SCALA PROGRAMMING WORKBOOK

Practice Sheets for Scala Programming

2016-2023

NUS-ISS

National University of Singapore, 25 Heng Mui Keng Terrace, Singapore 119615

Contents

Scala Programming Fundamentals

1. Introduction

This introductory practise are meant to get you up and running with Spark quickly with Scala Programming Language. Using the learning virtual machine image provided, we'll cover the details of available operations and distributed execution. You'll get a tour of the higher-level Scala language Facilities. We hope that this gives you the tools to quickly tackle data analysis problems, whether you do so on one machine or hundreds.

2. Programming in Scala

Scala is one of the hottest modern programming languages. It is the Cadillac of programming languages. It is not only powerful but also a beautiful language. Spark is written in Scala, but it supports multiple programming languages, including Scala, Java, Python, and R.

Scala is a great language for developing big data applications. It provides a number of benefits. First, a developer can achieve a significant productivity jump by using Scala. Second, it helps developers write robust code with reduced bugs. Third, Spark is written in Scala, so Scala is a natural fit for developing Spark applications.

This tutorial introduces Scala as a general-purpose programming language. Our goal is not to make you an expert in Scala, but to help you learn enough Scala to understand and write Spark applications in Scala.

With the preceding goal in mind, this tutorial covers the fundamentals of programming in Scala. To effectively use Scala, it is important to known functional programming, so functional programming is introduced first. The tutorial wraps up with a sample standalone Scala application.

3. Functional Programming (FP)

Functional programming is a programming style that uses functions as a building block and avoids mutable variables, loops, and other imperative control structures. It treats computation as an evaluation of mathematical functions, where the output of a function depends only on the arguments to the function. A program is composed of such functions. In addition, functions are first-class citizens in a functional programming language.

First, functional programming provides a tremendous boost in developer productivity. It enables you to solve a problem with fewer lines of code compared to imperative languages. Second, functional programming makes it easier to write concurrent or multithreaded applications. The ability to write multi-threaded applications has become very important with the advent of multi-CPU or multi-core computers. Third, functional programming helps you to write robust code. It helps you avoid common programming errors. Finally, functional programming languages make it easier to write elegant code, which is easy to read, understand, and reason about. A properly written functional code looks beautiful; it is not complex or messy. You get immense joy and satisfaction from your code.

a) FP is composed of 'Functions'

A *function* is a block of executable code. Functions enable a programmer to split a large program into smaller manageable pieces. In functional programming, an application is built entirely by assembling functions. Functional programming languages treat functions as a first-class citizen. In addition, in functional programming, functions are composable and do not have side effects. The following paragraphs detail the nature of functions.

i) First-Class Citizens:

Functional Programs treats functions as their first-class citizens. A function has the same status as a variable or value. It allows a function to be used just like a variable. Imperative languages treat variables and functions differently. For example, C does not allow a function to be defined inside another function. It does not allow a function to be passed as an input parameter to another function.

Functional Programs allow a function to be passed as an input to another function. It allows a function to be returned as a return value from another function. A function can be defined anywhere, including inside another function. It can be defined as an unnamed function literal just like a string literal and passed as an input to a function.

ii) Composable

Functions in functional programming are composable. Function composition is a mathematical and computer science concept of combining simple functions to create a complex one. For example, two composable functions can be combined to create a third function. Consider the following two mathematical functions:

```
f(x) = x*2
g(x) = x+2
```

The function f takes a numerical input and returns twice the value of the input as output. The function g also takes a numerical input and returns two plus that number as output. A new function can be composed using f and g, as follows:

```
h(x) = f(g(x)) = f(x+2) = ((x+2)*2)
```

Using the function h is same as first calling function g with the input given to h, and then calling function f with the output from function g.

Function composability is a useful technique for solving a complex problem by breaking it into a bunch of simpler sub-problems. Functions can then be written for each sub-problem and assembled together in a top-level function.

iii) No Side Effects

A function in functional programming does not have side effects. The result returned by a function depends only on the input arguments to the function. The behaviour of a function does not change with time. It returns the same output every time for a given input, no matter how many times it is called. In other words, a function does not have a state. It does not depend on or update any global variable.

Functions with no side effects provide a number of benefits. First, they can be composed in any order. Second, it is easy to reason about the code. Third, it is easier to write multi-threaded applications with such functions.

iv) Simple

Functions in functional programming are simple. A function consists of a few lines of code and it does only one thing. A simple function is easy to reason about. Thus, it enables robustness and high quality.

Simple composable functions make it easy to implement complex algorithms with confidence. FP encourages recursively splitting a problem into sub-problems until you can solve a sub-problem with a simple function. Then, the simple functions can be assembled to form a function that solves a complex problem.

b) FP has Immutable Data Structures

Functional programming emphasizes the usage of immutable data structures. A purely functional program does not use any mutable data structure or variable. In other words, data is never modified in place, unlike in imperative programming languages such as C/C++, Java, and Python.

