Scala

Session 4

Muhammad Asif Fayyaz

Course Contents

1. The Basics
2. Getting up to speed
3. **Higher-Order functions**
4. **Classes and Objects**
5. Inheritance and Traits
6. Collections

Today’s Session (Getting Up to Speed)

- First-Class Functions
- Function Types
- Anonymous Functions
- Shortening Anonymous Functions
- Place Holders
- Currying
- Tail Recursion
- Classes and Objects

Higher Order (First Class) Functions

- Scala treat functions as first-class values.

- This means that like any other value, a function can be passed as a parameter and returned as a result.

- This provides a flexible way to compose programs.

- Functions that take other functions as parameters or that return functions are called higher order functions

Example

Take the sum of the integers between a and b:

```
def id(x: Int) : Int = x

def sumInts(a: Int, b: Int): Int =

     if (a > b) 0 else id(a) + sumInts(a + 1, b)
```

Take the sum of the cubes of all the integers between a and b :

```
def cube(x: Int): Int = x * x * x
def sumCubes(a: Int, b: Int): Int =
    if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Take the sum of the factorials of all the integers between a and b :

```
def fact(x: Int): Int = if (x ==0) 1 else x * fact(x-1)

def sumFactorials(a: Int, b: Int): Int =

     if (a > b) 0 else fact(a) + sumFactorials(a + 1, b)
```

The code above looks very similar. May be we can improve it

Summing with Higher-Order Function

Let’s define:

```
 def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0
    else f(a) + sum(f, a + 1, b)
```

We can then write:

```
def sumInts(a: Int, b: Int) = sum(id, a, b)

def sumCubes(a: Int, b: Int) = sum(cube, a, b)

def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

where

```
def id(x: Int): Int = x

def cube(x: Int): Int = x * x * x

def fact(x: Int): Int = if (x == 0) 1 else x * fact(x - 1)
```

Function Types

- Types are defined as
    - A => B
        - This is a type of function that takes argument of type A and return a result of type B
    - Example
        - Int => Int
        - (Int, Int) => Int
- Lets have a look again at the example

Anonymous Functions

- Passing function as arguments lead to defining many small functions.
  - Tedious
- We should be able to do the same as
  - def str = “Hello”; println(str)
  - can be rewritten as println(“Hello”) using string literal
- These are called *anonymous functions*

Anonymous Function Syntax

- (x: Int) => x * x * x
  - Takes parameter of type Int and return cube
  - The type of parameter can be omitted if it can be inferred by the compiler from the context

- If there are several parameters
  - (x: Int, y: Int) => x + y

- An anonymous function `(x1: T1,....xn:Tn) => E` can always be expressed using def

  ```
  def f(x1: T1,....xn:Tn) = E; f
  ```

Anonymous Function Usage

- Heavily used in scala libraries
- Examples
  - operations on data structures like List, Seq, Hashtable
    - foreach
    - filter

Summation with Anonymous Functions

- Using anonymous functions, we can write sums in a shorter way:

```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x*x*x, a, b)
```

Shortening Anonymous Function

- Shortening the syntax
- Types can be omitted if can be inferred
- Using Placeholders (Further make the syntax concise)

Currying

Lets have a look again at sum function

```
def sum(f: Int => Int, a: Int, b: Int): Int =
    if (a > b) 0
    else f(a) + sum(f, a + 1, b)

def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

Note that (a,b) are passed unchanged from sumInts and sumCubes to sum function.

Lets try to make this syntax even shorter

Currying (Cont…)

Lets rewrite sum as follows <<code>>

```
def sum(f: Int => Int): (Int, Int) => Int = {

  def sumF(a: Int, b: Int): Int =
    if (a > b) 0
    else f(a) + sum(f, a + 1, b)

  sumF

}
```

Sum is now a function that returns another function

The returned function sumF applies the given function parameter f and sums the results.

Currying -> Multiple Parameter Lists

- The definition of function that returns functions is so useful in functional programming that there is a special syntax for it in Scala.
- For example, the following definition of sum is equivalent to the one with the nested sumF function, but shorter:

```
def sum(f: Int => Int)(a: Int, b: Int): Int = {
    if (a > b) 0 else f(a) + sum(f, a + 1, b)
def cube = (x: Int) => x * x * x
sum(cube)(1,3)
```

This style of carried functions is called currying

Tail Recursion

- Which of the implementation is better

```
def factRec(x: Int): Int =  if (x == 0) 1 else x * factRec(x - 1)


def factIter(x: Int) = {
  var result = 1
  for (i <- x to 0 by -1) {
      result *= i
  }
  result
}
```

- A tail call is a function call performed as the final action of the function.
- Tail recursion is a special case of recursion where the calling function does no more computation after making a recursive call.

Classes and Objects

- Private members
- Creating and Accessing Objects
- Inheritance and Overriding
- Parameterless Methods
- Abstract Classes
- Traits
- Implementing Abstract Classes
- Dynamic Binding
- Objects
- Companion Objects
- Standard Classes

Partial Functions

Closure