

Chapter 17

Control and Iteration



17.1 Introduction

This chapter introduces control and iteration in Scala. In Scala, as in many other languages, the mainstay of the control and iteration processes is the `if` and `switch` statements and the `for` and `while` loops.

17.2 Control Structures

17.2.1 *The if Statement*

The basic format of an `if` statement in Scala is the same as that in languages such as C, Pascal and Java. A test is performed and, depending on the result of the test, a statement is performed. A set of statements to be executed can be grouped together in curly brackets `{ }`. For example,

```
if (a == 5)
  println("true")
else
  println("false")
if (a == 5) {
  print("a = 5")
  println("The answer is therefore true")
} else {
  print("a != 5")
  println("The answer is therefore false")
}
```

Of course, the `if` statement need not include the optional `else` construct:

```
if (a == 5) {
  print("a = 5")
  println("The answer is therefore true")
}
```

You must have a *Boolean* in a condition expression, so you cannot make the same equality mistake as in C. The following code always generates a compile time error:

```
if (a = 1) {
  ...
}
```

Unfortunately, assigning a Boolean to a variable results in a boolean (all expressions return a result) and thus the following code is legal, but does not result in the intended behaviour (the string “Hello” is always printed on the console):

```
var a = false
if (a = true)
  println("Hello")
```

You can construct nested `if` statements, as in any other language:

```
if (count < 100)
  if (index < 10)
    {...}
  else
    {...}
  else
    {...}
```

However, it is easy to get confused. Scala does not provide an explicit if-then-elseif-else type of structure. In some languages, you can write:

```
if (n < 10)
  print ("less than 10");
else if (n < 100)
  print ("greater than 10 but less than 100");
else if (n < 1000)
```

```
print ("greater than 100 but less then 1000");  
else  
print ("greater than 1000");
```

This code is intended to be read as laid out above. However if we write it in Scala, it should be laid out as below:

```
if (n < 10)  
  print ("less than 10")  
else if (n < 100)  
  print ("greater than 10 but less than 100")  
else if (n < 1000)  
  print ("> than 100 but < 1000")  
else  
  print ("> than 1000")
```

This code clearly has a very different meaning (although it may have the same effect). This can lead to the infamous “dangling else” problem. Another solution is the `switch` statement. However, as you will see the `switch` statement has significant limitations.

17.2.2 *If Returns a Value*

Almost all statements in Scala return a result, and the `if` statement is no different. This means that you can use an `if` statement to determine the value to assign to a value (or pass to a method, etc.). For example, the following code assigns either the string “Dad” or the string “No Data” to the value `role` defining the current string referenced by the variable name:

```
val role =  
  if (name == "John")  
    "Dad"  
  else  
    "No Data"
```

```
println(role)
```

This is a very useful feature of the `if` statement which can be used effectively in many situations.

17.3 Iteration

Iteration in Scala is accomplished using the `for`, `while` and `do-while` statements. Just like their counterparts in other languages, these statements repeat a sequence of instructions a given number of times.

17.3.1 For Loops

A `for` loop in Scala is very similar to a `for` loop in other languages. It is used to step a *variable* through a series of values until a given test is met. Many languages have a very simple `for` loop, for example,

```
for i = 0 to 10 do
  ...
endfor;
```

In this case a variable `I` would take the values 0, 1, 2, 3, etc., up to 10. The longhand form of this in Scala is:

```
for (i <- (0).to(10)) {
  print(i)
}
```

Note that in the above `to` is a method call on the `Int` (integer) type. In practice, this is a lower level implementation issue, and it would be far more common to write:

```
for (i <- 0 to 10) print(i)
```

This can be done as Scala can infer the brackets and the dot which has the benefit that it will look far more familiar as a language construct to those used to programming languages such as C and Pascal.

One thing to note is that the operator here includes the value 10, whereas in languages such as C and Java it would mean up to but not including 10.

Multiple indexes can be used with a `for` loop. For example, we could increment `I` from 1 to 3 and `j` from 5 to 7:

```
object MultipleForLoopTest extends App {
  for (i <- 1 to 3; j <- 5 to 7) {
    print("Value of i: " + i);
    println(" / Value of j: " + j);
  }
}
```

This may not have the effect you expect. This equates to loop the value of `I` through 1 to 3 for each of the values of `j`; thus, the output is:

```

Value of i: 1 / Value of j: 5
Value of i: 1 / Value of j: 6
Value of i: 1 / Value of j: 7
Value of i: 2 / Value of j: 5
Value of i: 2 / Value of j: 6
Value of i: 2 / Value of j: 7
Value of i: 3 / Value of j: 5
Value of i: 3 / Value of j: 6
Value of i: 3 / Value of j: 7

```

As you can see from this, the value of *i* remains constant for all values of *j* and is then incremented for a repeated for the values of *j*.