Immutable data structures provide a number of benefits. First, they reduce bugs. It is easy to reason about code written with immutable data structures. In addition, functional languages provide constructs that allow a compiler to enforce immutability. Thus, many bugs are caught at compile time. Second, immutable data structures make it easier to write multi-threaded applications. Writing an application that utilizes all the cores is not an easy task. Race conditions and data corruption are common problems with multi-threaded applications. Usage of immutable data structures helps avoid these problems.

c) FP is an Expression Language

In functional programming, every statement is an expression that returns a value. For example, the if-else control structure in Scala is an expression that returns a value. This behaviour is different from imperative languages, where you can just group a bunch of statements within if-else. This feature is useful for writing applications without mutable variables.

4. Scala Fundamentals

Scala is a hybrid programming language that supports both object-oriented and functional programming. It supports functional programming concepts such as immutable data structures and functions as first-class citizens. For object-oriented programming, it supports concepts such as class, object, and trait. It also supports encapsulation, inheritance, polymorphism, and other important object-oriented concepts. Scala is a statically typed language. A Scala application is compiled by the Scala compiler. It is a type-safe language and the Scala compiler enforces type safety at compile time. This helps reduce the number of bugs in an application.

Scala is a Java virtual machine (JVM)–based language. The Scala compiler compiles a Scala application into Java bytecode, which will run on any JVM. At the bytecode level, a Scala application is indistinguishable from a Java application. Since Scala is JVM-based, it is seamlessly interoperable with Java. A Scala library can be easily used from a Java application. More importantly, a Scala application can use any Java library without any wrapper or glue code. Thus, Scala applications benefit from the vast library of existing Java code that people have developed over the last two decades.

Although Scala is a hybrid object-oriented and functional programming language, it emphasizes functional programming. That is what makes it a powerful language. To reap greater benefit Scala must be used as a functional programming language than as an object-oriented programming language.

Scala is a powerful language. With power comes complexity. Thus it is advisable not to try to learn all the language features at once. In this further sections we will learn only the fundamental constructs needed to write a Spark application.

5. Getting Started

The best way to learn a programming language is to program in it. You are able to better understand the material presented in this chapter if you play with the code samples as you read.

a) Shell Terminal

The easiest way to get started with Scala is by using the Scala interpreter, which provides an interactive shell for writing Scala code. It is a REPL (read, evaluate, print, loop) tool. When you type an expression in the Scala shell, it evaluates that expression, prints the result on the console, and waits for the next expression. Installing the interactive Scala shell is as simple as downloading the Scala binaries and un-packaging it. The Scala shell is called scala. It is located in the bin directory.

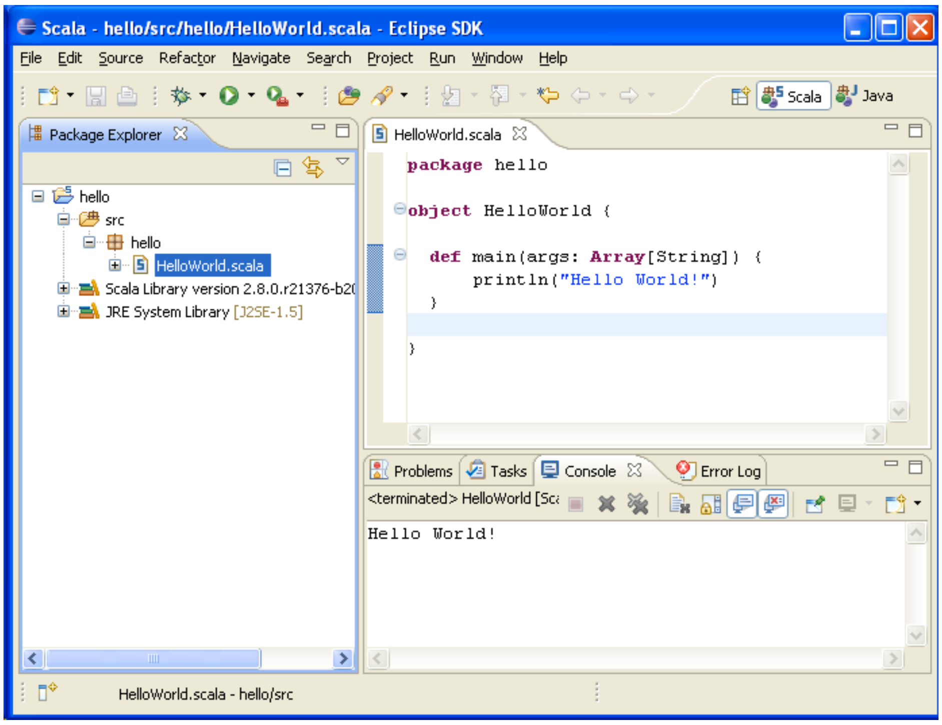
b) Using the Eclipse IDE

Here's a short tutorial how to create a "Hello World" application with Eclipse. It assumes that you have installed the plugin as described above and switched to the Scala perspective (*Window → Open Perspective → Other → Scala* then click *"OK"*).

