

Learn Scala for Java Developers

Toby Weston



baddotrobot.com

Learn Scala for Java Developers

Toby Weston

©2015 Toby Weston

Contents

Introduction	i
I. Scala Tour	1
The Scala Language	2
Pure Functions	2
Higher-Order Functions	3
Scala's Background	3
The Future	3
Installing Scala	5
The Scala Interpreter	5
Scala Scripts	6
scalac	7
Some Basic Syntax	9
Defining Values and Variables	9
Defining Functions	10
Operator Overloading and Infix Notation	13
Collections	14
Java Interoperability	15
Primitive Types	16
Scala's Class Hierarchy	18
AnyVal	19
Unit	20
AnyRef	21

CONTENTS

Bottom Types	23
ScalaDoc	25
Language Features	28
Working with Source Code	28
Working with Methods	29
Functional Programming	31
II. Key Syntactical Differences	32
Classes and Fields	35
Creating Classes	35
Derived Setters and Getters	36
Redefining Setters and Getters	41
Summary	45
Classes and Objects	47
Classes Without Constructor Arguments	47
Additional Constructors	49
Singleton Objects	52
Companion Objects	55

Introduction

This eBook contains sample chapters from *Learn Scala for Java Developers*. It includes all the chapters from Part I. and a selection from Part II.

Learn Scala for Java Developers is for Java developers looking to transition to programming in Scala. Scala is a concise, statically typed scripting language that runs on the Java Virtual Machine. It is both a functional programming language and an object-oriented language but its emphasis on functional programming is what sets it apart from Java.

This book will help you translate the Java you already know into Scala.

The Full Book

What's included in the full book.

- Tour Scala and learn the basic syntax, constructs and how to use the REPL
- Translate Java syntax that you already know into Scala
- Learn what Scala offers over and above Java, learn functional programming concepts and idioms
- Tips and advice useful when transitioning existing Java projects to Scala
- Syntax cheat sheet

The Structure of the Book

The full book is split into four parts: a tour of Scala, a comparison between Java and Scala, Scala-specific features and functional programming idioms, and finally a discussion about adopting Scala into existing Java teams.

In the first part, we're going to take a high-level tour of Scala. You'll get a feel for the language's constructs and how Scala is similar in a lot of ways to Java, yet very

different in others. We'll take a look at installing Scala and using the interactive interpreter and we'll go through some basic syntax examples.

Part II. talks about key differences between Java and Scala. We'll look at what's missing in Scala compared to Java and how concepts translate from one language to another.

Then in Part III., we'll talk about some of the language features that Scala offers that aren't found in Java. This section also talks a little about functional programming idioms.

Finally, we'll talk about adopting Scala into legacy Java projects and teams. It's not always an easy transition, so we'll look at why you would want to, and some of the challenges you might face.

I. Scala Tour

Welcome to *Learn Scala for Java Developers*. This book will help you transition from programming in Java to programming in Scala. It's designed to help Java developers get started with Scala without necessarily adopting all of the more advanced functional programming idioms.

Scala is both an object-oriented language and a functional language and although I do talk about some of the advantages of functional programming, this book is more about being productive with imperative Scala than getting to grips with functional programming. If you're already familiar with Scala but are looking to make the leap to pure functional programming, this probably isn't the book for you. Check out the excellent *Functional Programming in Scala*¹ by Paul Chiusano and Rúnar Bjarnaso instead.

The book often compares “like-for-like” between Java and Scala. So, if you're familiar with doing something a particular way in Java, I'll show how you might do the same thing in Scala. Along the way, I'll introduce the Scala language syntax.

¹<http://amzn.to/1Aegnwj>

The Scala Language

Scala is both a functional programming language *and* an object-oriented programming language. As a Java programmer, you'll be comfortable with the object-oriented definition: Scala has classes, objects, inheritance, composition, polymorphism — all the things you're used to in Java.

In fact, Scala goes somewhat further than Java. There are no “non”-objects. Everything is an object, so there are no primitive types like `int` and no static methods or fields. Functions are objects and even *values* are objects.

Scala can be accurately described as a functional programming language because it meets some fairly formal criteria. For example, it allows you both to define *pure functions* and use *higher-order functions*.

Pure Functions

Pure functions aren't associated with objects, and work without side effects. A key concern in Scala programming is avoiding mutation. There's even a keyword to define a fixed variable, a little like Java's `final: val`.

Pure functions should operate by *transformation* rather than *mutation*. That is to say, a pure function should take arguments and return results but not modify the environment it operates in. This *purity of function* is what enables *referential transparency*.

Higher-Order Functions

Functional languages should treat functions as *first-class* citizens. This means they support higher-order functions: functions that take functions as arguments or return functions and allow functions to be stored for later execution. This is a powerful technique in functional programming.