17.3.2 For Until

An alternative to the *to* operator is the *until* operator which indicates that a variable *i* should loop unto but not including the higher bound, thus:

```
for (i <- 1 until 4) println(i)
```

Producers the output:

```

1
2
3

```

But does not include the value 4.

17.3.3 For Loop with a Filter

Another option with the for loop is to include a filter into the looping process. This can be used to *filter* out those elements within a loop that you do not want to process. A filter is an additional logical test added to the for loop following the iteration values already presented. For example, assuming that the variable *files* contain some form of list of files, we can add an extra test to check so that we only print out files where the filename ends with “.txt”:

```
for (f <- files if f.getName.endsWith(".txt"))
  println(f)
```

With this loop each file in the list of files is tested such that the name is first obtained (*getName*) and then the string method *endsWith* tests to see if the filename ends with “.txt”; if it does, then the file is processed by the loop—which in this case involving printing out the file. If the filename does not end with “.txt,” then the loop immediately moves onto the name file in the list.

The complete program for this example is shown below:

```
import java.io.File

object FileLoopTest extends App {

  val files = (new File(".")).listFiles

  for (f <- files if f.getName.endsWith(".txt"))
    println(f)

}
```

Note that any number of if conditions can be added to provide multiple filters on a for loop. Each if condition is separated by a ‘;’, for example,

```
for (f <- files if f.getName.startsWith("Help"); if
f.getName.endsWith(".txt"))
  println(f)
```

17.3.4 Longhand for Loop

Although we have looked at a number of different for loops in the preceding sections, they are all subsets of the full for loop which is made up of a generator, an optional definition and a filter. Thus you could write:

```
for (
  p <- persons ; // a generator
  n = p.name ; // a definition
  if (n startsWith "To") // a filter
) println(n)
```

With the definition being reset each time round a loop.

17.3.5 For-Yield Loop

A special for loop is a for-yield loop. It is particularly useful for collecting together a set of results from a for loop that can be processed by other code rather than performing the processing directly within the for loop.

That is, the value of the all the previous for loops (from the point of view of the expression being evaluated when the for loop executes) is Unit or nothing. However, using a yield then each time around the loop a *yield* expression can be evaluated and the results of the expression are collected together and made available to subsequent lines of code once the for loop has completed.

The general syntax for a for-yield loop is:

```
for (sequence) yield expression
```

Examples of the use of the for-yield loop are shown below:

```
val data = for (i <- 1 to 5) yield 10 * i
```

produces a sequence of values (10, 20, 30, 40, 50) held in the val data

```
val info = for (n <- List("one", "two", "three")) yield n.substring(0, 2)
```

produces a list of values List(on, tw, th) held in info.

In both cases subsequent code could process either the data variable or the info variable. This is a very powerful construct for creating a collection of data items to be further processed from some loop-based operations.

17.3.6 While Loops

The while loop exists in almost all programming languages. In most cases, it has a basic form such as:

```
while (test expression)
  statement
```

This is also true for Scala. The while expression controls the execution of one or more statements. If more than one statement is to be executed, then the statements must be enclosed in curly brackets { }:

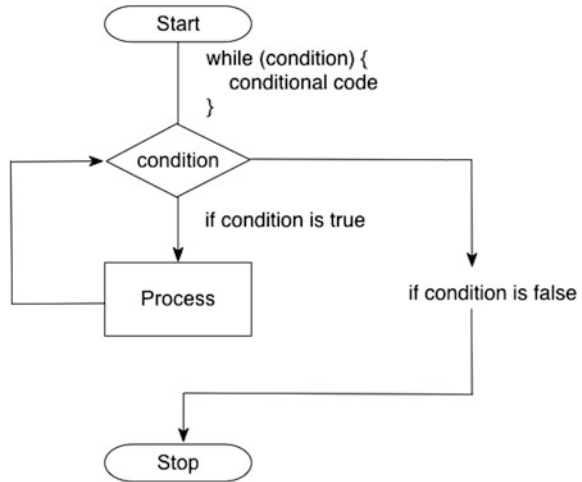
```
var i=0;
while (i < 10) {
  println(i)
  i += 1
}
```

The above loop tests to see if the value of *i* is less than or equal to 10, and then prints the current value of *i* before incrementing it by one. This is repeated until the test expression returns false (i.e. *i* = 11).