1. Create a new Scala project "hello"
    - First click the "*New → Scala Project*" item in the "*File*" menu.
    - Enter "hello" in the "*Project name*" field.
    - Click the "*Finish*" button.
2. Create a new Scala package in source folder "src"
    - Right-click on the "hello" project in the "*Package Explorer*" tab of the projects pane.
    - Select the "*New → Package*" menu item.
    - in the "*New Package*" window, enter "hello" in the "*Name*" field.
    - Press the "*Finish*" button.
3. Create a Scala object HelloWorld with a main method
    - First expand the "hello" project tree and right-click on the "hello" package

- Select the "*New → Scala Object*" menu item and enter HelloWorld in the "*Object name*" field.
- Press the "*Finish*" button.

4. Extend the code to print a message
    - Make the HelloWorld object implement the main method and add a println statement (see image, below).
5. Create a Run configuration for the Scala project
    - Right click on HelloWorld.scala in the Package Explorer,
    - Select "*Run as ...*" and select "*Scala Application*"
    - Select the first matching item, the "*HelloWorld*" class.
    - Click the "*Ok*" button.


Scala - hello/src/hello/HelloWorld.scala - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Scala
Java
Package Explorer
hello
src
hello
HelloWorld.scala
Scala Library version 2.8.0.r21376-b20
JRE System Library [J2SE-1.5]
HelloWorld.scala
Problems
Tasks
Console
Error Log
<terminated> HelloWorld [Sca
HelloWorld.scala - hello/src

```
package hello

object HelloWorld {

  def main(args: Array[String]) {
      println("Hello World!")
  }

}
```

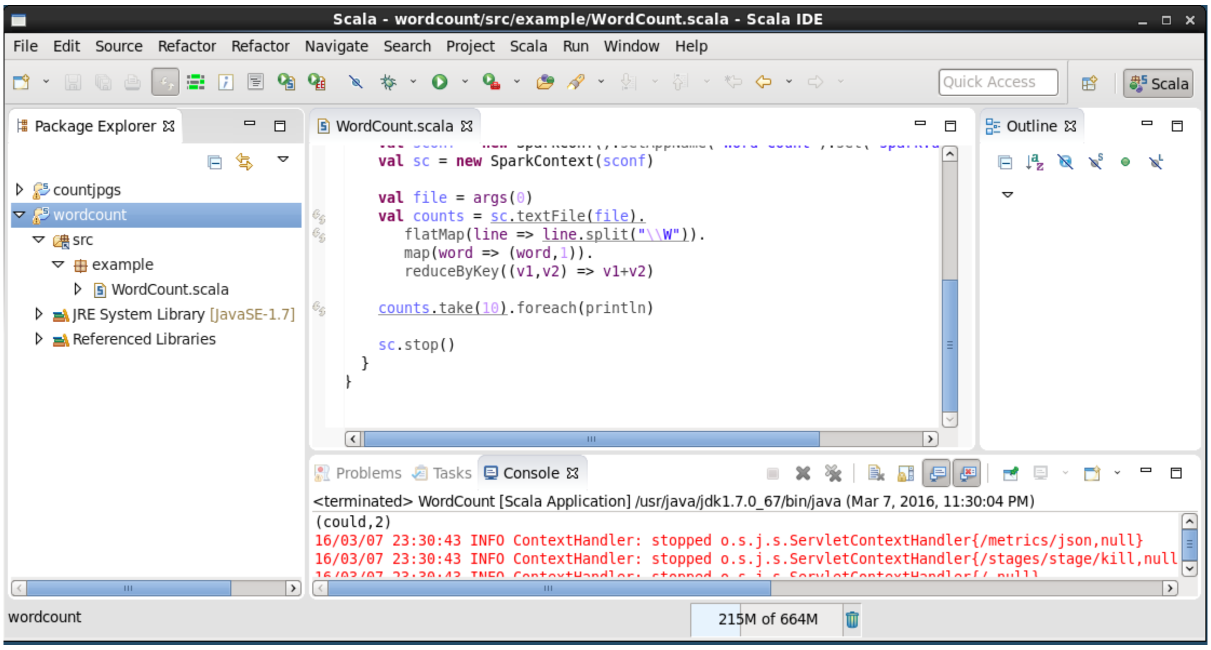
```
Hello World!
```

c) Creating Suitable Configurations Using the Eclipse IDE

6. Right click on the scala object having main method
7. Choose Run As -> Run Configurations
8. Double click on 'Scala Application' from Run Configurations window
9. Type the Object Name in the Main Class: (for eg: if HelloWorld.scala is the object having main method, give it as HelloWorld)

d) Executing Scala Code Using the Eclipse IDE

10. Select the "Run as... Scala application" menu item appears.
11. You may also want to pass values for args[. . .] using the Run Configurations as shown in the screen shots overleaf


Scala - wordcount/src/example/WordCount.scala - Scala IDE
File Edit Source Refactor Refactor Navigate Search Project Scala Run Window Help
Quick Access
Scala
Package Explorer
countjpgs
wordcount
src
example
WordCount.scala
JRE System Library [JavaSE-1.7]
Referenced Libraries
WordCount.scala
Outline
Problems Tasks Console
wordcount
215M of 664M

