Chapter 16
Scala Constructs

16.1 Introduction

This chapter presents more of the Scala language. It considers the representation and use of numbers, strings and characters. It also discusses assignments, literals and variables. Finally, it considers messages, message types and their precedence.

16.2 Numbers and Numeric Operators

16.2.1 Numeric Values

Just as in most programming languages, a numeric value in Scala is a series of numbers which may or may not have a preceding sign and may contain a decimal point:

```
25;   −10;   1996;   12.45;   0.13451345;   −3.14
```

Unusually for a programming language, Scala explicitly specifies the number of bytes that must be used for data types such as Short, Int, Long, Float and Double:
The Scala language designers' purpose in specifying the number of bytes to use for each data type was to enhance the portability of Scala implementations. In C, the number of bytes used for `int` and `long` is at the discretion of the compiler writers. The only constraint placed upon them is that `int` cannot be bigger than `long`. This means that a program that compiles successfully on one machine may prove unreliable and have errors when recompiled on another machine. This can make porting a program from one system to another extremely frustrating (ask anyone who has ever had to port a sizeable C system!) (Tables 16.1).

© Springer International Publishing AG 2018

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_16

Table 16.1 Standard numbers of bytes for numeric data types

Type	Bytes	Stores
Byte	1	Integers
Short	2	Integers
Int	4	Integers
Long	8	Integers
Float	4	32-bit IEEE 754 single-precision float
Double	8	64-bit IEEE 754 double-precision float

16.2.2 Arithmetic Operators

In general, the arithmetic operators available in Scala are the same as in any other language. There are also comparison functions and truncation functions (see Table 16.2). Numbers can also be represented by objects which are instances of classes such as `Integer`, `Float`. These classes are all subclasses of the class Number and provide different facilities. However, some of the methods are fairly common (Table 16.3).

A number of the numeric classes also provide class variables, such as `MAX_VALUE` and `MIN_VALUE` (i.e. in `Integer`, `Long`, `Double`, `Float`), and numbers such as `NEGATIVE_INFINITY` and `POSITIVE_INFINITY` (i.e. in `Double` and `Float`).

In addition, Scala provides a class called `Math`. This class, which is a subclass of `Object`, provides the usual range of mathematical operations (see Table 16.4). All these methods are class (or static) methods available from the class `Math`. You do not have to create an instance of the class to use them.

It is also interesting to notice that, to enhance the portability of Scala, the language designers have stated that the definitions of many of the numeric methods must produce the same results as a set of published algorithms.

Table 16.2 Basic numeric operators

+	Addition
−	Subtraction
*	Multiplication
/	Division
%	Remainder
==	Equality
<	Less than
>	Greater than
!=	Inequality
<=	Less than or equal to
>=	Greater than or equal to

Table 16.3 Methods provided by numeric classes

`equals()`	Equality
`doubleValue()`	Conversion
`toHexString()`	Conversion
`valueOf(aString)`	Conversion (class-side)
`toBinaryString()`	Conversion
`toOctalString()`	Conversion

Table 16.4 Mathematical functions provided by Math

max	Maximum
ceil	Round up
round	Round to nearest
abs	Absolute value
pow	Raises one number to the power of the other
min	Minimum
floor	Round down
sqrt	Square root
exp	Exponential
random	Random number generator

16.3 Characters and Strings

16.3.1 Characters

Characters in Scala are of type Char and are represented by 16-bit unsigned integers. In Scala, a single character is defined by surrounding it with single quotes:

```
'J'   'a'   '@'   '1'   '$'
```

16.3.2 Strings

`Strings` in Scala are represented by the (Java) class String and examples of a string are instances of this class. As such, they are made up of individual elements, similar to strings in C. However, this is the only similarity between strings in C and Scala. A Scala string is not terminated by a null character and should not be treated as an array of characters. It should be treated as an object which responds to an appropriate range of messages (e.g. for manipulating or extracting substrings) (Table 16.5).

A string is defined by one or more characters placed between double quotes (rather than the single quotes used for characters):

Table 16.5 Methods provided by the class String

charAt(index: Int)	Returns the character at position index
compareTo (anOtherString)	Compares two strings lexicographically
equals(String aString)	Compares two strings
equalsIgnoreCase (String aString)	Compares two strings, ignoring the case of the characters
indexOf (char aCharacter)	Returns the first index of the character in the receiving string
substring (int start, int stop)	Creates substring from start to stop (in the receiving string)
toLowerCase()	Returns the receiver in lower case letters
toUpperCase()	Returns the receiver in upper case letters

```
"John Hunt"   "Tuesday"   "dog"
```

You cannot create a string by generating an array of characters. This can be the source of much confusion and frustration when an apparently correct piece of code does not work. A string containing a single character is not equivalent to that single character:

```
'a' ! = "a"
```

The string `"a"` and the character `'a'` are, at best, instances of different classes and, at worst, one may be an instance and one a basic type. The fact that the string contains only one character is just a coincidence.

To denote that a variable should take an instance of String, define it as being of type String:

```
val aVariable: String = "John"
```

Of course due to type inference in most situations Scala can infer that the type of the variable should be String.