Scala's Background

Scala started life in 2003 as a research project at EPFL in Switzerland. The project was headed by Martin Odersky, who'd previously worked on Java generics and the Java compiler for Sun Microsystems.

It's quite rare for an academic language to cross over into industry, but Odersky and others launched Typesafe Inc., a commercial enterprise built around Scala. Since then, Scala has moved into the mainstream as a development language.

Scala offers a more concise syntax than Java but runs on the JVM. Running on the JVM should (in theory) mean an easy migration to production environments; if you already have the Oracle JVM installed in your production environment, it makes no difference if the bytecode was generated from the Java or Scala compiler.

It also means that Scala has Java interoperability built in, which in turn means that Scala can use any Java library. One of Java's strengths over its competitors was always the huge number of open source libraries and tools available. These are pretty much all available to Scala too. The Scala community has that same open source mentality, and so there's a growing number of excellent Scala libraries out there.

The Future

Scala has definitely moved into the mainstream as a popular language. It has been adopted by lots of big companies including Twitter, eBay, Yahoo, HSBC, UBS, and Morgan Stanley, and it's unlikely to fall out of favour anytime soon. If you're nervous about using it in production, don't be; it's backed by an international organisation and regularly scores well in popularity indexes.

The tooling is still behind Java though. Powerful IDEs like IntelliJ's IDEA and Eclipse make refactoring Java code straightforward but aren't quite there yet for Scala. The same is true of compile times: Scala is a lot slower to compile than Java. These things will improve over time, and on balance, they're not the biggest hindrances I encounter when developing.

Installing Scala

There are a couple of ways to get started with Scala.

1. You can run Scala interactively with the interpreter,
2. run shorter programs as shell scripts, or
3. compile programs with the `scalac` compiler.

The Scala Interpreter

Before working with an IDE, it's probably worth getting familiar with the Scala interpreter, or REPL.

Download the latest Scala binaries (from <http://scala-lang.org/downloads>) and extract the archive. Assuming you have Java installed, you can start using the interpreter from a command prompt or terminal window straight away. To start up the interpreter, navigate to the exploded folder and type²:

```
bin/scala
```

You'll be faced with the Scala prompt.

```
scala> _
```

You can type commands followed by `enter`, and the interpreter will evaluate the expression and print the result. It reads, evaluates and prints in a loop so it's known as a REPL.

²If you don't want to change into the install folder to run the REPL, set the `bin` folder on your path.

If you type `42*4` and hit enter, the REPL evaluates the input and displays the result.

```
scala> 42*4  
res0: Int = 168
```

In this case, the result is assigned to a variable called `res0`. You can go on to use this, for example to get half of `res0`.

```
scala> res0 / 2  
res1: Int = 84
```

The new result is assigned to `res1`.

Notice the REPL also displays the type of the result: `res0` and `res1` are both integers (`Int`). Scala has inferred the types based on the values.

If you add `res1` to the end of a string, no problem; the new result object is a string.

```
scala> "Hello Prisoner " + res1  
res2: String = Hello Prisoner 84
```

To quit the REPL, type:

```
:quit
```

Scala Scripts

The creators of Scala originally tried to promote the use of Scala from Unix shell scripts. As competition to Perl, Groovy or bash scripts on Unix environments it didn't really take off, but if you want to, you can create a shell script to wrap Scala:

```
1  #!/bin/sh
2  exec scala "$0" "$@"
3  !#
4  object HelloWorld {
5      def main(args: Array[String]) {
6          println("Hello, " + args.toList)
7      }
8  }
9  HelloWorld.main(args)
```

Don't worry about the syntax or what the script does (although I'm sure you've got a pretty good idea already). The important thing to note is that some Scala code has been embedded in a shell script and that the last line is the command to run.

You'd save it as a `.sh` file, for example `hello.sh`, and execute it like this:

```
./hello.sh World!
```

The `exec` command on line 2 spawns a process to call `scala` with arguments; the first is the script filename itself (`hello.sh`) and the second is the arguments to pass to the script. The whole thing is equivalent to running Scala like this, passing in a shell script as an argument:

```
scala hello.sh World!
```

scalac

If you'd prefer, you can compile `.scala` files using the Scala compiler.

The `scalac` compiler works just like `javac`. It produces Java bytecode that can be executed directly on the JVM. You run the generated bytecode with the `scala` command. Just like Java though, it's unlikely you'll want to build your applications from the command line.

All the major IDEs support Scala projects, so you're more likely to continue using your favorite IDE. We're not going to go into the details of how to set up a Scala

project in each of the major IDEs; if you're familiar with creating Java projects in your IDE, the process will be very similar.

For reference though, here are a few starting points.