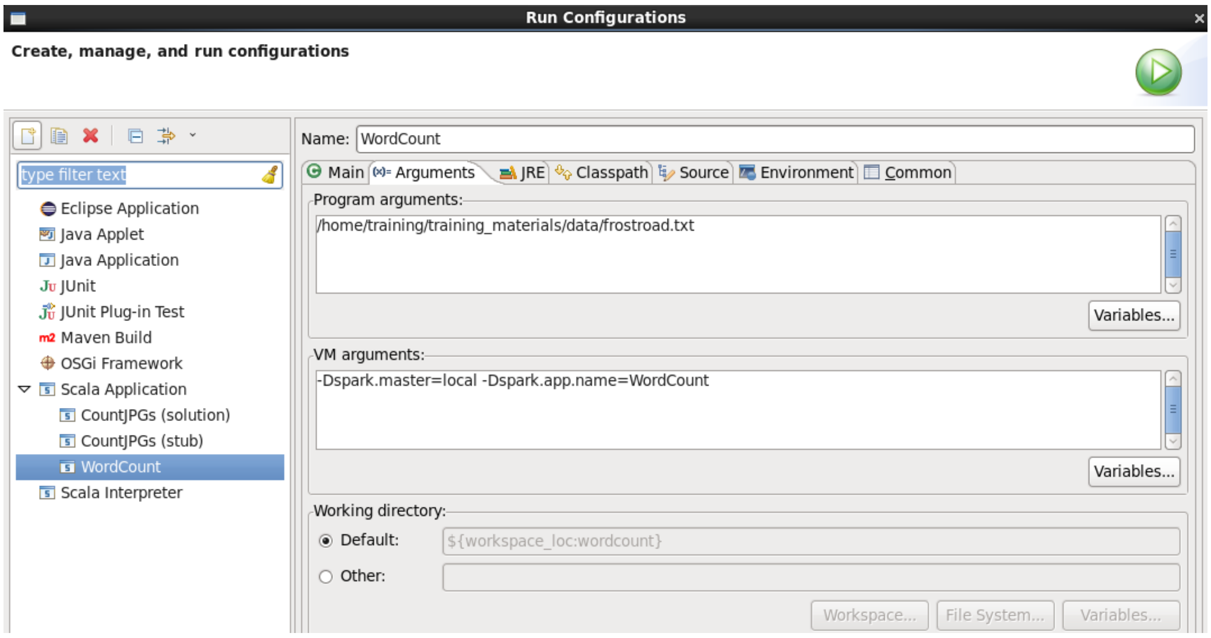
```
    val sc = new SparkContext(sconf)

    val file = args(0)
    val counts = sc.textFile(file).
       flatMap(line => line.split("\\W")).
       map(word => (word,1)).
       reduceByKey((v1,v2) => v1+v2)

    counts.take(10).foreach(println)

    sc.stop()
  }
}
```

```
<terminated> WordCount [Scala Application] /usr/java/jdk1.7.0_67/bin/java (Mar 7, 2016, 11:30:04 PM)
(could,2)
16/03/07 23:30:43 INFO ContextHandler: stopped o.s.j.s.ServletContextHandler{/metrics/json,null}
16/03/07 23:30:43 INFO ContextHandler: stopped o.s.j.s.ServletContextHandler{/stages/stage/kill,null
```



Name: WordCount

Program arguments:

/home/training/training_materials/data/frostroad.txt

VM arguments:

-Dspark.master=local -Dspark.app.name=WordCount

Working directory:

(●) Default: ${workspace_loc:wordcount}

( ) Other:

More Documentation is available in http://scala-ide.org/docs/user/gettingstarted.html

6. A Standalone Scala Application

So far, you have seen just snippets of Scala code. In this section, you will write a simple yet complete standalone Scala application that you can compile and run.

A standalone Scala application needs to have a singleton object with a method called *main*. This *main* method takes an input of type *Array[String]* and does not return any value. It is the entry point of a Scala application. The singleton object containing the main method can be named anything.

A Scala application that prints "Hello World!" is shown next.

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
      println("Hello World!")
```

```
        }
    }
```

You can put the preceding code in a file, and compile and run it. Scala source code files have the extension *.scala*. It is not required, but recommended to name a Scala source file after the class or object defined in that file. For example, you would put the preceding code in a file named *HelloWorld.scala*.

7. Introductory Scala Programming

In this section, you'll get an overview of Scala 's basic types, variables and functions.

a) Basic Types

Scala comes pre-packaged with a list of basic types and operations allowed on those types.

Value type	Range
Byte	8-bit signed two's complement integer (-2^7 to 2^7 - 1, inclusive)
Short	16-bit signed two's complement integer (-2^{15} to 2^{15} - 1, inclusive)
Int	32-bit signed two's complement integer (-2^{31} to 2^{31} - 1, inclusive)
Long	64-bit signed two's complement integer (-2^{63} to 2^{63} - 1, inclusive)
Char	16-bit unsigned Unicode character (0 to 2^{16} - 1, inclusive)
String	a sequence of Chars
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false

Collectively, types Byte, Short, Int, Long, and Char are called integral types. The integral types plus Float and Double are called numeric types. Other than String, which resides in package java.lang, all of the types shown in previous table are members of package scala. Note that Scala does not have primitive types. Each type in Scala is implemented as a class. When a Scala application is compiled to Java bytecode, the compiler automatically converts the Scala types to Java's primitive types wherever possible to optimize application performance.

b) Variables

Scala has two types of variables: mutable and immutable. Usage of mutable variables is highly discouraged. A pure functional program would never use a mutable variable. However, sometimes usage of mutable variables may result in less complex code, so Scala supports mutable variables too. It should be used with caution.

A mutable variable is declared using the keyword *var*; whereas an immutable variable is declared using the keyword *val*. A *var* is similar to a variable in imperative languages such as C/C++ and Java. It can be reassigned after it has been created. The syntax for creating and modifying a variable is shown next.

