Scala

Session 5

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 (Classes and Objects)

- Simple Classes and Parameterless Methods
- Properties with Getters and Setters
- Singleton Objects
- Companion Objects

Lets revise a bit what we covered

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

Lets Start with the 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 squares of all the integers between a and b :

```
def square(x: Int): Int = x * x

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

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 sumSquares(a: Int, b: Int) = sum(square, a, b)
```

// where

```
def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x
def square(x: Int): Int = x * x
```

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

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

- 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.
- << Code >>

Simple Classes and Parameterless Methods

- Simple Class Definition

```
class Counter {
    private var value = 0              // You must initialize the field
    def increment() { value += 1 } // Methods are public by default
    def current() = value
}
```

- To use this class

```
val myCounter = new Counter // Or new Counter()
myCounter.increment()
println(myCounter.current)
```

- To call a parameterless method (with or without parentheses)

```
myCounter.current // OK
myCounter.current() // Also OK
```

- Which form should we use?

Properties with Getters and Setters

- Scala provides getter and setter methods for every field

```
class Person {
    var age = 0
}
```

```
$ scalac Person.scala
$ scala -private Person
Compiled from "Person.scala"
public class Person extends java.lang.Object implements scala.ScalaObject{
  private int age;
  public int age();
  public void age_$eq(int);
  public Person();
}
```

Properties with Getters and Setters

- Four choices for implementing properties
  - var foo : Scala synthesizes a getter and a setter.
  - val foo : Scala synthesizes a getter.
  - You define methods foo and foo_= .
  - You define a method foo .

Object-Private Fields

- In Scala (like in Java or C++), a method can access the private fields of **all objects** of its class. For Example,

```
class Counter {
  private var value = 0
  def increment() { value += 1 }
  def isLess(other : Counter) = value < other.value
    // Can access private field of other object
}
```

- Scala allows even more severe access restriction with private[this] qualifier. This field is called object-private field.

```
private[this] var value = 0 // Accessing someObject.value is not allowed
```

Auxiliary Constructors

- Each auxiliary constructor must start with a call to a previously defined auxiliary constructor.

```
public class Person { // This is Java
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String name() { return this.name; }
  public int age() { return this.age; }
    ...
}
```

Primary Constructor

- The parameters of the primary constructor are placed immediately after the class name.

```
class Person(val name: String, val age: Int) {
 ...
}
```

- This one line is equivalent to following code in Java

```
public class Person { // This is Java
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String name() { return this.name; }
  public int age() { return this.age; }
    ...
}
```

Primary Constructor Parameters

Primary Constructor parameter	Generated Field/Methods
name: String	object-private field, or no field if no method uses name
private val / var name: String	private field, private getter / setter
val / var name: String	private field, public getter / setter

Nested Classes

- In Scala, you can nest just about anything inside anything
  - functions inside functions
  - classes inside other classes

```
class Network {
        class Member(val name: String) {
                val contacts = new ArrayBuffer[Member]
        }
        private val members = new ArrayBuffer[Member]
        def join(name: String) = {
                val m = new Member(name)
                members += m
                m
        }
}
```

Objects - Singletons

- Scala has no static methods or Fields
- Instead they use object construct
- An object defined a single instance of a class with the features that you want

```
object Accounts {
  private var lastNumber = 0
  def newUniqueNumber() = { lastNumber += 1; lastNumber }
}
```