Scala

Overview

Before we jump into the examples, here are a few important things to know about Scala:

- It’s a high-level language
- It’s statically typed
- Its syntax is concise but still readable — we call it expressive
- It supports the object-oriented programming (OOP) paradigm
- It supports the functional programming (FP) paradigm
- It has a sophisticated type inference system
- Scala code results in .class files that run on the Java Virtual Machine (JVM)
- It’s easy to use Java libraries in Scala

Hello, world

```
object Hello extends App {
    println("hello, world")
}
```

```
scalac Hello.scala
```

If you’re coming to Scala from Java, scalac is just like javac, and that command creates two files:

- Hello$.class
- Hello.class
- Hello$delayedInit$body.class

```
scala Hello
```

hello, world

```
>scala
Welcome to Scala 2.13.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_181).
Type in expressions for evaluation. Or try :help.
```

```
scala> val x = 1
val x: Int = 1

scala> val y = x + 1
val y: Int = 2

scala>
```

Two types of variables

- **val** is an immutable variable — like **final** in Java — and should be preferred
- **var** creates a mutable variable, and should only be used when there is a specific reason to use it

```
val x = 1   //immutable
var y = 0   //mutable
```

Declaring variable types

```
val x = 1
val s = "a string"
val p = new Person("Regina")
```

When you do this, Scala can usually infer the data type for you, as shown in these REPL examples:

```
scala> val x = 1
val x: Int = 1

scala> val s = "a string"
val s: String = a string
```

Control structures

if/else

```
if (test1) {
    doA()
} else if (test2) {
    doB()
} else if (test3) {
    doC()
} else {
    doD()
}
```

However, unlike Java and many other languages, the if/else construct returns a value, so, among other things, you can use it as a ternary operator:

```
val x = if (a < b) a else b
```

match expressions

Scala has a **match** expression, which in its most basic use is like a Java switch statement:

```
val result = i match {
    case 1 => "one"
    case 2 => "two"
    case _ => "not 1 or 2"
}
```

The **match** expression isn't limited to just integers, it can be used with any data type, including booleans:

```
val booleanAsString = bool match {
    case true => "true"
    case false => "false"
}
```

Here's an example of **match** being used as the body of a method, and matching against many different types:

```
def getClassAsString(x: Any):String = x match {
    case s: String => s + " is a String"
    case i: Int => "Int"
    case f: Float => "Float"
    case l: List[_] => "List"
    case p: Person => "Person"
    case _ => "Unknown"
}
```

try/catch

Scala's try/catch control structure lets you catch exceptions. It's similar to Java, but its syntax is consistent with match expressions:

```
try {
    writeToFile(text)
} catch {
    case fnfe: FileNotFoundException => println(fnfe)
    case ioe: IOException => println(ioe)
}
```

for loops and expressions

Scala **for** loops — which we generally write in this book as **for-loops** — look like this:

```
for (arg <- args) println(arg)

// "x to y" syntax
for (i <- 0 to 5) println(i)

// "x to y by" syntax
for (i <- 0 to 10 by 2) println(i)
```

You can also add the yield keyword to for-loops to create for-expressions that yield a result. Here's a for-expression that doubles each value in the sequence 1 to 5:

```
val x = for (i <- 1 to 5) yield i * 2
```

Here's another for-expression that iterates over a list of strings:

```
val fruits = List("apple", "banana", "lime", "orange")

val fruitLengths = for {
    f <- fruits
    if f.length > 4
} yield f.length
```

while and do/while

Scala also has **while** and **do/while** loops. Here's their general syntax:

```
// while loop
while(condition) {
    statement(a)
    statement(b)
}

// do-while
```

```
do {
   statement(a)
   statement(b)
}
while(condition)
```

Classes

Here’s an example of a Scala class:

```
class Person(var firstName: String, var lastName: String) {
    def printFullName() = println(s"$firstName $lastName")
}
```

This is how you use that class:

```
val p = new Person("Julia", "Kern")
println(p.firstName)
p.lastName = "Manes"
p.printFullName()
```

Notice that there’s no need to create “get” and “set” methods to access the fields in the class.

```
class Pizza (
    var crustSize: CrustSize,
    var crustType: CrustType,
    val toppings: ArrayBuffer[Topping]
) {
    def addTopping(t: Topping): Unit = toppings += t
    def removeTopping(t: Topping): Unit = toppings -= t
    def removeAllToppings(): Unit = toppings.clear()
}
```

In that code, an **ArrayBuffer** is like Java’s **ArrayList**. The **CrustSize**, **CrustType**, and **Topping** classes aren’t shown, but you can probably understand how that code works without needing to see those classes.

Scala methods

Just like other OOP languages, Scala classes have methods, and this is what the Scala method syntax looks like:

```
def sum(a: Int, b: Int): Int = a + b
def concatenate(s1: String, s2: String): String = s1 + s2
```

You don't have to declare a method's return type, so it's perfectly legal to write those two methods like this, if you prefer:

```
def sum(a: Int, b: Int) = a + b
def concatenate(s1: String, s2: String) = s1 + s2
```

This is how you call those methods:

```
val x = sum(1, 2)
val y = concatenate("foo", "bar")
```

There are more things you can do with methods, such as providing default values for method parameters, but that's a good start for now.

Traits

Traits in **Scala** are a lot of fun, and they also let you break your code down into small, modular units. To demonstrate traits, here's an example from later in the book. Given these three traits:

```
trait Speaker {
    def speak(): String  // has no body, so it's abstract
}

trait TailWagger {
    def startTail(): Unit = println("tail is wagging")
    def stopTail(): Unit = println("tail is stopped")
}

trait Runner {
    def startRunning(): Unit = println("I'm running")
    def stopRunning(): Unit = println("Stopped running")
}
```

You can create a **Dog** class that extends all of those traits while providing behavior for the **speak** method.

```
class Dog(name: String) extends Speaker with TailWagger with Runner {
    def speak(): String = "Woof!"
}
```

Similarly, here's a Cat class that shows how to override multiple trait methods:

```
class Cat extends Speaker with TailWagger with Runner {
    def speak(): String = "Meow"
```

```scala
    override def startRunning(): Unit = println("Yeah ... I don’t run")
    override def stopRunning(): Unit = println("No need to stop")
}
```