16.4 Assignments

A variable name can refer to different objects at different times. You can make *assignments* to a variable name, using the = operator. It is often read as “becomes equal to” (even though it is not preceded by a colon as in languages such as Ada).

Some examples of assignment statements follow:

```
currentEmployeeIndex = 1;
newIndex = oldIndex;
myName = "John Hunt";
```

Like all Scala operators, the assignment operator returns a value. The result of an assignment is the value of that assignment (thus the value of the expression `x = 2 + 2;` is 4). This means that several assignments can be made in the same statement:

```
nextObject = newObject = oldObject;
```

The above example also illustrates a feature of Scala style—variable names that indicate their contents. This technique is often used where a more meaningful name (such as `currentEmployeeIndex`) is not available (`temp` might be used in other languages).

Although variables in Scala are strongly typed, this typing is perhaps not as strong as in languages such as Pascal and Ada. You can state that a variable is of type `Any`. As `Any` is a class, such a variable can possess instances of the class Any or *one of its subclasses*! This means that a variable that holds a String may then be assigned a Person or a List (a type of data structure) instance. This is quite legitimate:

```
var temp: Any = new Person()
temp = "John"
temp = List(..)
```

An important point to note is that assignment is by reference when dealing with objects. This means that, in the following example, `nextObject`, `newObject` and `oldObject` all refer to the *same* object (as illustrated in Fig. 16.1)

```
newObject = oldObject = new Person(..)
nextObject = newObject;
```

As all three variables point to an instance of a class (in this case `Person`), if an update is made to the contents of any one of the properties maintained by the person (such as the age property), it is made for all three!

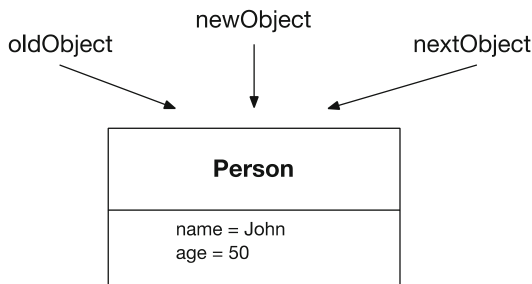

Fig. 16.1 Result of a multiple assignment

16.5 Variables

16.5.1 Temporary Variables

These variables exist only for the duration of some activity (e.g. the execution of a method). They can be defined anywhere within a method (as long as they are defined before they are used). The definition takes the form of the type (or class) of the variable and the variable name followed by any initialisation required:

```
var aChar: Char;
var anotherChar = 'a';
var anInstance: AnyRef;
var myName = "John Hunt";
```

Note all of these are written as vars but they could equally have been vals. The scope of a temporary variable depends on the context in which it is defined. For example, variables declared at the top level of a method are in scope from the point at which they are declared. However, block variables only have scope for the block within which they are defined (including nested blocks). Loop variables only have scope for the loop within which they are defined. Thus the scope of each of the following variables is different:

```
def add (a: Int, b: Int): Int = {
      val result = 0                   r
      for (i < - 0 to 5) {            ir
         if (a < i) {                 ir
         var total = b               tir
              total = total + c * i  tir
          }                           ir
      }                                r
return result                          r
}
```

In the right-hand column, `r` indicates that `result` is in scope, `i` indicates the scope of the loop variable and `t` indicates the scope of the inner block variable, `total`.

16.5.2 Pseudo-Variables

A pseudo-variable is a special variable, whose value is changed by the system, but which cannot be changed by the programmer. The value of a pseudo-variable is determined by the current context and can be referenced within a method.

`this` is a pseudo-variable that refers to the receiver of a message itself. The search for the corresponding method starts in the class of the receiver. To ensure that your source code does not become cluttered, Scala assumes you mean `this` object if you just issue a reference to a method. The following statements have the same effect:

```
this.myName()
myName()
```

You can use `this` to pass a reference to the current object to another object:

```
otherObject.addLink(this)
```

16.5.3 Variable Scope

Temporary variables are only available within the method in which they are defined. However, both class variables and instance variables are in scope (or are visible) at a number of levels. An instance variable can be defined to be visible (available) outside the class or the package, only within the package, within subclasses or only within the current class. The scope is specified by modifiers which precede the variable definition:

```
Public val myName = "John Hunt";
```

16.5.4 Option, Some and None

Sometimes what we need to represent is that a variable currently does not hold anything. The approach taken in Java was to represent such *values* as *null*. The idea was that the null value is an object that represents nothing or no object. It is not of any type nor it is an instance of any class (including `Object`). It really does means *nothing* or *no value*. However, this has lead to the now much discussed NullPointerException in Java which is generally considered now to be a weakness of the language.

The approach adopted within Scala is to use a type called an Option. An Option can hold any type or can be set to None. None indicates the absence of an actual value but is not the same as Null in Java.

For example, using Option you can indicate that a variable date should hold a Date type but currently a data has not been specified, for example:

```
val date: Option[Date]  = None
```

This declares that the val *date* is holding an Option wrapper, around an instance of Data but that currently this is initialised to None.

