Chapter 29
Further Language Constructs

29.1 Introduction

This chapter introduces a number of new concepts. The first is the use of *implicit*. Implicit facilities are a range of language facilities that provide *implicit* conversions between one type and another (typically in order to access some functionality or to provide compatibility between types). This chapter then introduces Scala annotations (a form of meta data for types), Enumerations and lazy evaluation.

29.2 Implicit Conversions

Explicit conversions are achieved programmatically using conversion methods and functions. An example of an explicit conversion is the way in which a string can be converted to an *Int* using the *toInt* function. For example:

```
package com.jeh.scala.convert

object TestStringConversion extends App {

  val s = "32"
  val i:Int = s.toInt
  println(i)

}
```

The val *s* holds a reference to a String containing the characters '3' and '2'. Using the *toInt* operation we can convert it into the integer 32, which is stored in the val 'i' (which we have explicitly declared as an Int although this is not actually necessary as Scala can infer the type of i). The result of this program is that the integer 32 is printed out.

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_29

Implicit conversions, by contrast, are conversions from one type to another type that happens automatically. Scala supports the idea of implicit conversions by looking for any available, appropriate conversion functions when it finds that the type made available does not match the type required. For example, if a function or method required an integer and the programmer provided a string, Scala would look to see if there was an appropriate conversion method available (within scope) that could be used to convert a string into an integer.

For a method or function to be used in this way it must be marked with the keyword *implicit*. This means that only those methods marked by the developer as being suitable to be used with an implicit conversion can be used in this way.

Implicit conversion methods and functions are a very useful way of extending the types that a library of code can work with. For example, you cannot extend the class Integer, but in Scala you can extend the concept of an Integer, for example, what if you want to be able to use a 'square' operator to square any integer value. There is no support in the current Integer class for this, but we can add it. This can be done by creating a new class, the IntOperations class, and providing an implicit converter that converts from a standard Integer into an IntOperations class. For example:

```
class IntOperations(originalValue: Int) {
  def square = originalValue * originalValue
}

object IntSquare {
  implicit def intToIntOperations (n: Int):
              IntOperations = new IntOperations(n)
}
```

The key here is that the intToIntOperations method is marked as being implicit and thus can be used when an implicit conversion is required.

At one level we have defined a new class that builds on an Integer and allows that Integer to be squared. It also provides a method that allows a string to be converted into an IntOperations instance. This code could be used as shown below:

```
object IntOperationsApp extends App {
  val res1 = (new IntOperations(5)).square
  println(res1)
}
```

This would result in the following output:

```
25
```

However, this does not meet the proposed aim described earlier. We have not extended the concept of an Integer. However, the following example appears to suggest that we have done exactly that:

```
object IntOperationsApp extends App {
  import IntSquare._
  val result = 5.square
  println(result)
}
```

In this case the Integer "5" does not have an operation 'square' defined for it. However, Scala looks to see if there is an implicit method available that can convert an Integer into a type that does support the 'square' operation. If the intToIntOperations method is in scope (e.g. because it has been imported), then Scala can use that method to convert the Integer '5' into a IntOperations instance and then invoke the '*' operator on it. As the return type of the '*' method is a String the val *result* holds a String. Thus it appears that Strings now support the '*' operator. The result of printing out result is again:

```
25
```

Implicit conversions can be very powerful; however, you should be careful how and when you use implicit conversions as there can be unintended consequences. It is therefore important to manage the visibility of your implicit conversion methods appropriately.

29.3 Implicit Parameters

An implicit parameter is a parameter to a method, function or constructor that is marked as implicit. Such a parameter does not need to have been provided by the caller. Instead, if an implicit parameter is missing when the method is invoked, Scala will attempt to provide an appropriate value. It does this by looking at the set of variables currently in scope and attempts to find one that has been marked as implicit and is of the right type (or that can be implicitly converted to the right type).

Note that this is not the same as providing a default value for a parameter, rather it provides a way for Scala to find suitable parameters from those that are currently in scope. That is, the compiler will search for an implicit value defined within the current scope (according to the resolution rules). Implicit parameters are very good for simplifying an API by providing values that can be imported and used when the programmer does not want (or need) to provide them.