```
var x = 10
x = 20
```

A *val* cannot be reassigned after it has been initialized. The syntax for creating a val is shown next.

```
val y = 10
```

If you try to change the value (example: `y = 20`), the compiler will generate an error. It is important to point out a few conveniences that the Scala compiler provides. First, semicolons at the end of a statement are optional. Second, the compiler infers type wherever possible. Scala is a statically typed language, so everything has a type. However, the Scala compiler does not force a developer to declare the type of something if it can infer it. Thus, coding in Scala requires less typing and the code looks less verbose. The following two statements are equivalent.

```
val y: Int = 10;
val y = 10
```

c) Functions

As mentioned previously, a function is a block of executable code that returns a value. It is conceptually similar to a function in mathematics; it takes inputs and returns an output.

Scala treats functions as first-class citizens. A function can be used like a variable. It can be passed as an input to another function. It can be defined as an unnamed function literal, like a string literal. It can be assigned to a variable. It can be defined inside another function. It can be returned as an output from another function.

A function in Scala is defined with the keyword *def*. A function definition starts with the function name, which is followed by the comma-separated input parameters in parentheses along with their types. The closing parenthesis is followed by a colon, function output type, equal sign, and the function body in optional curly braces. An example is shown next.

```
def add(firstInput: Int, secondInput: Int): Int = {
  val sum = firstInput + secondInput
  return sum
}
```

In the preceding example, the name of the function is *add*. It takes two input parameters, both of which are of type *Int*. It returns a value, also of type *Int*. This function simply adds its two input parameters and returns the sum as output.

Scala allows a concise version of the same function, as shown next.

```
def add(firstInput: Int, secondInput: Int) = firstInput + secondInput
```

The second version does the exact same thing as the first version. The type of the returned data is omitted since the compiler can infer it from the code. However, it is recommended not to omit the return type of a function. The curly braces are also omitted in this version. They are required only if a function body consists of more than one statement.

In addition, the keyword *return* is omitted since it is optional. Everything in Scala is an expression that returns a value. The result of the last expression, represented by the last statement, in a function body becomes the return value of that function.

The preceding code snippet is just one example of how Scala allows you to write concise code. It eliminates boilerplate code. Thus, it improves code readability and maintainability.

8. Types of Functions

Scala supports different types of functions. Let's discuss them next.

a) Methods

A method is a function that is a member of an object. It is defined like and works the same as a function. The only difference is that a method has access to all the fields of the object to which it belongs.

b) Local Functions

A function defined inside another function or method is called a local function. It has access to the variables and input parameters of the enclosing function. A local function is visible only within the function in which it is defined. This is a useful feature that allows you to group statements within a function without polluting your application's namespace.

c) Higher-Order Methods

A method that takes a function as an input parameter is called a higher-order method. Similarly, a high-order function is a function that takes another function as input. Higher-order methods and functions help reduce code duplication. In addition, they help you write concise code.

The following example shows a simple higher-order function.

```
def encode(n: Int, f: (Int) => Long): Long = {
  val x = n * 10
  f(x)
}
```

The `encode` function takes two input parameters and returns a `Long` value. The first input type is an `Int`. The second input is a function `f` that takes an `Int` as input and returns a `Long`. The body of the `encode` function multiplies the first input by 10 and then calls the function that it received as an input.

You will see more examples of higher-order methods when Scala collections are discussed.

d) Function Literals

A function literal is an unnamed or anonymous function in source code. It can be used in an application just like a string literal. It can be passed as an input to a higher-order method or function. It can also be assigned to a variable.

A function literal is defined with input parameters in parenthesis, followed by a right arrow and the body of the function. The body of a functional literal is enclosed in optional curly braces. An example is shown next.

```
(x: Int) => {
              x + 100
            }
```

If the function body consists of a single statement, the curly braces can be omitted. A concise version of the same function literal is shown next.

```
(x: Int) => x + 100
```

The higher-order function encode defined earlier can be used with a function literal, as shown next.

```
val code = encode(10, (x: Int) => x + 100)
```

e) Closures

The body of a function literal typically uses only input parameters and local variables defined within the function literal. However, Scala allows a function literal to use a variable from its environment. A closure is a function literal that uses a non-local non-parameter variable captured from its environment. Sometimes people use the terms *function literal* and *closure* interchangeably, but technically, they are not the same. The following code shows an example of a closure.