You must assign *i* an initial value before the condition expression. If you do not provide an initial value for *i*, it defaults to none value, and the comparison with a numeric value raises an exception.

The behaviour of the while loop is illustrated in Fig. 17.1.

Fig. 17.1 Behaviour of a while loop



17.3.7 Do Loops

In some cases, we want to execute the body of statements at least once; you can accomplish this with the `do` loop construct:

```

do
  statement
while (test expression);
  
```

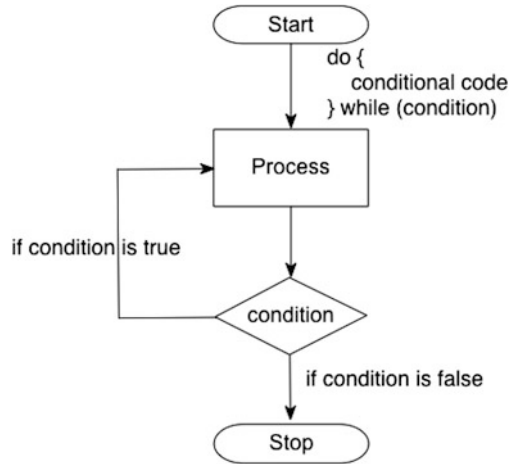
This loop is guaranteed to execute at least once, as the test is only performed after the statement has been evaluated. As with the `while` loop, the `do` loop repeats until the condition is false. You can repeat more than one statement by bracketing a series of statements into a block using curly brackets `{ }`:

```

var n = 10
do {
  println(n)
  n = n - 1
} while (n > 0)
  
```

The above `do` loop prints the numbers from 10 down to 1 and terminates when `n = 0`. The logic of the while loop is illustrated in Fig. 17.2.

Fig. 17.2 Behaviour of a do loop



17.3.8 An Example of Loops

As a concrete example of the `for` and `while` loops, consider the following class. It possesses a method that prints numbers from 0 to the `MaxValue` variable:

```

class Counter {
  var MaxValue = 10
  def count() = {
    var i = 0
    println("----- For -----");
    for (i <- 0 to MaxValue) {
      print(" " + i)
    }
    println(" ")
    println("----- While -----")
    i = 0
    while (i <= MaxValue) {
      print(" " + i)
      i = i + 1
    }
    println(" ")
    println("-----")
  }
}

object Counter extends App {
  val c = new Counter()
  c.count
}
  
```

The result of running this application will be:

```
-----For-----
0 1 2 3 4 5 6 7 8 9 10
-----While-----
0 1 2 3 4 5 6 7 8 9 10
-----
```

17.4 Equality

Two instances may be considered equivalent if their contents are the same. This equivalence is defined by the equals method on a class and is used by the ‘==’ and ‘!=’ operators, where:

- == tests for equality
- != tests for not equals

The equality operators are actually invoked on the left-hand operand with the right-hand operand being passed to the operator as a parameter.

You can compare two instances using == and != , for example,

- 1 == 2 // false
- 1 != 2 // true
- 1 == 1.0 // true
- List(1, 2, 3) == List(1, 2, 3) // true
- List(1, 2, 3) == "John" // false

Note that there is also a *referential* equality operator in Scala. This is provided by ‘eq’ method. This method tests that the two instances being compared are literally the same instance rather than just equivalent in value.

17.5 Recursion

Recursion is a very powerful programming idiom found in many languages. Scala is no exception. The following class illustrates how to use recursion to generate the factorial of a number:

```

class Factorial {
  def factorial(number: Int): Int = {
    println(number)
    if (number == 1)
      return 1
    else
      return { number + factorial(number - 1) }
  }
}

```

```

object FactorialTest extends App {
  val f = new Factorial()
  println("= " + f.factorial(5))
}

```

The result of running this application is illustrated here.

```

Run FactorialTest
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk
5
4
3
2
1
= 15
Process finished with exit code 0