To explore implicit parameters, we will use the following sample code:

```
object PrinterApp extends App {

  def printer(content: String)(implicit i:Int): Unit = {
    for (i <- 0 until i) print(content)
  }

  implicit val v=2

  printer("John")(4)
  println("\n----------------")
  printer("John")

}
```

The printer method defines two parameter lists with the second parameter list containing a single implicit parameter 'i':

The printer method can now be called within 1 or 2 arguments (as long as an implicit variable is available to provide the second parameter). Thus we can call the printer function in the following ways:

```
printer("John")(4)
println("\n----------------")
printer("John")
```

The first invocation of printer passes in the string "John" and the integer 4. The output of this is

```
JohnJohnJohnJohn
```

The second invocation again passes in the string "John", but Scala looks for an implicit argument to provide for the second parameter list. If none is available, a compilation error will be generated. In this case a definition of an implicit val integer 'v' is in scope:

```
implicit val v=2
```

Thus Scala uses the value in 'v' as the value to use for the second parameter which means that the second invocation of printer is the same as writing (printer ("John")(2)), with the result that the output generated is:

```
JohnJohn
```

Defining a *val* or *var* (or indeed a method that will supply an appropriate value) as *implicit* means that the value held will be considered during any *implicit resolution*. However, you should note that an explicit invocation would always override an implicit value.

There are some restrictions on implicit parameters, including:

- There can only be a single implicit keyword per method
- The implicit parameter must be at the start of a parameter list (although there can be multiple parameter lists)
- When the implicit keyword is used with a parameter list, then it makes all values of that parameter list implicit
- If there are multiple parameter lists, then only the last one may be implicit.

29.4 Implicit Objects

It is also possible to have a companion object that contains appropriate implicit values or implicit conversion method. Scala will search companion objects for implicit values or conversion methods/functions. This is a very useful feature and can be used for building implicit adapters that convert one type to another.

For example, in the following listing a trait LabelMaker has a companion object LabelMaker. This companion object defines two things. The first is an AddressLabelMaker inner object. This AddressLabelMaker is a subtype of the LabelMaker trait. It defines a method output that takes an Address instance and returns a string representing the Address as a Label. It is also marked as an *implicit* object and thus can be used whenever an implicit value is required.

```
case class Address(street: String, number: Int)

// Defines a trait to convert things into labels
trait LabelMaker[T] {
  def output(t: T): String
}

object LabelMaker {
  // Adapter class that converts Address to labels
  // and an instance created from it for use
  // with implicit params
  implicit object AddressLabelMaker
                  extends LabelMaker[Address] {
    def output(address: Address): String = {
      address.number + " @ " + address.street
    }
  }
  // label method that uses an implicit param
  def label[T](t: T)(implicit lm: LabelMaker[T]) =
    lm.output(t)

}
```

The other method defined by the companion object LabelMaker is *label*. This method is a generic method which defines the generic type 'T' as the parameter type for the single parameter in the first parameter list. It then specifies an implicit parameter in the second parameter list. This second parameter requires an instance of r object that is a type of LabelMaker. This label maker is used to convert the parameter 't' into a label. Any client of the LabelMaker can either call the printLabel using the two parameter lists, for example:

```
label(address)(myLabelMaker)
```

or they can call it with a single parameter list, for example:

```
label(address)
```

In this latter case, Scala will look within the current context to see if there is an implicit value that can be used for the required second parameter. If there is, it will use it; if there is not, then the code will not compile.

In the test application presented below, we are importing the LabelMaker definitions into the Test object. Note that we have used the import statement within the body of the Test object. This means that the properties, methods and functions defined on the LabelMaker concept will only be directly accessible within Test. Also note that the LabelMaker *concept* is comprised of the LabelMaker Trait and the LabelMaker *companion* object, and thus, the contents of both are imported at this point. The Test object creates an Address to use and then calls the *println* function passing in the result returned from the label method. Note that it is the single parameter list version that is being invoked here, and thus, Scala looks for an implicit value to use for the second parameter list for the LabelMaker. As the LabelMaker object has been imported it finds an implicit object AddressLabelMaker that it can use and thus the second parameter is bound to the AddressLabelMaker.

```
object LabelMakerApp extends App {

  import LabelMaker._

  val a = Address("High Street", 10)
  // method label uses Label maker to convert to labels
  // implicitly uses the object instance of
  // AddressLabelMaker
  println(label(a))

}
```