- You can create bootstrap projects with Maven and the `maven-scala-plugin`.
- You can create a new Scala project directly within IntelliJ IDEA once you've installed the `scala` plugin (available in the JetBrains repository).
- Similarly, you can create a new Scala project directly within Eclipse once you have the Scala IDE plugin. Typesafe created this and it's available from the usual update sites. You can also download a bundle directly from the `scala-lang` or `scala-ide.org` sites.
- You can use SBT and create a build file to compile and run your project. SBT stands for Simple Build Tool and it's akin to Ant or Maven, but for the Scala world.
- SBT also has plugins for Eclipse and IDEA, so you can use it directly from within the IDE to create and manage the IDE project files.

Some Basic Syntax

Defining Values and Variables

Let's look at some syntax. We'll start by creating a variable:

```
val language: String = "Scala";
```

We've defined a variable as a `String` and assigned to it the value of "Scala". I say "variable", but we've actually created an immutable *value* rather than a *variable*. The `val` keyword creates a constant, and `language` cannot be modified from this point on. Immutability is a key theme you'll see again and again in Scala.

If we will want to modify `language` later, we can use `var` instead of `val`. We can then reassign it if we need to.

```
var language: String = "Java";  
language = "Scala";
```

So far, this doesn't look very different from Java. In the variable definition, the type and variable name are the opposite way round compared to Java, but that's about it. However, Scala uses type inference heavily, so Scala knows that the `var` above is a string, even if we don't tell it.

```
val language = "Scala";
```

Similarly, it knows that the expression is finished without needing to tell it explicitly with the semicolon. So we can drop that too.

```
val language = "Scala"           // no semicolon
```

You only need to add semicolons when you use multiple expressions on the same line; otherwise things get too complex for the compiler.

Operator precedence is just as you'd expect in Java. In the example below, the multiplication happens before the subtraction.

```
scala> val age = 35
scala> var maxHeartRate = 210 - age * .5
res0: Double = 191.5
```

Defining Functions

Function and method definitions start with the `def` keyword, followed by the signature. The signature looks similar to a Java method signature but with the parameter types the other way round again, and the return type at the end rather than the start.

Let's create a function to return the minimum of two numbers.

```
def min(x: Int, y: Int): Int = {
  if (x < y)
    return x
  else
    return y
}
```

We can test it in the REPL by calling it:


```
scala> min(34, 3)
res3: Int = 3
```

```
scala> min(10, 50)
res4: Int = 10
```

Note that Scala can't infer the types of function arguments.

Another trick is that you can drop the return statement. The last statement in a function will implicitly be the return value.

```
def min(x: Int, y: Int): Int = {
  if (x < y)
    x
  else
    y
}
```

Running it the REPL would show the following:

```
scala> min(300, 43)
res5: Int = 43
```

In this example, the `else` means the last statement is consistent with a `min` function. If I forgot the `else`, the last statement would be the same regardless and there would be a bug in our implementation:

```
def min(x: Int, y: Int): Int = {
  if (x < y)
    x
  y           // bug! where's the else?
}
```

It always returns y:

```
scala> min(10, 230)
res6: Int = 230
```

If you don't use any return statements, the return type can usually be inferred.

```
// the return type can be omitted
def min(x: Int, y: Int) = {
  if (x < y)
    x
  else
    y
}
```

Note that it's the equals sign that says this function returns something. If I write this function on one line, without the return type and just the equals sign, it starts to look like a real expression rather than a function.

```
def min(x: Int, y: Int) = if (x < y) x else y
```

Be wary, though; if you accidentally drop the equals sign, the function won't return anything. It'll be similar to the void in Java.

```
def min(x: Int, y: Int) {
  if (x < y) x else y
}
<console>:8: warning: a pure expression does nothing in statement position;
           you may be omitting necessary parentheses
      if (x < y) x else y
          ^
<console>:8: warning: a pure expression does nothing in statement position;
           you may be omitting necessary parentheses
      if (x < y) x else y
          ^
min: (x: Int, y: Int)Unit
```

Although this compiles okay, the compiler warns that you may have missed off the equals sign.

Operator Overloading and Infix Notation

One interesting thing to note in Scala is that you can override operators. Arithmetic operators are, in fact, just methods in Scala. As such, you can create your own. Earlier, we saw the integer `age` used with a multiplier.

```
val age: Int  
age * .5
```

The value `age` is an integer and there is a method called `*` on the integer class. It has the following signature:

```
def *(x: Double): Double
```

Numbers are objects in Scala, as are literals. So you can call `*` directly on a variable or a number.

```
age.*( .5)  
5.*(10)
```

Using the *infix notation*, you're able to drop the dot notation for variables and literals and call:

```
age * .5
```

or, as another example:

```
35 toString
```

Remember, `35` is an instance of `Int`.

Specifically, Scala support for infix notation means that when a method takes zero or one arguments you can drop the dot and parentheses, and if there is more than one argument you can drop the dot.

For example:

```
35 + 10  
"aBCDEFG" replace("a", "A")
```

It's optional though; you can use the dot notation if you prefer.

