Unit 4a: First Class Functions & Nesting

1 Introduction

In programming, a ‘function’ is a self-contained module of code that accomplishes a specific task; functions provide a way to make code modular. In OO programming these belong to classes and are called methods.

In mathematics, the word ‘function’ has a more specific meaning:

> *A function is a relation, or mapping between a set of inputs and a set of permissible outputs with the property that each input is related to exactly one output*

For example:

$f(x) = x^2$

maps any number (x) to the square of that number, e.g. f(3) = 9, f(4) = 16.

For given input x you always get the same result; you don’t change x in process.

Functional programming is based on the mathematical definition.

2 Composition

Composition is a combination of individual units to perform a more complex task. In OO programming we compose objects to model a complex scenario, so classes are the building blocks of OO programming.

In functional programming, we can compose functions rather like the way mathematical functions can be composed. Functions are the building blocks of functional programming.

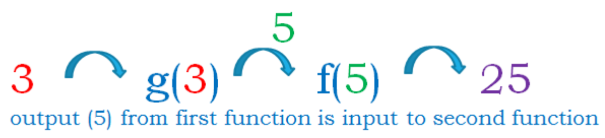
For example, mathematical functions f, g and h can be defined as

f(x) = x^2

g(x) = x + 2

h(x) = f(g(x)) - h is a composition of f and g

So h(3) = f(g(3)) = f(3+2) = f(5) = 25



What about (g(f(x))?

3 First class functions

The core value of functional programming is that functions should be *first-class*. They can be declared and invoked and can be used in every segment of the language as just another data type.

A function declaration is of the form:

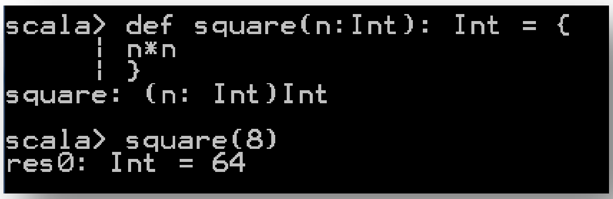
```
def functionName([list of parameters]) : [return type] =
{
    function body
    return [expr]
}
```

A function that doesn't return anything can return a Unit (equivalent to void in Java); the return statement is optional, if not present the function will return the last expression evaluated in the body.

3.1 Declaring a function

You can declare a function in REPL; in a worksheet and as part of class/object:

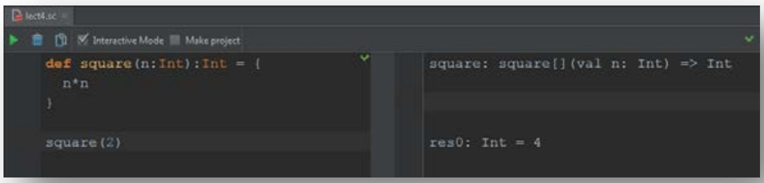
3.1.1 Standalone in REPL



```
scala> def square(n:Int): Int = {
     | n*n
     | }
square: (n: Int)Int

scala> square(8)
res0: Int = 64
```

3.1.2 Scala Worksheet



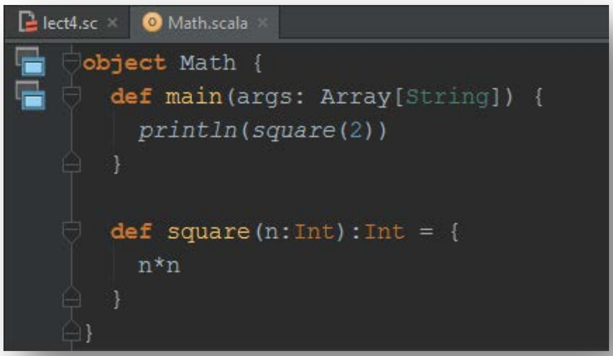
```
def square(n:Int):Int = {
  n*n
}

square(2)
```

```
square: square[](val n: Int) => Int


res0: Int = 4
```

3.1.3 In a class/object as a method



```
object Math {
  def main(args: Array[String]) {
    println(square(2))
  }

  def square(n:Int):Int = {
    n*n
  }
}
```

3.2 Function examples

3.2.1 Example 1

The function *higher* takes two **Int** parameters and returns an **Int**. Note that the value returned is the expression evaluated in whichever branch of the *if* statement (i.e. the value of x or y) is taken.

```
scala> def higher(x:Int, y:Int):Int = {
  if(x >= y) x
  else y
}
higher: (x: Int, y: Int) => Int
```

In REPL, the function definition is evaluated and displayed - this shows that the *value of the function definition is a function* of two **Int**s that returns an **Int**.

```
scala> higher(2,4)
res0: Int = 4
```

3.2.2 Example 2

```
def alert(message:String):Unit = {
  println(message)
}

alert("Hi")
```

This function *alert* simply prints the value of the **String** parameter; it does not return any value, so the return type is *Unit*.

3.2.3 Example 3

These function examples are declared as *function literals* (or anonymous functions) using lambda expressions. Each function is assigned to a variable – *the variable refers to a function*, not to a value, and you can call the function using the variable name.

```
var s = (n:Int) => n*n

s(3)
```

This is a simple function where the body is a single expression. Here is an example function with a multiline body:

```
var h = (x:Int,y:Int) => {
  if(x > y) x
  else y
}
h(3,4)
```

3.2.4 Example 4

This example shows a function with a variable number of parameters, indicated by the * in the function declaration:

```
def sum(x:Int*):Int = {
  var result = 0
  for(i <- x){
    result = result + i
  }
  result
}

sum(1,2,3)
sum(3,4,5,6,7)
```

Note this example does work, and does illustrate variable parameters – but is it written in a functional programming style?

4 Nested functions

You can declare a function *anywhere*, including nested inside another function.

4.1 Example

This example is a function *min* that declares a nested, or inner, function *lower* to break down the job of calculating the minimum of three numbers into calculating the lower of two numbers twice.

```
def min(x: Int, y: Int, z: Int) = {
  def lower(i: Int, j: Int) = {
    if (i < j) i else j
```

```
        }
        lower(x,lower(y,z))
    }

    min(4,2,9)
```

How might you have written the last line previously? You might have done it like this:

```
    var result = lower(x,y)
    result = lower(result, z)
```

That would work, but it is not as “functional” as it could be as it explicitly defines steps and stores an intermediate result. The version here simply evaluates an expression with function calls.

Nested functions are often used to help optimise recursion as seen previously, although this example is not recursive.