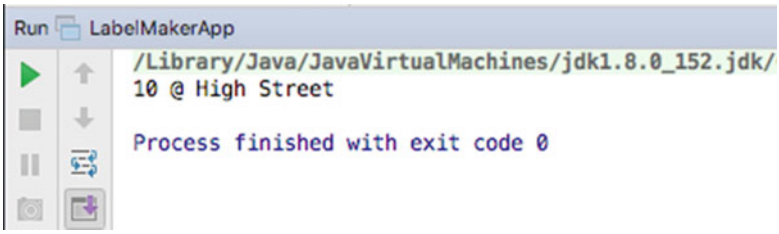

Fig. 29.1 Output from the LabelMaker test application

The result of executing this program is shown in Fig. 29.1.

29.5 Implicit Classes

Implicit classes (which were introduced in Scala 2.10) can be used to automatically provide a factory conversion function that will convert a given type into the implicit class type. That is, using the implicit keyword in front of the class definition causes Scala to create a factory method that will convert another type of object into an instance of the implicit class.

Classes annotated with the *implicit* keyword are referred to as implicit classes. There are a set of restrictions on implicit class that must be met, and these are:

1. All *implicit* classes must have a primary constructor with *exactly* one argument in its first parameter list (although it can have additional parameter lists that may include additional implicit parameters).
2. Implicit class cannot be top-level classes, and they must be contained within another type (but may be defined within an object, a trait or another class).
3. There cannot be any method, member or object in scope with the same name as the implicit class.

For example, in the following listing the implicit inner class RandomString makes its primary constructor available for implicit conversions. Thus if we need to convert a string into a RandomString in order to invoke the random method, then the RandomString will be invoked by creating a new instance of the RandomString based on the original string (which is passed in the value required by the single parameter constructor).

```
object Util {
  implicit class RandomString(value: String) {
    val size = value.length
    def random(n: Int): String = {
      import scala.util.Random
      (1 to n).map(x => value(Random.nextInt.abs %
                              size)).mkString
    }
  }
}
```

The compiler effectively converts this into the following:

```
object Util {
  class RandomString(value: String) {
    val size = value.length
    def random(n: Int): String = {
      import scala.util.Random
      (1 to n).map(x => value(Random.nextInt.abs % size)).mkString
    }
  }
  implicit final def RandomString(value: String):
          RandomString = new RandomString(value)
}
```

The above has the same behaviour as the implicit class—although the implicit class is both more concise and simpler to understand.

This Util class and its implicit inner class RandomString can be used in the following way:

```
object RamdomStringApp extends App {
  import Util._

  val str = "The_Great_Big_Scala_World"
  val randomString = str random 6
  println(s"'$randomString'")

}
```

Note that again although the Util object is imported at the top of the file the RandomString implicit class is only imported (without qualification) within the RamdomStringApp object. This means that the string "The_Great_Big_Scala_World" is implicitly converted into a RandomString using the RandomString constructor so that the random method can be invoked on that instance.

In this case the integer 6 is passed into the *random* method and a random string is generated from the contents of the original string. The end result is that the String

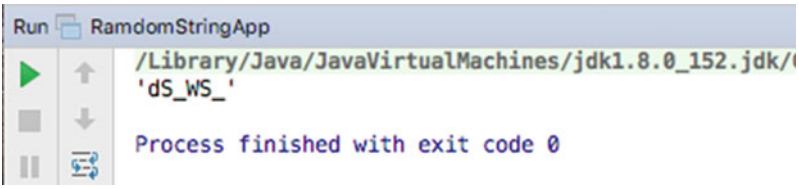

Fig. 29.2 Using the implicit class

"The_Great_Big_Scala_World" is used to generate a random string such as the string "dS_WS_" below, as illustrated in Fig. 29.2.

29.6 Scala Annotations

Annotations are meta data about program code; that is, they provide additional information about a class, object, trait, method, function, parameter, etc., to the compiler and to the runtime environment within which the code executes. Scala is not the only programming language to have annotations (or annotation like constructs). Both Java and C# support annotation style constructs.

In general annotations are used to provide:

- Directives to the compiler
- Generating additional material—Scaladoc
- Pretty printing
- Checking for common errors
- Information to third-party frameworks.