What this means is that you can define your own plus or minus method on your own classes and use it naturally with infix notation. For example, you might have a `Passenger` join a `Train`.

```
train + passenger
```

There are not many restrictions on what you can call your functions and methods; you can use any symbol that makes sense to your domain.

Collections

Scala comes with its own immutable collection types as well as mutable versions. By default immutability is preferred, so we can create a list with the following:

```
val list = List("a", "b", "c")
```

And create a map with:

```
val map = Map(1 -> "a", 2 -> "b")
```

where the arrow goes from the key to the value. These will be immutable; you won't be able to add or remove elements.

You can process them in a similar way to Java 8's `forEach` and lambda syntax:

```
list.foreach(value => println(value))           // scala
```

which is equivalent to the following in Java:

```
list.forEach(value -> System.out.println(value)); // java
```

Like Java 8's method reference syntax, you can auto-connect the lambda argument to the method call.

```
list.foreach(println)                           // scala
```

which is roughly equivalent to this Java:

```
list.forEach(System.out::println);              // java
```

There are lots of other Scala-esque ways to process collections. We'll look at these later, but the most common way to iterate is a `for` loop written like this:

```
for (value <- list) println(value)
```

which reads, "for every value in list, print the value". You can also do it in reverse:

```
for (value <- list.reverse) println(value)
```

or you might like to break it across multiple lines:

```
for (value <- list) {  
  println(value)  
}
```

Java Interoperability

I mentioned that you can use any Java class from Scala. For example, let's say we want to create a `JavaList` rather than the usual Scala immutable `List`.

```
val list = new java.util.ArrayList[String]
```

All we did was fully qualify the class name (`java.util.ArrayList`) and use `new` to instantiate it. Notice the square brackets? Scala denotes generics using `[]` rather than `<>`. We also didn't have to use the parentheses on the constructor, as we had no arguments to pass in.

We can make method calls — for example, adding an element — just as you'd expect:

```
list.add("Hello")
```

or, using infix:

```
list add "World!"
```

Primitive Types

In Java there are two integer types: the primitive (non-object) `int` and the `Integer` class. Scala has no concept of primitives — everything is an object — so, for example, Scala's integer type is an `Int`. Similarly, you'll be familiar with the other basic types:

```
Byte  
Short  
Int  
Long  
Char  
String  
Float  
Double  
Boolean
```

Although Scala has its own richer types, typically they just wrap the Java types. When working with these basic types, nine times out of ten you won't need to worry if you're using Scala or Java types. Things are pretty seamless. For example, Scala has a `BigDecimal` type with a `+` method which means you can add two big decimals with much less code than in Java.

Compare the following Scala to Java:

```
// scala
```

```
val total = BigDecimal(10000) + BigDecimal(200)
```

```
// java
```

```
BigDecimal total = new BigDecimal(10000).add(new BigDecimal(200));
```

Scala hasn't reimplemented Java's `BigDecimal`; it just delegates to it and saves you having to type all that boilerplate.

Scala's Class Hierarchy

Scala's class hierarchy starts with the `Any` class in the `scala` package. It contains methods like `==`, `!=`, `equals`, `##`, `hashCode`, and `toString`.

```
abstract class Any {  
  final def ==(that: Any): Boolean  
  final def !=(that: Any): Boolean  
  def equals(that: Any): Boolean  
  def ##: Int  
  def hashCode: Int  
  def toString: String  
  // ...  
}
```

Every class in Scala inherits from the abstract class `Any`. It has two immediate subclasses, `AnyVal` and `AnyRef`.

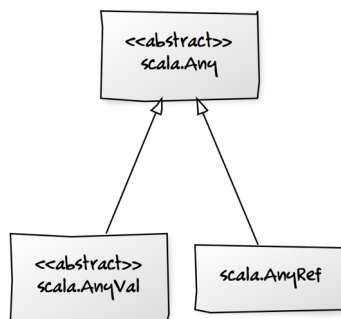


Fig. 1.1. Every class extends the `Any` class.

AnyVal

`AnyVal` is the super-type to all *value types*, and `AnyRef` the super-type of all *reference types*.

Basic types such as `Byte`, `Int`, `Char`, etc. are known as value types. In Java value types correspond to the primitive types, but in Scala they are objects. Value types are all predefined and can be referred to by literals. They are usually allocated on the stack but are allocated on the heap in Scala.

All other types in Scala are known as reference types. Reference types are objects in memory (the heap), as opposed to pointer types in C-like languages, which are addresses in memory that point to something useful and need to be dereferenced using special syntax (for example, `*age = 64` in C). Reference objects are effectively dereferenced automatically.

