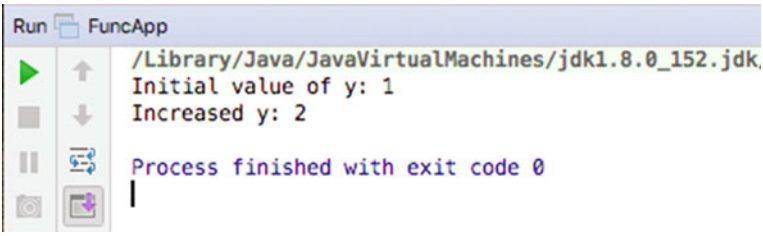
And the same function would have been evaluated. Although in reality mult and times reference the same function definition, the effect is that mult and times are aliases for the same functionality.

As another example, consider the following lines of code.

```
object FuncApp extends App {

  var increase = (x: Int) => x + 1
  var y = 1
  println("Initial value of y: " + y)
  y = increase(y)
  println("Increased y: " + y)

}
```

In this block of code a variable *increase* is defined that references a function. This function takes an Int and adds one to whatever value is passed into it. As the '+' operator returns a result this is used as the value returned by the function. In this

case a variable y has an initial value 1 assigned to it. The result generated by the function referenced by the *increase* variable is then assigned to y. The output is



Note that as *increase* is a var we can reassign to it—thus we can change the function referenced by *increase*, for example,

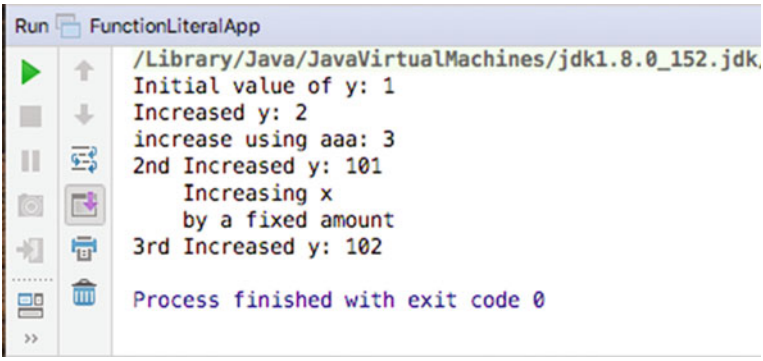
```
increase = (x: Int) => x + 99
y = increase(y)
```

The variable increase now references a function that adds 99 to the integer passed to it. Thus if we invoke *increase* it now appears that its functionality has changed. Be careful of doing this indiscriminately as it can make programs harder to understand and debug.

The above samples are grouped together in the following listing as a set of worked examples:

```
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)

}
```

The result of executing this simple application is shown below:



22.4 Class, Objects, Methods and Functions

Classes and objects can have both methods and functions defined for them. In many cases you can ignore the difference between the two, they are just parts, or members of, the class or object:

- A method defines behaviour which is tied to the class or object and which can be invoked via the dotation and
- A function defines an operation held by the class or object that can be invoked via the dot notation.

For example:

```
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)
}
```

In the above listing we have defined a class Calculator that defines both a method max and a function increment. From the client of the class Calculator they look the same. They are both invoked via the dot natation, for example,

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

However, the way in which they were defined gives a hint to the differences. The method is an integral part of the definition of the class Calculator. The method is literally part of the fabric of that class. In contrast increment is a read-only val property which holds a reference to the function (x: int) => x + 1. This means that the function referenced by the property could be assigned to another variable, for example,

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

When we execute these lines the result is

```
5
```

Such assignment of functionality is not available for methods (as they are not separate entities in the way that functions are). This also means that functions can be passed as parameters into other functions or method, etc.

The above example uses a class to define the method max and the function increment. We could also have defined them within an object, for example,

```
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))
}
```

Although this is now an object, everything that was said about methods and functions for classes is also true for methods and class within an object. Thus the max method is an integral part of the object Math, whereas increment is a read-only property holding a reference to a function that increments the value passed to it.

