Chapter 6
Scala Background

6.1 Introduction

I first encountered the Scala language in 2010. I was working on a graduate training programme for an international banking organisation when I was asked to give an hour talk to the graduates on Scala. At that point, I had heard the name mentioned but had no idea what it was. I therefore did some reading, installed the IDE being used and tried out some examples—and was hooked.

Since then I have trained a wide range of people in Scala, used it to develop large commercial systems and written a book on Scala and Design Patterns. I am still hooked on it and find myself discovering new aspects to the language and the environment on almost every Scala project I am involved with.

6.2 The Class Person

The following classes define a simple class `Person` that has a first name and a last name and an age (these examples were first considered in the introduction). A `Person` is constructed by providing the first and last names, and their age and setters and getters are provided for each.

Here is the Java class:

```
class Person {
    private String firstName;
    private String lastName;
    private int   age;
    public Person(String firstName,
                  String lastName,
                  int age) {
```


J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_6

```
        this.firstName = firstName;
        this.lastName  = lastName;
        this.age  = age;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
     }
    public void String getFirstName() {
        return this.firstName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
     }
    public void String getLastName() {
        return this.lastName;
     }
     public void setAge(int age) {
         this.age = age;
     }
     public void int getAge() {
        return this.age;
     }
}
```

And here is the equivalent Scala class:

```
class Person(
    var firstName: String,
    var lastName: String,
    var age: Int)
```

Certainly the Java class is longer than the Scala class, but look at the Java class "How many times have you written something like that?" Most of this Java class is boilerplate code. In fact, it is so common that tools such as Eclipse allow us to create the boilerplate code automatically which may mean that we do not have to type much more in the Java case than the Scala case. However, when I look back and have to read this code I may have to wade through a lot of such boilerplate code in order to find the actual functionality of interest. In the Scala case this boilerplate code is handled by the language meaning that I can focus on what the class actually does.

Actually the Object-Oriented side of Scala is both more sophisticated than that in either Java or C# and also different in nature. For example, many people have found the distinction between the *static* side of a class and the instance side of a class confusing. Scala does away with this distinction by not including the static concept.

Instead it allows the user to define singleton objects, if these singleton objects have the same name as a class and are in the same source file as the class, then they are referred to as companion objects. Companion objects then have a special relationship with the class that allows them to access the internals of a class (private fields and methods) and can provide the Scala equivalent of static behaviour.

The class hierarchy in Scala is based on single inheritance of classes but allows multiple traits to be mixed into any given class. A Trait is a structure within the Scala language that is neither a class nor an interface (note Scala does not have interfaces even though it compiles to Java Byte Codes). It can however, be combined with classes to create new types of classes and objects. As such a Trait can contain data, behaviour, functions, type declarations, abstract members, etc. but cannot be instantiated itself.

The analogy might be that a class is like a flavour of ice cream. You can have vanilla as the basic flavour with all the characteristics of ice cream; chocolate could be a subclass of Vanilla which extends the concept to a chocolate flavour of ice cream. Separately we could have bowls containing chocolate chips, mint chips, M&Ms, sprinkles of various types. We can combine the vanilla ice cream with the mint chips to create vanilla mint chip ice cream. This provides a new type of ice cream but those mint choc chips are not in and of themselves an ice cream. Traits are like the mint chocolate chips, while classes are like the ice cream.

6.3 Functional Programming

So much for the Object-Oriented view of Scala, what about this functional programming concept? For those of you coming from a Java background this may seem a bit alien; however, functional programming languages have a long history from LISP developed in the late 1950s to more recent functional languages such as ML and Haskell.

Working with functions is *not* that difficult although until you become familiar with the syntax they may seem unwieldy—but the key is to hang in there and keep trying.

The following provides a simple example of a function literal in Scala that takes two numbers and adds them together:

$$\text{val add} = (\text{a: Int}, \text{b: Int}) = > \text{a} + \text{b}$$

This defines a new function that takes two integers in the parameters a and b and returns the result of adding a to b. The function can be accessed via the variable add. This is a variable of type Function. We can invoke this function as the following:

$$\text{add}(4, 5)$$

which should return the value 9. In Scala we can then partially apply this function. This means that we can bind one of the parameters to a value to create a new function that only takes one parameter; for example,

$$\text{val addTwo} = (2, \_ : \text{Int})$$

This function, addTwo, now adds 2 to whatever integer is passed to it, for example,

$$\text{addTwo}(5)$$

will return 7.

6.4 A Hybrid Language

If all Scala did was provide the ability to program functionally all that would do is provide yet another functional programming language. However, it is the fact that Scala mixes the two paradigms that allow us to create software solutions that are both concise and expressive.

The Object-Oriented paradigm has been such a success because it can be used to model concepts and entities within problem domains. When this is combined with the ability to treat functions as first-class entities we obtain a very powerful combination.

For example, we can now create classes that will hold data (including other objects) and define behaviours in terms of methods but which can easily and naturally be given functions that can be applied to the members of that object.

```
val numbers = List(1, 2, 3, 4, 5)
println(numbers)
```

In this case I have created a list of integers (note that this is a list of Integers as the type has been inferred by Scala) that are stored in the variable `numbers`.

```
val filtered = numbers.filter((n: Int) => n < 3)
println(filtered)
```

I have then applied a function to each of the elements of the list. This function is an anonymous function that takes an `Int` (and stores that `Int` in the variable `n`). It then tests to see if the value of `n` is less than `3`. If it is it returns `true` otherwise it

returns `false`. The method `filter` uses the function passed to it to determine whether the value passed it should be included in the result or not. This means that the variable `filtered` will hold a list of integers where each integer is less than the value `3`. Again note that this is again a `List` of `Ints` as once again Scala has inferred the type.

The output from these statements I shown below:

```
List(1, 2, 3, 4, 5)
List(1, 2)
```