In Scala annotations can be applied to:

- vals, vars, defs, classes, objects, traits, types and expressions

Basic Scala annotations have the format:

```
@<annotation-name>
```

For example:

```
@deprecated
class Author {...}
```

Annotations with parameters have round brackets '(..)' and can take a comma separated list of expressions, for example:

```
@SerialVersionUID(1L)
```

Table 29.1 Annotations in Scala

Annotation	Meaning
scala.SerialVersionUID	It indicates serial version UID value. A serial version unique ID is used to determine whether a serialised instance is compatible with the class that will used when the instance is deserialised. If the number associated with the class is the same as that stored with the instance, then they are compatible. If they are not the same, then a serialisation exception is thrown
scala.serializable	This annotation designates the class to which it is applied as serializable
scala.cloneable	It indicates that the instance of the associated type can be cloned (copied)
scala.deprecated	It indicates that the associated type or construct is deprecated. This means that the type or construct should no longer be used and (typically) alternatives have been provided
scala.inline	It indicates that compiler should try to inline annotated method
scala.native	Type checking is present, but implementation will be native
scala.remote	It indicates can be accessed remotely
scala.throws	It indicates that a method throws a checked exception
scala.transient	It indicates field is transient
scala.reflect.BeanProperty	It adds setter and getter methods to a field
scala.volatile	This annotation allows programmers to use mutable state in concurrent programs
scala.unchecked	This annotation gets applied to a selector in a match expression. If present, exhaustiveness warnings for that expression will be suppressed

Currently there is a range of annotations that can be used as shown in Table 29.1.

The following listing illustrates a single class which has a number of annotations applied to it. Note that more than one annotation can be applied to a particular program element. In this case the class Person has two annotations applied, @SerialVersionUID and @deprecated. In addition the method toString has been marked with the @inline annotation.

```
package com.jjh.scala.anno

@SerialVersionUID(1L)
@deprecated
class Person(name : String, var age : Int) extends Serializable {
  @inline
  override def toString() = s"$name: $age"
}
```

29.7 Type Declarations

A type declaration can be used to create a new type or provide an alias to an existing type. It can also be used as a way of defining an abstract type that must be provided at a later date. It is particularly useful as a way of proving a more semantically meaningful name for a generic type. Types are defined using the keyword *type*; for example, a type can be defined based on an existing type as follows:

```
type T:=String
```

Alternatively a type declaration can be based on a function signature:

```
type F=Int=>String
```

which is a very useful way of specifying a meaningful name for such a signature. A type can also be an abstract declaration:

```
type T//actual type to be supplied in sub type
```

in which case the concrete type used with T must be provided in a subtype (for classes) or in the class (or object) that the trait is being mixed into.

An abstract type may also have a bound that is used to restrict the range of concrete types that can be used with it. For example:

```
type T<: Transport
```

Thus the abstract type 'T' has an upper bound of Transport. This means that the concrete type used in a class or object must either be of type Transport or a subtype of Transport.

Examples of the above are shown in the following listing:

```
class Transport

class Test {
  type T = String
  type F = Int => String
  type X                  // abstract type
  type Y <: Transport     // Upper Bounds
  type Z >: Transport     // Lower Bounds
}
```

29.8 Enumerations

There is a type Enumeration in Scala. This type is a trait that defines the core concepts of an Enumeration. An enumeration is a collection of entities that represent a complete, ordered set of all of the entities in that set. For example, the days of the week could be represented by an Enumeration as there is a finite set of days in the (working) week.

In Scala an enumerated set is an object that mixes in the trait Enumeration and defines the vals that will form the set of values that comprise that enumeration. For example, the days of the (working) week enumeration could be defined as follows:

```
package days

object DaysOfWeek extends Enumeration {
  val Monday = Value
  val Tuesday = Value
  val Wednesday = Value
  val Thursday = Value
  val Friday = Value
}
```

Note that each of the *val* in the enumeration is of type Value and is represented by a Value instance. Because all of the values in the enumeration are of the same type, a shorthand form can be used to define them, as follows:

```
object Weekend extends Enumeration {
  val Saturday, Sunday = Value
}
```

Here both Saturday and Sunday are values of the enumeration Weekend and both are represented by instances of Value.