There are nine value types in Scala:

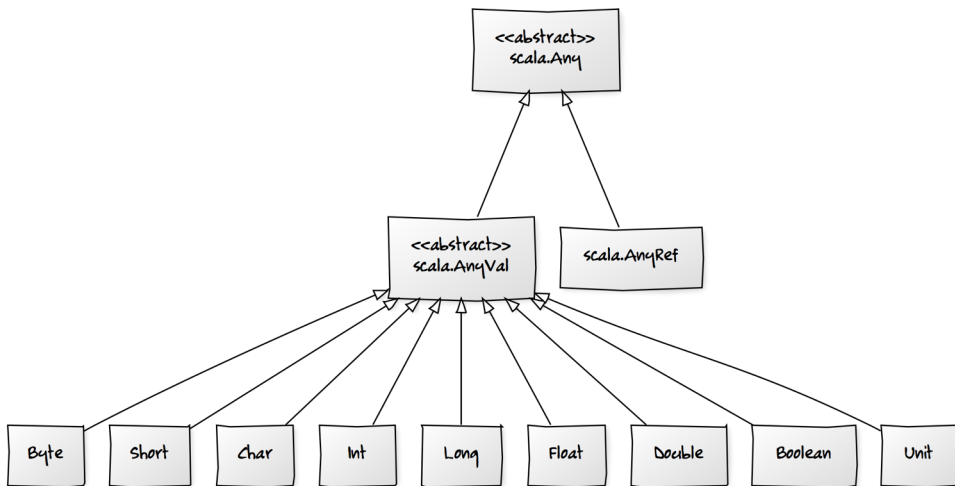


Fig. 1.2. Scala's value types.

These classes are fairly straightforward; they mostly wrap an underlying Java type and provide implementations for the `==` method that are consistent with Java's `equals` method.

This means, for example, that you can compare two number objects using `==` and get a sensible result, even though they may be distinct instances.

So, `42 == 42` in Scala is equivalent to creating two `Integer` objects in Java and comparing them with the `equals` method: `new Integer(42).equals(new Integer(42))`. You're not comparing object references, like in Java with `==`, but natural equality. Remember that `42` in Scala is an instance of `Int` which in turn delegates to `Integer`.

Unit

The `Unit` value type is a special type used in Scala to represent an uninteresting result. It's similar to Java's `Void` object or `void` keyword when used as a return type. It has only one value, which is written as an empty pair of brackets:

```
scala> val example: Unit = ()
example: Unit = ()
```

A Java class implementing `Callable` with a `Void` object as a return would look like this:

```
// java
public class DoNothing implements Callable<Void> {
    @Override
    public Void call() throws Exception {
        return null;
    }
}
```

It is identical to this Scala class returning `Unit`:

```
// scala
class DoNothing extends Callable[Unit] {
  def call: Unit = ()
}
```

Remember that the last line of a Scala method is the return value, and `()` represents the one and only value of `Unit`.

AnyRef

`AnyRef` is actually an alias for Java's `java.lang.Object` class. The two are interchangeable. It supplies default implementations for `toString`, `equals` and `hashCode` for all reference types.

There used to be a subclass of `AnyRef` called `ScalaObject` that all Scala reference types extended. However, it was only there for optimisation purposes and has been removed in Scala 2.11. (I mention it as a lot of documentation still refers to it.)

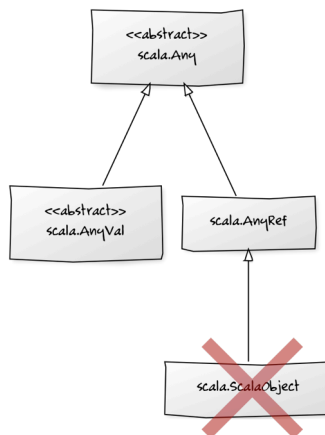


Fig. 1.3. Scala `Any`. The `ScalaObject` class no longer exists.

The Java `String` class and other Java classes used from Scala all extend `AnyRef`. (Remember it's a synonym for `java.lang.Object`.) Any Scala-specific classes, like Scala's implementation of a list, `scala.List`, also extend `AnyRef`.

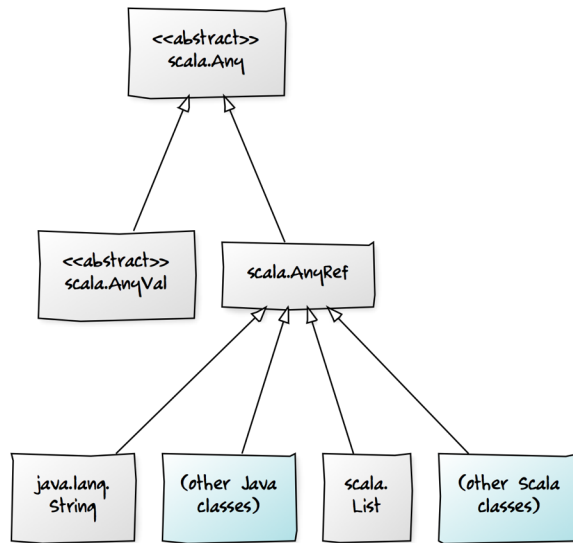


Fig. 1.4. Scala's reference types.

