Chapter 8
Scala Building Blocks

8.1 Introduction

This chapter presents an introduction to the Scala programming language. As such, it is not intended to be a comprehensive guide. It introduces the basic elements of the Scala language, including Apps, discusses the concept of classes and instances and how they are defined, presents methods and method definitions and considers constructors and their role.

8.2 Apps and Applications

In the previous chapter we wrote a simple object that possessed a `main` method. This `main` method was the starting point or entry point for an application. That is, unless we are writing software that will be controlled by something else, such as a Web application server or a code library, every application needs a starting point. This is provided by the `main` method. In the last chapter this `main` method was defined in the object Hello as shown below:

```
object Hello {
  def main(args: Array[String]): Unit = {
    println("Hello Scala")
  }
}
```

This method is called *main*, takes one parameter (the data passed into it) and this parameter must be of type `Array` of Strings. This means that a sequence of strings can be made available to the program. It returns `Unit` (which indicates that it does not return anything at all). Any return values must be managed via an explicit call to a special object called sys (e.g. `sys.exit(0)`). This is because in a long-running application the value to be returned may not be available in the original main method.

However, you cannot change the definition of the main method—it must be as shown, i.e.

```
def main(args: Array[String]): Unit = {...}
```

This is because the underlying JVM expects there to be a main method with this signature available.

However, from Scala's point of view this is boilerplate code—that is it is always the same and in general in Scala where something can be inferred by Scala let Scala do that and we can simplify our code. In this case you must remember to provide a parameter to the method main even if you never expect to pass any data into your program! You must also remember (or learn) the syntax for array in Scala—which we will not be looking at for a while. Actually, the name of the parameter (*args* in this case) is not fixed but by convention is called *args*.

To simplify this issue Scala provides a way of avoiding the need to write the main method. If the object that will be used as the entry point to your application extends a trait called App then any code not placed into any named function or method (main is an example of a named method) will be assumed to be part of the main method. We could therefore rewrite the Hello object from above as:

```
class Hello extends App {
  println("Hello World")
}
```

This is clearly simpler to write and remember than having to include the def main … element.

It should be noted that if you search Scala applications on the Web you may well find a reference to a trait called Application. This plays a similar role to App but was replaced by App in Scala 2.9 (App is more efficient as it is implemented in a different way to Application).

It should also be noted that if you include the App trait (using the keyword extends) you cannot also write the main method—you can choose one or other approach but not both.

You may be wondering at this point about the term *trait* that has been used to describe both App and Application—for the moment just treat this as some functionality that can be incorporated into or mixed into an object (or indeed a class). We will return to *traits* again later in the book.

8.3 The Basics of the Language

All Scala programmers make extensive use of the existing Scala libraries even when they write relatively trivial code. For example, the following version of the "Hello World" program reuses existing classes rather than just using the language (for the moment do not worry too much about the syntax of the definition or use of key-words such as `extends`—this just allows us to mix in the `App` trait):

```
object HelloJohn extends App {
  val myName = "John Hunt"
  if (myName.endsWith("Hunt")) {
    println("Hello " + myName)
  } else {
    println("Hello World")
  }
}
```

In this example, I have reused the `String` class to represent the string "John Hunt" and to find a substring in it using the message `endsWith()`. Some of you may say that there is nothing unusual in this and that many languages have string handling extensions. However, in this case, it is the String contained within `myName` which decides how to handle the `endsWith` message and thus whether it contains the substring "Hunt". That is, the data itself handles the processing of the string! What is printed to standard output (i.e. the Console) thus depends on which object receives the message. These features illustrate the extent to which existing classes are reused: you cannot help but reuse existing code in Scala, and you do so by the very act of programming.

As well as possessing objects and classes, Scala also possesses an inheritance mechanism. This feature separates Scala (and languages such as Java and C#) from object-based languages, such as Ada, which does not possess inheritance. For example, in the simple program above, I reuse the class Any (the root of all classes in Scala) and the class `HelloJohn` automatically inherits all the features of `Any`.

Inheritance is very important in Scala. It promotes the reuse of classes and enables the explicit representation of abstract concepts that can then be turned into concrete concepts.

8.3.1 Some Terminology

We now recap some of the terminology introduced so far in this book, explaining it with reference to Scala.

In Scala programs, actions or operations are performed by passing *messages* to and from instances. An instance (the *sender* of the message) uses a message to request that some behaviour (referred to in Scala as a *method* or indeed a *function*) be performed by another instance (the *receiver* of the message). Just as procedure calls can contain parameters, so can messages.

Scala is a strongly typed language; however, the typing relates to the class of an object (or the trait a class mixes in—we will return to this later) rather than its specific type. Thus, by saying that a method can take a parameter of a particular type, you actually mean that any instance of that class (or one of its subclasses) can be passed into that method.

8.3.2 The Message Passing Mechanism

The Scala message passing mechanism is somewhat like a procedure call in a non-Object-Oriented language:

- The point of control moves to the receiver; the object sending a message is suspended until it receives a response.
- The receiver of a message is not determined when the code is created (at *compile time*); it is identified when the message is sent (at *runtime*).

This *dynamic* (or *late*) binding mechanism is the feature that gives Scala its polymorphic capabilities (see Chap. 1 for a discussion of polymorphism).

8.3.3 The Statement Terminator

In Scala, although a semicolon can be used as a statement terminator, the majority of statements do not need an explicit statement termination (as this can be inferred by the compiler) and thus for the majority of your code a semicolon is not required at the end of a statement. Therefore, the following are equivalent:

```
println("Hello");
println("World");
```

And

```
println("Hello")
println("World")
```

Generally, in Scala, the latter style is preferred.