22.5 Lifting a Method

Above we said that one of the differences between a function and a method is that the method is tied to the class/object it is defined in; whereas the function is really a free-standing element that just happens to have been defined within a class or an object.

While this is true, it does not actually mean that we cannot treat a method as if it was a free-standing function. A concept known as *lifting* can be used to apparently convert a method into a function, for example,

```
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))
}
```

In the above listing the method max on an instance of the class Calculator is assigned to a variable func of type (Int, Int) => Int; that is a function that takes two Integer parameters and returns an Integer.

What is happening here is that the method max has been lifted to a function; in practice this means that it is wrapped up in a function that will invoke the method for us.

If we look at this in the IntelliJ debugger we can see that func holds a reference to a lambda (function literal) that has been created for us by Scala and that references the method max on the instance held in the val c.

22.6 Single Abstract Method Traits

Chapter 19: Further Traits introduced the concept of a single abstract method (SAM) trait. This is a trait that contains a single abstract method that can be *implemented* via a functional literal (lambda).

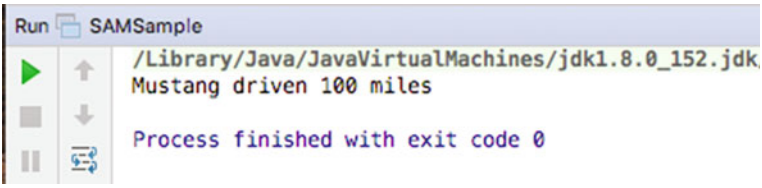
An example of such a SAM trait was given and is reproduced here:

```
trait Drivable {
  // exactly one abstract method
  def drive(miles: Int): Unit
}
```

A function literal can be used to provide a concrete implementation of the trait as shown below:

```
object SAMSample extends App {
  val d1: Drivable = (m: Int) => println(s"Mustang driven $m miles")
  d1.drive(100)
}
```

The output of this sample application is



However, it is worth noting the difference between using a function literal (or lambda) to implement a SAM trait and merely creating a reference to a function literal.

```
object SAMSample extends App {
  val d1: Drivable = (m: Int) => println(s"Mustang driven $m miles")
  d1.drive(100)
  // d1.apply(100)
  // d1(100)

  val f1 = (m: Int) => println(s"Mustang driven $m miles")
  f1.apply(100)
 f1(100)
 // f1.drive(100)

}
```

In the above code the val d1 is of type Drivable, whereas the val f1 is function of type Int => Int (that is it is a function that takes an Integer and returns an Integer).

This means that the method drive can be invoked on the object referenced by d1 but it cannot (quiet obviously be invoked on f1).

In turn both the apply method and the () operator can be used with the function f1, but they have no meaning for a Drivable type.

22.7 Closure

One question that might well be on your mind now is what happens when a function references some data that is in scope where it is defined but is no longer available when it is evaluated? This question is answered by the implementation of a concept known as closure.

Within computer science (and programming languages in particular) a closure (or a lexical closure or function closure) is a function (or more strictly 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. This referencing environment is one of the distinguishing features between a functional language and a language that supports function pointers (such as C).

The general concept of a lexical closure was first developed during the 1960s but was first fully implemented in the language Scheme in the 1970s. It has since been used within many functional programming languages including LISP, ML and Scala.

At the conceptual level, closure allows a function to reference a variable available in the scope where the function is originally but not available by default in the scope where it is executed.

For example, in the following simple program, the variable more is defined outside the body of the function referenced by increase. This is permissible as the variable is defined within the body of the main method of the ClosuresTest object, as is the function. Thus the variable more is *within scope* at the point of definition.

```
object ClosureExamplesApp extends App {

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

}
```

Within the main method we then invoke the *increase* method by passing in the value 10. This is done twice with the variable more being reset to 50 between the two. The output from this program is shown below:
The result of running this simple application is:

```
110
60
```

Note that it is the *current* value of more that is being used with the function executes and not the value of more present at the point that the function was defined. Hence the out is 110 and 60 that is 100 + 10 and then 50 + 10.