For reference types like these, `==` will delegate to the `equals` method like before. For pre-existing classes like `String`, `equals` is already overridden to provide a natural notion of equality. For your own classes, you can override the `equals` just as you would in Java, but still be able to use `==` in code.

For example, you can compare two strings using `==` in Scala and it would behave just as it would in Java if you used the `equals` method:

```
new String("A") == new String("A")           // true in scala, false in java
new String("B").equals(new String("B"))       // true in scala and java
```

If, however, you want to revert back to Java's semantics for `==` and perform reference equality in Scala, you can call the `eq` method defined in `AnyRef`:

```
new String("A") eq new String("A")           // false in scala
new String("B") == new String("B")           // false in java
```

Bottom Types

A new notion to many Java developers will be the idea that a class hierarchy can have common *bottom types*. These are types that are subtypes of *all* types. Scala's types `Null` and `Nothing` are both bottom types.

All reference types in Scala are super-types of `Null`. `Null` is also an `AnyRef` object; it's a subclass of every reference type.

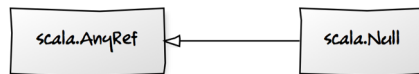


Fig. 1.5. The `Null` extends `AnyRef`.

Both value and reference types are super-types of `Nothing`. It's at the bottom of the class hierarchy and is a subtype of all types.

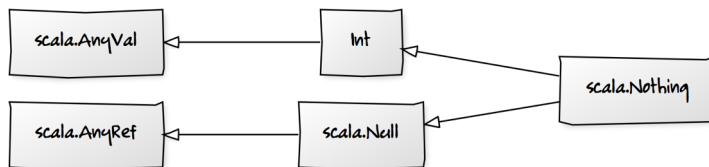


Fig. 1.6. `Nothing` extends `Null`.

The entire hierarchy is shown in the diagram below. I've left off `scala.Any` to save space. Notice that `Null` extends all reference types and that `Nothing` extends all types.

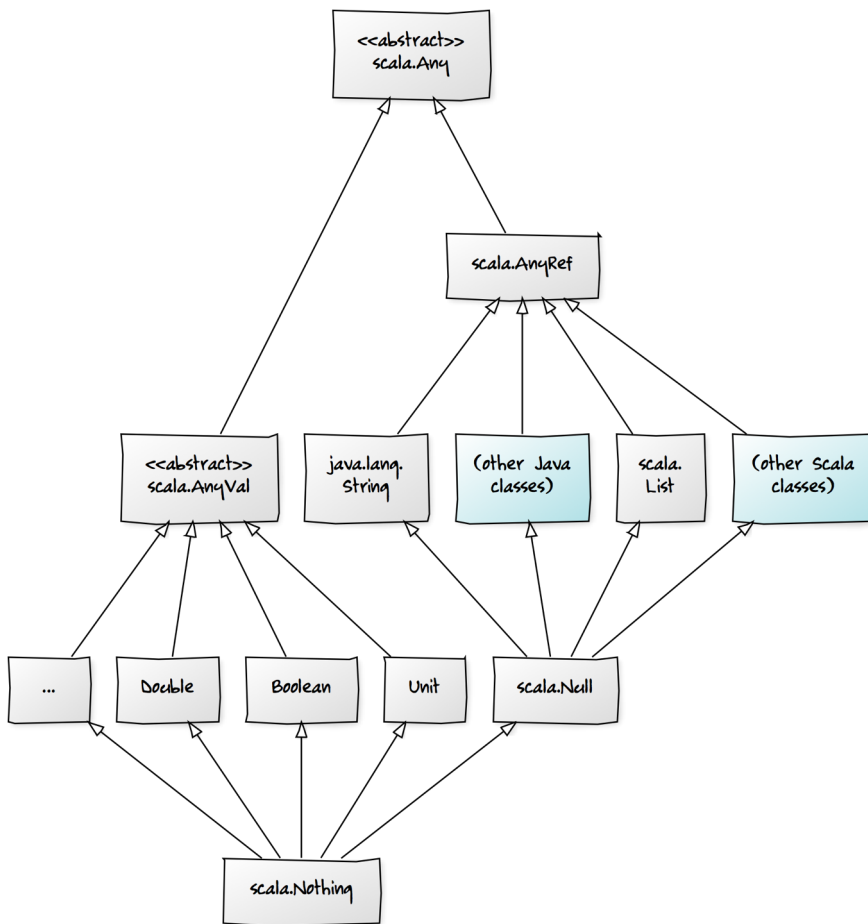


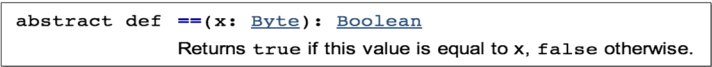
Fig. 1.7. Full hierarchy with the bottom types `Null` and `Nothing`.