The Value of an enumeration entry is actually an inner class of the Enumeration trait. The Value inner class:

- has an id—the value for this element of the enumeration. Note that enumerations are Zero based by default and thus the first element in an enumeration has the *value* 1.
- provides numerous methods +, <, equals, etc.
- defines a number of *implicit* methods.

Each call to a Value method adds a new unique value to the enumeration set where the id (or value) is incremented by one. Thus in the above example, Monday has id0, Tuesday the id1, Wednesday the id2, etc.

Once we have defined the enumeration it can be imported into other code using the import statement. Once it is imported then it can be used to provide values for variables, etc. For example:

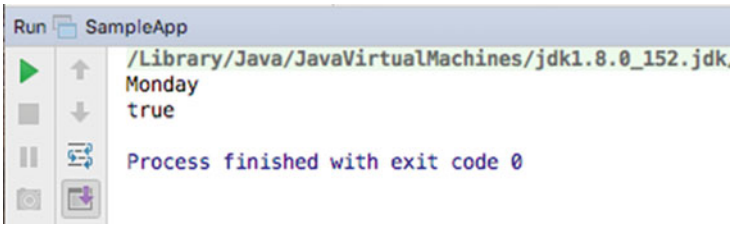

Fig. 29.3 Output from enumeration

```
import days.DaysOfWeek

object SampleApp extends App {
  val today = DaysOfWeek.Monday
  println(today)
  println(today < DaysOfWeek.Friday)
}
```

In the above code the DaysOfWeek enumeration has been imported and the enumeration element Monday has been used for the value of the *val* today. We then compare the value of today to see if it less than the value for Friday. As an enumeration is an ordered set there is an ordering to the element in the set and thus we can perform such comparisons. The result of executing this program is shown in Fig. 29.3. As you can see the value of today is printed as Monday, the string representation of an element of an enumeration is the name of the element (and not the underlying id).

In Scala is it also possible to import the values defined within an enumeration so that they can be used directly (without the need to prefix them with the name of the Enumeration). This can be done by directly importing the contents of the enumeration. For example:

```
import DaysOfWeek._
```

It is now possible to use the Enumeration values directly, for example:

```
object SampleApp extends App {
  import DaysOfWeek._
  val today = Monday
  println(today)
  println(today < Friday)
}
```

Note that we have imported the enumeration inside the definition of SampleApp, and thus, the enumerated values are only directly visible within the scope of SampleApp.

It is also possible to obtain all of the values in an Enumeration using the *values* property of the enumeration. For example, to print out all of the values in the DaysOfWeek enumeration we can:

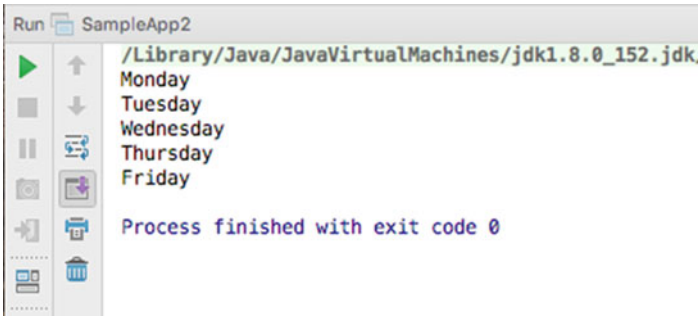

Fig. 29.4 Output from the Test2 program

```
object SampleApp2 extends App {
  DaysOfWeek.values.foreach(println(_))
}
```

This results in the output shown in Fig. 29.4.

If you wish you can of course get hold of the underlying id of each of the values in an Enumeration. This is done by accessing the readonly id property of any value within the enumeration. For example, to obtain the id of Monday we can:

```
object SampleApp3 extends App {
  val today = DaysOfWeek.Monday
  println(today.id)
}
```

This also means that it is possible to access a specific value of an enumeration using the id. Referencing the Enumeration and specifying the index to use as a parameter to a factory function that returns the appropriate value do this. For example:

```
object SampleApp4 extends App {
  val d = Weekend(0)
  println(d)
}
```

In the above code, we access the value *Saturday* using the index '0'.