This might seem obvious as the variable more is still in scope within the same method as the invocations of the function referenced by *increase*. However, consider the following example:

```
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
    }
  }

}
```

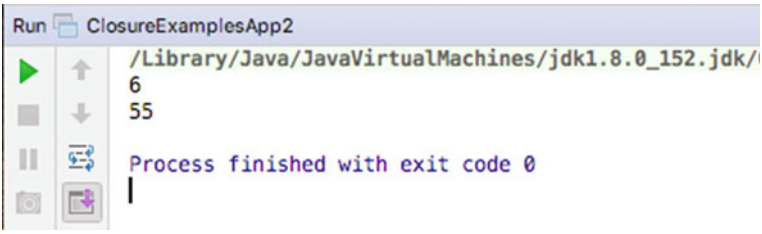
In the above listing a property *increment* holds a reference to a function. Initially the function being referenced adds 1 to whatever value has been passed to it. In the main method this function is called with the value 5 and the result returned by the function is printed. This will be the value 6.

However, after this a second method, resetFunc() is invoked. This method has a variable that is local to the method. That is, *normally* it would only be available within the method resetFunc. This variable is called addition and has the value 50.

The variable *addition* is however, used within the method body of a new function definition. This function takes an integer and adds the value of addition to that integer and returns this as the result of the function. This new function is then assigned to the property *increment*.

Now, when the second invocation of increment occurs, back in the main method, the resetFunc() method has terminated and *normally* the variable addition would no longer even be in existence. However when this program runs the value 55 is printed out from the second invocation of increment. That is the function being referenced by *increment* when it is called the second time in the main method is the one defined within resetFunc() and which uses the variable addition.

The actual output is shown below:



So what has happened here? It should be noted that the value 50 was not copied into the second function body. Rather it is a concrete element of the use of a reference environment used with the closure concept. Scala ensures that the variable addition is available to the function, even if the invocation of the function is somewhere different to where it was defined by binding any free variables (those defined outside the scope of the function) and storing them so that they can be accessed for the functions context (in effect moving the variable from the local stack to the heap).

22.8 Referential Transparency

An important concept within the world of functional programming is that of referential transparency. An operation is said to be referentially transparent if it can be replaced with its corresponding value, without changing the programs behaviour, for a given set of parameters.

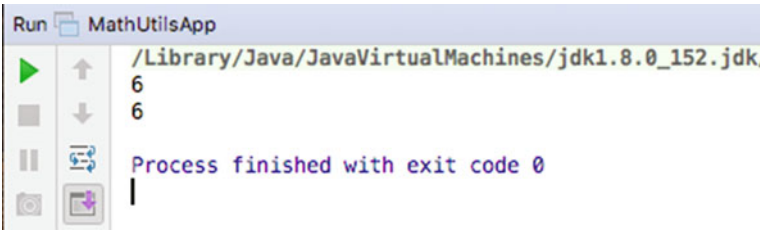
For example, let us assume that we have defined the function increment on an object MathUtils as shown below.

```
object MathUtils {

  def increment(x: Int) = x + 1

}
```

If we use this simple example in an application to increment the value 5:

```
object MathUtilsApp extends App {
  println(MathUtils.increment(5))
  println(MathUtils.increment(5))
}
```

We can say that the operation (in this case a method but it could be a function) is referentially transparent (or RT) if it always returns the same result for the same value (i.e. that increment(5) always returns 6):



Any function or method that references a value which has been captured from its surrounding context and which can be modified cannot be guaranteed to be RT. Thus some of the examples presented earlier, such as that in the ClosureExamplesApp example is therefore not guaranteed to be referentially transparent. This can have significant consequences for the maintainability of the resulting code.

A closely related idea is that of No Side Effects. That is a function should not have any side effects, it should base its operation purely on the values it receives and its only impact should be the result returned. Any hidden side effects again make software harder to maintain.

Of course within most applications there is a significant need for side effects; for example any logging function has a side effect of recording the logged information somewhere (typically in a file), any database updates have some side effect of updating the database.

However, for pure functions it is a useful consideration to follow.