ScalaDoc

Scala has ported the idea of JavaDoc and creatively called it ScalaDoc. Adding ScalaDoc to your Scala source works similarly to adding JavaDoc, and is done with markup in comments. For example, the following fragment in source code:

```
/** Returns `true` if this value is equal to x, `false` otherwise. */  
def ==(x: Byte): Boolean
```

...can be turned into the following fragment in HTML:



```
abstract def ==(x: Byte): Boolean  
Returns true if this value is equal to x, false otherwise.
```

Fig. 1.8. Embedded ScalaDoc markup gets rendered in HTML.

To see the documentation for the Scala API, head over to <http://scala-lang.org/documentation>. You'll notice it is broadly similar to JavaDoc. You can see the classes along the left; they're not grouped by package like in JavaDoc but they're clickable to get more information.

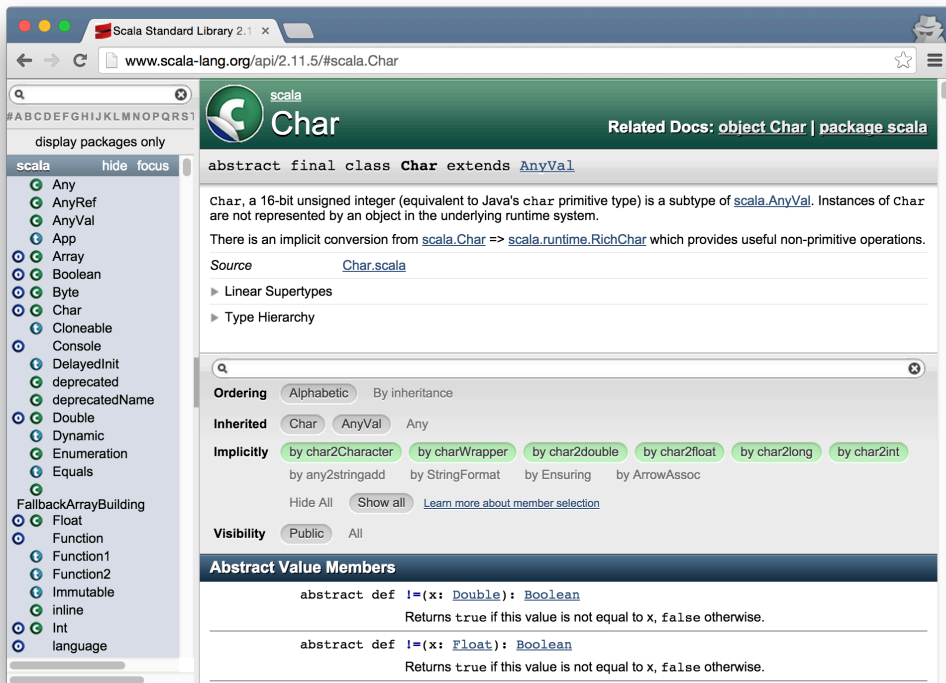


Fig. 1.9. The basic ScalaDoc for Char.

A neat feature of ScalaDoc is that you can also filter the content. For example, you can show only methods inherited from Any.

If you're interested in the hierarchy of a class, you can look at its super- and subtypes. You can even see a navigable diagram of the type hierarchy. For example, the type hierarchy diagram for Byte shows it is a subtype of AnyVal, and you can navigate up through the hierarchy by clicking on the diagram.

The screenshot displays the ScalaDoc website for the `Byte` class. The browser address bar shows `www.scala-lang.org/api/2.11.5/#scala.Byte`. The left sidebar contains a search bar and a list of Scala classes, with `Byte` selected. The main content area shows the `Byte` class definition: `abstract final class Byte extends AnyVal`. Below this, there is a description of `Byte` as a 8-bit signed integer and a note about an implicit conversion from `scala.Byte` to `scala.runtime.RichByte`. The `Source` link points to `Byte.scala`. The `Type Hierarchy` section shows a diagram where `Byte` inherits from `AnyVal` and has an implicit conversion to `java.lang.Byte`. Other related classes like `RichByte`, `Double`, `Float`, `Long`, `Int`, and `Short` are also shown. The bottom section includes filters for `Ordering` (Alphabetic, By inheritance), `Inherited` (Byte, AnyVal, Any), `Implicitly` (by byte2Byte, by byteWrapper, by byte2double, by byte2float, by byte2long, by byte2int, by byte2short, by any2stringadd, by StringFormat, by Ensuring, by ArrowAssoc), and `Visibility` (Public, All).

Fig. 1.10. ScalaDoc showing the type hierarchy diagram.

Language Features

On our tour we've seen some example syntax, walked through the class hierarchy, and briefly looked at ScalaDoc, but Scala offers heaps of other interesting language features.