```
def encodeWithSeed(num: Int, seed: Int): Long = {
  def encode(x: Int, func: (Int) => Int): Long = {
    val y = x + 1000
    func(y)
   }
   val result = encode(num, (n: Int) => (n * seed))
   result
}
```

In the preceding code, the local function *encode* takes a function as its second parameter. The function literal passed to encode uses two variables *n* and *seed*. The variable *n* was passed to it as a parameter; however, *seed* is not passed as parameter. The function literal passed to the *encode* function captures the variable *seed* from its environment and uses it.

9. Classes

A class is an object-oriented programming concept. It provides a higher-level programming abstraction. At a very basic level, it is a code organization technique that allows you to bundle data and all of its operations together. Conceptually, it represents an entity with properties and behaviour.

A class in Scala is similar to that in other object-oriented languages. It consists of fields and methods. A field is a variable, which is used to store data. A method contains executable code. It is a function defined inside a class. A method has access to all the fields of a class.

A class is a template or blueprint for creating objects at runtime. An object is an instance of a class. A class is defined in source code, whereas an object exists at runtime. A class is defined using the keyword *class*. A class definition starts with the class name, followed by comma-separated class parameters in parentheses, and then fields and methods enclosed in curly braces. An example is shown next.

```
class Car(mk: String, ml: String, cr: String) {
  val make = mk
  val model = ml
  var color = cr
  def repaint(newColor: String) = {
    color = newColor
  }
}
```

An instance of a class is created using the keyword new.

```
val mustang = new Car("Ford", "Mustang", "Red")
val corvette = new Car("GM", "Corvette", "Black")
```

A class is generally used as a mutable data structure. An object has a state, which changes with time. Therefore, a class may have fields that are defined using `var`. Since Scala runs on the JVM, you do not need to explicitly delete an object. The Java garbage collector automatically removes objects that are no longer in use.

10. Case Classes

A case class is a class with a case modifier. An example is shown next.

```
case class Message(from: String, to: String, content: String)
```

Scala provides a few syntactic conveniences to a case class.

First, it creates a factory method with the same name. Therefore, you can create an instance of a case class without using the keyword `new`. For example, the following code is valid.

```
val request = Message("harry", "sam", "fight")
```

Second, all input parameters specified in the definition of a case class implicitly get a `val` prefix. In other words, Scala treats the case class `Message` as if it was defined, as shown next.

```
class Message(val from: String, val to: String, val content: String)
```

The `val` prefix in front of a class parameter turns it into a non-mutable class field. It becomes accessible from outside the class.

Third, Scala adds these four methods to a case class: `toString, hashCode, equals`, and `copy`. These methods make it easy to use a case class.

Case classes are useful for creating non-mutable objects. In addition, they support pattern matching, which is described next.

11. Pattern Matching

Pattern matching is a Scala concept that looks similar to a switch statement in other languages. However, it is a more powerful tool than a switch statement. It is like a Swiss Army knife that can be used for solving a number of different problems.

One simple use of pattern matching is as a replacement for a multi-level if-else statement. An if-else statement with more than two branches becomes harder to read. Use of pattern matching in such situations improves code readability.

As an example, consider a simple function that takes as an input parameter a string representing a color and returns 1 if the input string is “Red”, 2 for “Blue”, 3 for “Green”, 4 for “Yellow”, and 0 for any other color.

```
def colorToNumber(color: String): Int => {
  val num = color match {
              case "Red" => 1
              case "Blue" => 2
              case "Green" => 3
              case "Yellow" => 4
              case _ => 0
            }
  num
}
```

Instead of the keyword *switch*, Scala uses the keyword *match*. Each possible match is preceded by the keyword *case*. If there is a match for a case, the code on the right-hand side of the right arrow is executed. The underscore represents the default case. If none of the prior cases match, the code for the default case is executed.

The function shown earlier is a simple example, but it helps illustrate a few important things about pattern matching. First, once a match is found, only the code for the matched case is executed. Unlike a *switch* statement, a *break* statement is not required after the code for each case. Code execution does not fall through the remaining cases.

Second, the code on the right-hand side of each right arrow is an expression returning a value. Therefore, a pattern-matching statement itself is an expression returning a value. The following example illustrates this point better.

```
def f(x: Int, y: Int, operator: String): Double = {
  operator match {
    case "+" => x + y
    case "-" => x - y
    case "*" => x * y
    case "/" => x / y.toDouble
  }
}
val sum = f(10,20, "+")
val product = f(10, 20, "*")
```

12. Traits

A *trait* represents an interface supported by a hierarchy of related classes. It is an abstraction mechanism that helps development of modular, reusable, and extensible code.

Conceptually, an interface is defined by a set of methods. An interface in Java only includes method signatures. Every class that inherits an interface must provide an implementation of the interface methods.

Scala traits are similar to Java interfaces. However, unlike a Java interface, a Scala trait can include implementation of a method. In addition, a Scala trait can include fields. A class can reuse the fields and methods implemented in a trait.

A trait looks similar to an abstract class. Both can contain fields and methods. The key difference is that a class can inherit from only one class, but it can inherit from any number of traits.

An example of a trait is shown next.