It is also possible to define your own ids for the value in the enumeration (i.e. they do not need to start Zero and be incremented by one). For example, in the following listing the Direction enumeration contains four values for North, South, East and West with the degrees used to represent these compass points used as the *value* of each element in the enumeration. Note that the value to use as the id is used with the factory constructor for the Value inner class.

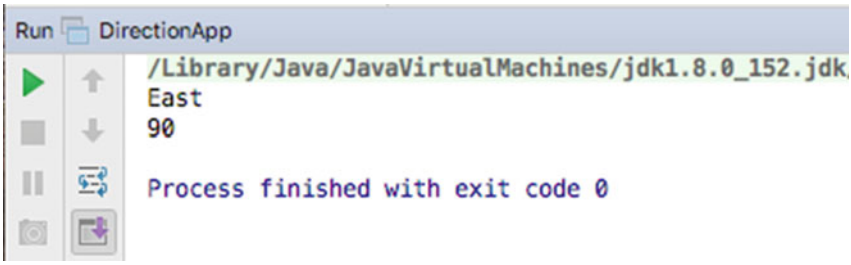

Fig. 29.5 Using a custom value

```
object Direction extends Enumeration {
  val North = Value(0)
  val East =  Value(90)
  val South = Value(180)
  val West =  Value(270)
}
```

As before it is possible to access the id of the value as well as the value itself:

```
object DirectionApp extends App {
  val d = Direction.East
  println(d)
  println(d.id)
}
```

The result of running this program is shown in Fig. 29.5. In this figure you can see that the result of printing the enumerated value is East but the id of this value is now 90.

Note that you can access the custom values in exactly the same way as you can access the default index values. You can therefore specify the enumerated element using values such as 0, 90, 180 and 270:

```
val d1 = Direction(90)
println(d1)
```

However, values are not restricted to integers as indexes. It is also possible to define values based on Strings. These are referred to as named values. In the following example the strings “n”, “e”, “s” and “e” are used to uniquely name (or identify) the individual members of the Enumeration.

```
object StringDirection extends Enumeration {
  val North = Value("n")
  val East =  Value("e")
  val South = Value("s")
  val West =  Value("e")
}
```

As before we can access the members of the enumeration using North, East, West and South or using their names such as “n”. However to do this we use an accessor function on the enumeration called *withName*. For example:

```
val d3 = StringDirection.withName("n")
println(d3)
```

One issue with the examples we have looked at so far is that the type of the members of the enumeration is Value. Thus if we wish to create properties or functions that work with the values of the enumeration the specified type will be Value. For example, the type of the parameter today is *Value*:

```
val today = DaysOfWeek.Monday
println(today)
```

Thus if we wrote a method to take this value and print it we would write:

```
def printDay(d: Value) = println(d)
```

While this is legal Scala it is not very meaningful to a human reader of the code. There is some semantic meaning that has been lost from the Enumeration to the inner Value. It is thus common to find that a recurring idiom is present within an Enumeration. This idiom defines a new type with a more meaningful alias for Value that can be used with clients of the enumeration. This type is defined within the scope of the Enumeration and thus does not pollute the namespace of a system. For example within the Enumeration *DaysOfWorkingWeek* we have defined a new type called Day that is an alias for Value. Monday, Tuesday, Wednesday, etc., are still instances of the inner class Value. However, we can refer to this type either through the name Value or the name Day.

Thus given the definition of the Enumeration *DaysOfWorkingWeek*:

```
object DaysOfWorkingWeek extends Enumeration {
 type Day=Value
 val Monday, Tuesday, Wednesday, Thursday, Friday=Value
}
```

We can now import the contents of the Enumeration and directly refer to the type Day as well as the values Monday, Tuesday, Wednesday, etc. Thus the parameter to the function *printDay* can now be Day (rather than Value):

```
import DaysOfWorkingWeek._
def printDay(d: Day)=println(d)
```

29.9 Lazy Evaluation

Scala possesses the ability to mark a *val* or *var* as lazy; for example, the following standard definition of *val* is evaluated immediately:

```
val x=15
```

However, you can use the prefix *lazy* to indicate that the *val* should only be evaluated the first time it is accessed:

```
lazy val x=15
```

This can be used to improve the performance of a system where some resource is expensive to create (either in terms of time and/or resources) but is only infrequently required.