Such values can then be used within a match statement to perform one action if a value is present or another action if there is no value (or None), for example:

```
def printDate = date match {
  case Some(d) => print(d)
  case None => println("No Date")
}
```

Although a more idiomatic Scala approach would be to use the getOrElse method on Option which indicates that you should return the value held by an option or return some default value, for example:

```
def printDate2 = println(date getOrElse "No Date")
```

As a more concrete example of using an option consider the following class Event. This class represents some interesting event that has occurred within some system at some point in time.

```
case class Event(name: String,
                 date: Option[Date] = None,
                 state: String = "New") {
  def printDate = date match {
    case Some(d) => print(d)
    case None => println("No Date")
  }

  def printDate2 = println(date getOrElse "No Date")
}

object Event {
  implicit def apply(d: Date): Option[Date] =
                              Option(new Date())
}
```

When the data associated with the Event is printed via the printDate method where we either print the date or a string “No date”. Note that the companion object Event defines a utility conversion method that will take a date and convert it into an Option so that users of the class Event do not have to do this themselves. As the apply is marked as implicit, if the method is in scope, then when Scala is looking for a way to convert a Date into an option it can use this method automatically without the programmer explicitly specifying it.

A simple example of using this class is shown below:

```
object Test extends App {
  val e1 = Event("New Appointment")
  e1.printDate2

  val e2 = Event("New Appointment",
             Option(new Date()))
  e2.printDate2

  import Event._
  val e3 = Event("New Appointment", new Date())
  e3.printDate2
}
```

Note that the second Event created uses the implicit apply conversion method to convert the new instantiated Date into an option. The output from this application is

```
No Date
Tue Dec 19 17:19:37 GMT 2017
Tue Dec 19 17:19:37 GMT 2017
```

16.5.5 Boolean Values

In Scala there is a specific type used to represent truth or falsehood. This is the Boolean type. It has two values true and false which can be written as literals and can be assigned to variables and values and used in logical operations.

16.5.6 Literals

All of the preceding types can be written in literal form. That is 23 is a literal Int, 23.0 a literal Double, 'A' a Char and "John" a String literal. Scala also supports literals written using:

– Hexadecimal preceding the literal with Ox
– Octal preceding the literal with O5=
– Integer ending with L or l is a Long
– Character literals in " e.g. 'A'
– Character literal preceded by \u is a Unicode character, e.g. '\u0041'
– Symbol literal is 'aSymbol

16.6 Messages and Message Selectors

16.6.1 Invoking Methods

Invoking a method is often referred to as *sending a message* to the object that owns the method. The expression which invokes a method is composed of a receiving object (the receiver), the method name and Zero or more parameters. The combination of method name and parameters is often called the message and it indicates, to the class of the receiving object, which method to execute. Figure 16.2 illustrates the main components of a message expression.

The value of an expression is determined by the definition of the method it invokes. Some methods are defined as returning no value (e.g. `Unit`) while others may return a Value Type (such as Int) or instance. In the following code, the result returned by the method `marries` is saved into the variable `newStatus`:

```
newStatus = thisPerson.marries(thatPerson)
```

16.6.2 Precedence

The rules governing precedence in Scala are similar to those in other languages. Precedence refers to the order in which operators are evaluated in an expression. Many languages, such as C, explicitly specify the order of evaluation of expressions such as the following:

```
2 + 5 * 3 − 4 / 2;
```

Scala is no exception. The rules regarding precedence are summarised in Table 16.6. The above expression would be evaluated as:

```
(2 + (5 * 3)) − (4 / 2);
```

Notice that if operators with the same precedence are encountered they are evaluated strictly from left to right.

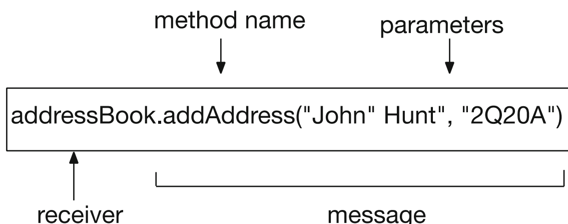

Fig. 16.2 Components of a message expression

Table 16.6 Operator precedence

Operation	Meaning	Precedence
x++ −−x	Prefix increment/decrement	16
x++ x−−	Postfix increment/decrement	15
– ! ~	Arithmetic negation/logical not/flip	14
(typename)	Cast (type conversion)	13
* / %	Multiplication/division/remainder	12
+ −	Addition/subtraction	11
≪ ≫ >>>	Left and right bitwise operators	10
< > <= >=	Relational operators	9
== !=	Equality operators	8
&	Bitwise and	7
^	Bitwise exclusive or	6
|	Bitwise or	5
&&	Conditional and	4
||	Conditional or	3
? :	Conditional operators	2
=	Assignment operator	1

16.7 Summary

In this chapter and the previous, you have learnt about classes in Scala, how they are defined, how instance variables are specified and how methods are constructed. You have also encountered many of the basic Scala language structures.