```

One problem for recursion is that although it is elegant to program, it may not be the most efficient runtime solution. However, Scala can optimise tail-recursive methods such that it can be expressed via recursion in terms of the program but optimised into an iterative loop at runtime. Since Scala 2.8, you can now mark a method that you expect to use tail recursion with the annotation `@tailrec`. An annotation is a piece of metadata that the runtime can use to perform additional processing, etc. This allows you to indicate that the method should be optimisable for tail recursion by the compiler. It thus allows the compiler to provide a warning if the method does not succeed in being tail recursive. For example,

```

package com.jjh.scala.tail

import scala.annotation.tailrec

object Util {
  def factorial(n: Int): Int = {
    @tailrec
    def factorialAcc(acc: Int, n: Int): Int = {
      if (n <= 1) acc
      else factorialAcc(n * acc, n - 1)
    }
    factorialAcc(1, n)
  }
}

```

To understand why this makes a difference consider the following method:

```

package com.jeh.scala.tail

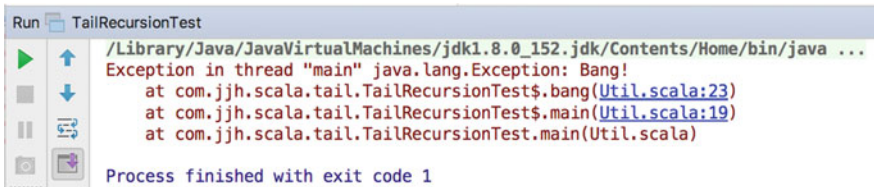
object TailRecursionTest {

  def main(args: Array[String]): Unit = {
    bang(4)
  }

  def bang(x: Int): Int = {
    if (x == 0) throw new Exception("Bang!")
    else bang(x - 1) // +1
  }
}

```

The method `bang` intentionally throws an `Exception` (causes an error to occur) when `x` is Zero. This allows you to see what the compiler has done with the runtime version of the code. With the `+1` element of the `else` part of the `if` statement commented out this is a tail-recursive method. When we run this program, the output is as shown below:

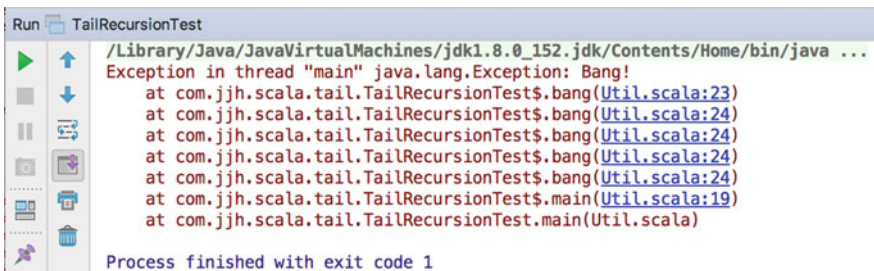


```

Run TailRecursionTest
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
Exception in thread "main" java.lang.Exception: Bang!
    at com.jjh.scala.tail.TailRecursionTest$.bang(Util.scala:23)
    at com.jjh.scala.tail.TailRecursionTest$.main(Util.scala:19)
    at com.jjh.scala.tail.TailRecursionTest.main(Util.scala)
Process finished with exit code 1

```

If we now uncomment the `+1` at the end of the `if` statement and rerun this program, we now get:



```

Run TailRecursionTest
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java ...
Exception in thread "main" java.lang.Exception: Bang!
    at com.jjh.scala.tail.TailRecursionTest$.bang(Util.scala:23)
    at com.jjh.scala.tail.TailRecursionTest$.bang(Util.scala:24)
    at com.jjh.scala.tail.TailRecursionTest$.bang(Util.scala:24)
    at com.jjh.scala.tail.TailRecursionTest$.bang(Util.scala:24)
    at com.jjh.scala.tail.TailRecursionTest$.bang(Util.scala:24)
    at com.jjh.scala.tail.TailRecursionTest$.main(Util.scala:19)
    at com.jjh.scala.tail.TailRecursionTest.main(Util.scala)
Process finished with exit code 1

```

What you can see is that the first version has been converted into an iterative loop which does not need to keep calling itself (which is inefficient at runtime).

However, with the second example we have called the same bang method multiple times which has resulted in the need to handle each call separated (set-up the call stack for each method invocation) which is far less efficient.

Thus knowing whether the recursive method is tail recursive or not is an important consideration.