```
trait Shape {
   def area(): Int
}
class Square(length: Int) extends Shape {
   def area = length * length
}
class Rectangle(length: Int, width: Int) extends Shape {
   def area = length * width
}
val square = new Square(10)
val area = square.area
```

13. Tuples

A *tuple* is a container for storing two or more elements of different types. It is immutable; it cannot be modified after it has been created. It has a lightweight syntax, as shown next.

```
val twoElements = ("10", true)
val threeElements =  ("10", "harry", true)
```

A tuple is useful in situations where you want to group non-related elements. If the elements are of the same type, you can use a collection, such as an array or a list. If the elements are of different types, but related, you can store them as fields in a class. However, a class may be overkill in some situations. For example, you may have a function that returns more than one value. A tuple may be more appropriate in such cases.

An element in a tuple has a one-based index. The following code sample shows the syntax for accessing elements in a tuple.

```
val first = threeElements._1
val second = threeElements._2
val third = threeElements._3
```

14. Option Type

An Option is a data type that indicates the presence or absence of some data. It represents optional values. It can be an instance of eisther a case class called `Some` or singleton object called None. An instance of `Some` can store data of any type. The `None` object represents absence of data.

The Option data type is used with a function or method that optionally returns a value. It returns either `Some(x)`, where `x` is the actual returned value, or the `None` object, which represents a missing value. The optional value returned by a function can be read using pattern matching. The following code shows sample usage.

```
def colorCode(color: String): Option[Int] = {
  color match {
    case "red" => Some(1)
    case "blue" => Some(2)
    case "green" => Some(3)
    case _ => None
  }
}
val code = colorCode("orange")
code match {
  case Some(c) => println("code for orange is: " + c)
  case None => println("code not defined for orange")
}
```

The Option data type helps prevent null pointer exceptions. In many languages, null is used to represent absence of data. For example, a function in C/C++ or Java may be defined to return an integer. However, a programmer may return null if no valid integer can be returned for a given input. If the caller does not check for null and blindly uses the returned value, the program will crash. The combination of strong type checking and the Option type in Scala prevent these kinds of errors.

15. Collections

A collection is a container data structure. It contains zero or more elements. Collections provide a higher-level abstraction for working with data. They enable declarative programming. With an easy-to-use interface, they eliminate the need to manually iterate or loop through all the elements.

Scala has a rich collections library that includes collections of many different types. In addition, all the collections expose the same interface. As a result, once you become familiar with one Scala collection, you can easily use other collection types.

Scala collections can be broadly grouped into three categories: sequences, sets, and maps. This section introduces the commonly used Scala collections.

a) Sequences

A *sequence* represents a collection of elements in a specific order. Since the elements have a defined order, they can be accessed by their position in a collection. For example, you can ask for the *nth element* in a sequence.

i) Array

An *Array* is an indexed sequence of elements. All the elements in an array are of the same type. It is a mutable data structure; you can update an element in an array. However, you cannot add an element to an array after it has been created. It has a fixed length.

A Scala array is similar to an array in other languages. You can efficiently access any element in an array in constant time. Elements in an array have a zero-based index. To get or update an element, you specify its index in parenthesis. An example is shown next.

```
val arr = Array(10, 20, 30, 40)
arr(0) = 50
val first = arr(0)
```

Basic operations on an array include

- Fetching an element by its index
- Updating an element using its index

ii) List

A *List* is a linear sequence of elements of the same type. It is a recursive data structure, unlike an array, which is a flat data structure. In addition, unlike an array, it is an immutable data structure; it cannot be modified after it has been created. List is one of the most commonly used data structures in Scala and other functional languages.

Although an element in a list can be accessed by its index, it is not an efficient data structure for accessing elements by their indices. Access time is proportional to the position of an element in a list.

The following code shows a few ways to create a list.

```
val xs = List(10,20,30,40)
val ys = (1 to 100).toList
val zs = someArray.toList
```

Basic operations on a list include

- Fetching the first element. For this operation, the `List` class provides a method named `head`.
- Fetching all the elements except the first element. For this operation, the `List` class provides a method named `tail`.
- Checking whether a list is empty. For this operation, the `List` class provides a method named `isEmpty`, which returns true if a list is empty.

iii) Vector

The Vector class is a hybrid of the List and Array classes. It combines the performance characteristics of both Array and List. It provides constant-time indexed access and constant-time linear access. It allows both fast random access and fast functional updates.

An example is shown next.

```
val v1 = Vector(0, 10, 20, 30, 40)
val v2 = v1 :+ 50
val v3 = v2 :+ 60
val v4 = v3(4)
val v5 = v3(5)
```

b) Sets

Set is an unordered collection of distinct elements. It does not contain duplicates. In addition, you cannot access an element by its index, since it does not have one. An example of a set is shown next.

```
val fruits = Set("apple", "orange", "pear", "banana")
```

Sets support two basic operations.

- `contains`: Returns true if a set contains the element passed as input to this method.
- `isEmpty`: Returns true if a set is empty.

c) Map

Map is a collection of key-value pairs. In other languages, it known as a dictionary, associative array, or hash map. It is an efficient data structure for looking up a value by its key. It should not be confused with the map in Hadoop MapReduce. That map refers to an operation on a collection. The following code snippet shows how to create and use a Map.

