Scala IV

Muhammad Tahir

Lecture 14

Contents

The Keyword Implicit

- The keyword implicit can be used in three contexts
  - implicit function
  - implicit class
  - implicit parameters
- An implicit function provides extension method and is called automatically when required by the compiler and is in the scope
- The implicit keyword makes the class's primary constructor available for implicit conversions when the class is in the scope

The Keyword Implicit Cont'd

- An implicit class can not be defined at the top level and must be defined inside a class, object, or package object
- An implicit parameter to a function is annotated with the keyword implicit
- Implicit parameter values are passed automatically by the compiler when not provided by the programmer
- Compiler looks for implicit parameter values in its implicit scope

Implicit Function

```
import scala.language.implicitConversions

class Implicit_Function (i : Int) {
      // Explicit conversion to Seq (List) and then increment
      val i_explicit_seq_inc = Seq(i).map {
            n => n + 1
      }

      val i_inc = i + 2

      // Implicit conversion to Seq (List) and then increment
      val i_implicit_seq_inc = i.map(_ + 3)

      // Implicit function for conversion
      implicit def any_name(i: Int): Seq[Int] = Seq(i)
}

val convert = new Implicit_Function(5)
println("Explicit conversion and increment: " + convert.i_explicit_seq_inc)
println("Simple increment: " +convert.i_inc)
println("Implicit conversion and increment: " +convert.i_implicit_seq_inc)

// The output at the terminal
Explicit conversion and increment: List(6)
Simple increment: 7
Implicit conversion and increment: List(8)
```

Implicit class

```
class StringModifier(s: String) {
     def increment = s.map(c => ((c + 1).toChar).toUpper)
}

val result = (new StringModifier("hal")).increment
println("hal is modified to: " + result)

// The output at the terminal
hal is modified to: IBM
```

```
implicit class StringModifier(s: String) {
  def increment = s.map(c => ((c + 1).toChar).toUpper)
}

val result = ("hal").increment
println("hal is modified to: " + result)

// The output at the terminal
HAL is modified to: IBM
```

Implicit Parameters

- A method or constructor can have only one implicit parameter list
- It must be the last parameter list in the set of parameters
- In case several arguments match implicit parameter's type, the most specific is chosen using the rules of static overloading resolution

Implicit Parameters Cont'd

```
class LinearMap {
  def multiplyOffset(x: Int) (implicit weight: Int, offset: Float) =
      (weight * x).toFloat + offset
}

// initial value assignment
implicit val weightage = 3
implicit val offset = 4.0F
val x = 5

val scale = new LinearMap
println("The result with Implicit parameters omitted: " + scale.
    multiplyOffset(x))
println("The result with Implicit parameters passed explicitly: " +
    scale.multiplyOffset(x)(weightage, offset))

// output at the terminal is
The result with Implicit parameters omitted: 19.0
The result with Implicit parameters passed explicitly: 19.0
```

Lazy val

- The `val`
  - is evaluated only once at the time of definition
  - once evaluated, the same value is used for all future references (without reevaluation)
- The `def`
  - is used to define functions or methods
  - is not evaluated at the time of definition
  - evaluated when called and is evaluated every time whenever called

Lazy val Cont'd

- The `lazy val`
  - is not evaluated at the time of definition
  - evaluation is delayed till its use for the first time
  - once evaluated, the same value is used for all future references (without reevaluation)

Lazy val Cont'd

```
// val evaluation illustration
object valApp extends App{
  val x = { println("x is initialized,"); 99 }
  println("before we access 'x'")
  println(x + 1)
}

// Output at the terminal
x is initialized,
before we access 'x'
100
```

Listing 1: The `val` evaluation.

Lazy val Cont'd

The `lazy val` evaluation

```
// lazy val illustration
object lazyvalApp extends App{
  lazy val x = { println("x is NOT initialized."); 99 }
  println("Unless we access 'x',")
  println(x + 1)
  println(x + 2)
}
```

Lazy val Cont'd

The `lazy val` evaluation

```
// lazy val illustration
object lazyvalApp extends App{
  lazy val x = { println("x is NOT initialized."); 99 }
  println("Unless we access 'x',")
  println(x + 1)
  println(x + 2)
}
```

```
// Output at the terminal
Unless we access 'x',
x is NOT initialized.
100
101
```

Scala Non-Strict (lazy) Collections

- Scala collection operations can be grouped in two categories
  - `transformation` operations (also termed as `transformers`) that return another collection (e.g. filter, map, zip),
  - `reduction` operations that result in a single value (e.g. isEmpty, foldLeft, reduce).
- There are two ways to implement `transformers`
  - *strict*: a new collection with all elements is constructed due to transformation
  - *lazy*: constructs only a representative (proxy) for resulting collection, while the elements are constructed on demand

Scala Non-Strict (lazy) Collections Cont'd

- By default Scala collections are strict when implementing transformers (except `Stream`)
- Stream implements all its transformer methods lazily
- However, every collection can be turned into a lazy one and vice versa, using `views`
- The `view` is a special kind of collection representing some base collection, but implements all its transformers lazily
- Going from a collection to its view, requires invocation of view method on the collection

View Method

```
val uVec = Vector(1 to 10: _*)

// variant of expression uVec.map(_ + 1).map(_ * 2)
val uVecMapped = uVec map(_ + 1) map (_ * 2)  // syntactic
    sugar

println(uVec)
println(uVecMapped)

// Output at the terminal
Vector(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

- expression `uVec map (_ + 1)` constructs a new second vector
- the second vector is then transformed to a third vector by the call to `map (_ * 2)`

View Method Cont'd

- For performance improvement, avoid evaluation of intermediate results
- Can be achieved by first converting (using view method) the vector to its equivalent `view` collection
- Apply all transformations to the resulting `view` collection
- Finally force the `view` collection to the vector

View Method Cont'd

```
val uVec = Vector(1 to 10: _*)

// view method and resulting view collection
val uVecView = uVec.view
val iView = uVecView map (_ + 1) map (_ * 2) filter (_ > 0)
val uVecMapped = iView.force

println(iView)
println(uVecMapped)

// Output at the terminal
SeqViewMMF(...)
Vector(4, 6, 8, 10, 12, 14, 16, 18, 20, 22)
```

Reading List I

- For lazy val see Chapter 20 of [Odersky et al., 2016]
- Read Chapter 24 of [Odersky et al., 2016] for view collections
- The immutable collections are listed at [Scala-Collections-API, 2020] and is a good resource for quick reference

References

Odersky, M., Spoon, L., and Venners, B. (2016).
Programming in Scala.
Artima Incorporation.

Scala-Collections-API (2020).
Scala immutable collections.
https://www.scala-lang.org/api/2.13.1/scala/collection/immutable/index.html.