17.5.1 The Match Expression

Scala's match expression allows for a selection to be made between a number of alternative tests and as such is similar in nature to the case statement in Pascal and C or the switch statement in Java. However, compared to the switch statement in Java allows much wider pattern matching capability in the case clause of the expression this provides for a far more powerful and flexible construct. Also note that the match expression is an expression (and not just a statement), thus it returns a value and can be used as part of an assignment clause.

The pattern element of the match expression is much more flexible than in languages such as C and Java and can be any one of the following:

- a literal,
- a wild card (to match anything),
- a type,
- a variable which will be populated,
- of different types,
- tuple patterns, etc.

As an example, consider the following simple literal match test:

```
object MatchTest extends App {
  val arg = "John"
  val relationship =
    arg match {
      case "John" => "Dad"
      case "Denise" => "Mum"
      case "Phoebe" => "Daughter"
      case "Adam" => "Son"
      // Default / wildcard
      case _ => "WhoAreYou?"
    }
  println(relationship)
}
```

This example compares the `arg` var1 with the String literals “John”, “Denise”, “Phoebe” and “Adam”. If it is one of these values, it returns the string associated with that literal. Thus if `arg` holds the string “John”, then the match expression will return the String “Dad”. The result of the match expression is then saved into the `relationship` var and printed out. If the value held in `arg` is not one of the strings explicitly mentioned then it will use the default (wild card) case test. Thus any other string in `arg` will result in “WhoAreYou?” being returned by the match expression. Note in match expression “_” is being used as a wild card that will match any string not mentioned above in the case tests.

However, unlike many other languages the literals used in the individual case tests do not need to be of the same type. For example, the following uses four different types of literal from an Int, to a Boolean, to a string and an empty list (Nil):

```
object MatchTest2 {

  def main(args: Array[String]): Unit = {
    println(describe(5))
    println(describe(true))
    println(describe("hello"))
    println(describe(Nil))
    println(describe(List(1, 2, 3)))
  }

  def describe(x: Any) = x match {
    case 5 => "five"
    case true => "truth"
    case "hello" => "welcome message"
    case Nil => "The empty list"
    case _ => "something else"
  }
}
```

The type of the parameter `x` is `Any` which is the root of everything in Scala, and thus `x` can indeed hold any type of value. We can now use the method `describe` to produce a printed string for any type of value in Scala—although unless it is the value 5, true, “hello” or Nil, it will print the description as “something else”.

A variable on the last example allows us to use a variable in the case test of the wild card to obtain the actual value passed in and to use that in the expression being evaluated:

```

object MatchTest3 {

  def main(args: Array[String]): Unit = {
    println(describe(5))
    println(describe(true))
    println(describe("hello"))
    println(describe(Nil))
    println(describe(List(1, 2, 3)))
  }

  def describe(x: Any) = x match {
    case 5 => "five"
    case true => "truth"
    case "hello" => "welcome message"
    case Nil => "The empty list"
    case somethingElseVariable => "something else: " +
somethingElseVariable
  }
}

```

Now when something other than the value 5, true, “hello” or Nil is passed in, then the result will be that the string “something else: “concatenated with that thing in string format will be returned by the described method.

We can also match by type rather than by specific value. For example, in the following example if the type of data passed in is a String, then the first case test will pass and the string will be *bound* to the variables and thus available for processing in the resultant operation associated with that test. In turn if the instance passed into `getSize` is a List, then it will pass the second test and be bound to the variable `l` and be available to the operation following this test. If the instance passed to `getSize` is actually a Map, then it will meet the criteria specified by the third case test. The result in each case is that an appropriate method is called to determine the size of the instance passed to `getSize`. If the instance is not a String, a List or a Map, then `-1` is returned by default:

```
object MatchTest4 {  
  
  def main(args: Array[String]): Unit = {  
    val name = "John"  
    println(getSize(name))  
    val xxx = List(1, 2, 3)  
    println(getSize(xxx))  
    val myMap = Map("London" -> "01", "Cardiff" -> "020")  
    println(getSize(myMap))  
    val otherMap = Map(1 -> "a", 2 -> "b", 3 -> "c")  
    println(getSize(otherMap))  
    val lectureMap =  
      Map("John" -> "Scala", "Fed" -> "PHP")  
    println(getSize(lectureMap))  
    val info = (1, 2, 3)  
    println(getSize(info))  
  }  
  
  def getSize(x: Any) = x match {  
    case s: String => s.length()  
    case l: List[_] => l.size  
    case m: Map[_,_] => m.size  
    case _ => -1  
  }  
}
```