```
val capitals = Map("USA" -> "Washington D.C.", "UK" -> "London", "India" -> "New Delhi")
val indiaCapital = capitals("India")
```

Scala supports a large number of collection types. Covering all of them is out of the scope for this book. However, a good understanding of the ones covered in this section will be enough to start productively using Scala.

16. Higher-Order Methods on Collection Classes

The real power of Scala collections comes from their higher-order methods. A higher-order method takes a function as its input parameter. It is important to note that a higher-order method does not mutate a collection.

This section discusses some of the most commonly used higher methods. The List collection is used in the examples, but all Scala collections support these higher-order methods.

a) map

The *map* method of a Scala collection applies its input function to all the elements in the collection and returns another collection. The returned collection has the exact same number of elements as the collection on which *map* was called. However, the elements in the returned collection need not be of the same type as that in the original collection. An example is shown next.

```
val xs = List(1, 2, 3, 4)
val ys = xs.map((x: Int) => x * 10.0)
```

It should be noted that in the preceding example, *xs* is of type *List[Int]*; whereas *ys* is of type *List[Double]*.

If a function takes a single argument, opening and closing parentheses can be replaced with opening and closing curly braces, respectively. The two statements shown next are equivalent.

```
val ys = xs.map((x: Int) => x * 10.0)
val ys = xs.map{(x: Int) => x * 10.0}
```

As mentioned earlier in this chapter, Scala allows you to call any method using operator notation. To further improve readability, the preceding code can also be written as follows.

```
val ys = xs map {(x: Int) => x * 10.0}
```

Scala can infer the type of the parameter passed to a function literal from the type of a collection, so you can omit the parameter type. The two following two statements are equivalent.

```
val ys = xs map {(x: Int) => x * 10.0}
val ys = xs map {x => x * 10.0}
```

If an input to a function literal is used only once in its body, the right arrow and left-hand side of the right arrow can be dropped from a function literal. You can write just the body of the function literal. The following two statements are equivalent.

```
val ys = xs map {x => x * 10.0}
val ys = xs map {_ * 10.0}
```

The underscore character represents an input to the function literal passed to the map method. The preceding code can be read as multiplying each element in the collection *xs* by 10.

To summarize, the following code sample shows both the verbose and concise version of the same statement.

```
val ys = xs.map((x: Int) => x * 10.0)
val ys = xs map {_ * 10.0}
```

As you can see, Scala makes it easier to write concise code, which is sometimes also easier to read.

b) flatMap

The *flatMap* method of a Scala collection is similar to *map*. It takes a function as input, applies it to each element in a collection, and returns another collection as a result. However, the function passed to *flatMap* generates a collection for each element in the original collection. Thus, the result of applying the input function is a collection of collections. If the same input function were passed to the map method, it would return a collection of collections. The *flatMap* method instead returns a flattened collection. The following example illustrates a use of *flatMap*.

```
val line = "Scala is fun"
val SingleSpace = " "
val words = line.split(SingleSpace)
val arrayOfChars = words flatMap {_.toList}
```

The *toList* method of a collection creates a list of all the elements in the collection. It is a useful method for converting a string, an array, a set, or any other collection type to a list.

c) filter

The *filter* method applies a predicate to each element in a collection and returns another collection consisting of only those elements for which the predicate returned true. A predicate is function that returns a Boolean value. It returns either true or false.

```
val xs = (1 to 100).toList
val even = xs filter {_ %2 == 0}
```

d) foreach

The *foreach* method of a Scala collection calls its input function on each element of the collection, but does not return anything. It is similar to the map method. The only difference between the two methods is that map returns a collection and *foreach* does not return anything. It is a rare method that is used for its side effects.

```
val words = "Scala is fun".split(" ")
words.foreach(println)
```

e) reduce

The *reduce* method returns a single value. As the name implies, it reduces a collection to a single value. The input function to the *reduce* method takes two inputs at a time and returns one value. Essentially, the input function is a binary operator that must be both associative and commutative. The following code shows some examples.

```
val xs = List(2, 4, 6, 8, 10)
val sum  = xs reduce {(x,y) => x + y}
val product  = xs reduce {(x,y) => x * y}
val max = xs reduce {(x,y) => if (x > y) x else y}
val min = xs reduce {(x,y) => if (x < y) x else y}
```

Here is another example that finds the longest word in a sentence

```
val words = "Scala is fun" split(" ")
val longestWord = words reduce {(w1, w2) => if(w1.length > w2.length) w1 else w2}
```

Note that the map and reduce operations in Hadoop MapReduce are similar to the *map* and *reduce* methods that we discussed in this section. In fact, Hadoop MapReduce borrowed these concepts from functional programming.

17. Summary

Scala is a powerful JVM-based statically typed language for developing multi-threaded and distributed applications. It combines the best of both object-oriented and functional programming. In addition, it is seamlessly interoperable with Java. You can use any Java library from Scala and vice-versa.

The key benefits of using Scala include a significant jump in developer productivity and code quality. In addition, it makes it easier to develop robust multi-threaded and distributed applications. Spark is written in Scala. It is just one example of the many popular distributed systems built with Scala.