In this chapter, we won't really talk about syntax but we'll discuss some of the things that make Scala an interesting and powerful language when working with source code, working with methods, and using its functional programming features.

Working with Source Code

Source Files. What you put in source files is much more flexible in Scala than in Java. There's no restriction on what a `.scala` file contains. A file called `Customer.scala` might contain a class called `Customer`, but it doesn't have to. Similarly, it might contain four classes, none of which are called `Customer`.

Packages. Packages are similar. Although they are essentially the same thing as in Java, classes in packages don't have to live in folders of the same name like they do in Java. There are some differences in scoping; for example, there's no `protected` keyword in Scala but you can use special syntax (`variable[package]`) to achieve the same thing.

Package Objects. Scala also has the idea of *package objects*. These are objects that you can put useful chunks of code in, for re-use within the package scope. They're available to other classes in the package, and if someone imports that package, everything within the package object is available to them too. Libraries often use these to allow you to import all their classes in one go.

Import Alias. Imports are about the same as in Java but once you've imported a class in Scala, you can rename it within your class. In other words, you can create an alias for a class within your class. This can be useful when you've got a name clash, for example between libraries.

Type Aliases. Scala also supports type aliases. You can give an alias to a complex type definition to help clarify the intent. It's similar to a structureless `typedef` or `#define` macro in C, or what's called *type synonyms* in Haskell.

Traits. Although Scala has classes and objects, there is no “interface” keyword. Instead, there is the idea of a `trait` which is similar to an interface but can have methods. It's somewhere between Java 8's default methods and Ruby's mixins.

Working with Methods

Generics. There's better support for generic covariance and contravariance in Scala than Java. This means that you can be more general and more flexible in your method signatures when generic types are used as arguments.

```
class Stack[+A] {  
  def push[B >: A](b: B): Stack[B] = ...  
}
```

Variable Arguments. When working with methods, Scala supports variable arguments or `varargs` just like Java. They look different (`def sum(numbers: Int*)`), but behave as you'd expect.

```
public add(String... names) // java
```

```
def add(names: String*) // scala
```

Named Method Arguments. Something Java doesn't offer is named method arguments and default values. In Scala, you can call a method with its arguments out of order, as long as you name them. So, given the function `def swap(first: Int, second: Int)`, you can call it explicitly, naming its arguments. Because they're named, the compiler can work out which is which regardless of their position. So the following is fine:

```
swap(first = 3, second = 1)
swap(second = 1, first = 3)
```

Default Values. You can add a default value by using `=` after the parameter declaration. For example, `def swap(first: Int, second: Int = 1)`. The second value will default to 1 if you leave it off when you call the function. You can still supply a value to override the default, and still use named parameters.

```
swap(3)
swap(3, 2)
swap(first = 3)
swap(first = 3, second = 1)
```

Lambdas. Scala supports lambdas or anonymous functions. You can pass function literals as arguments to methods and use a function signature as an argument in a method signature. So the `test` method below takes a function with no arguments which returns a `Boolean`.

```
def test(f: () => Boolean) = ...
```

When you call it, you can pass in a function literal as a parameter.

```
test(() => if (!tuesday) true else false)
```

As another example, you can create a function signature to represent a function from a `String` value to a `Boolean` like this:

```
def test(f: String => Boolean): Boolean = ...
```

and call it with a function literal like this:

```
test(value => if (value == "tuesday") true else false)
```

Functional Programming

Some other Scala features aimed more at functional programming include:

Pattern matching. This is a hugely powerful feature which at first blush looks similar to switches but can be used for much more.

For comprehensions. For comprehensions are subtly different than regular for loops, and are useful when working with functional constructs. When you first encounter them, they'll look like an alternative syntax to Java's for loop.

Currying. Although you can write your own currying functions in any language, Scala supports currying as a language feature. If you're unsure what currying is, you probably don't need to worry about it right now. See the [currying chapter](#) for more details.

Functional Literals. The language supports literals to represent some useful types like tuples. Popular Java functional libraries like [totally-lazy](#)³ or [functional-java](#)⁴ have these kinds of things; Scala just makes them easier to work with.

Recursion. Most languages support recursion, but Scala has compiler support for tail call optimisation, which means it can support recursive calls that would result in a stack overflow in languages like Java. The compiler can even perform some checks for you if you use the `@tailrec` annotation.

³<http://totallylazy.com/>

⁴<http://www.functionaljava.org/>

II. Key Syntactical Differences

This part of the book is about the key differences between Java and Scala language syntax. Given some typical Java code, we'll look at equivalent Scala syntax. In Part III., we'll look more at Scala features for which there is no direct equivalent in Java.

We're going to look at:

- Lots of things around classes and objects, creating classes, fields, methods. We'll do some round-tripping from Scala-generated bytecode back to Java, so that you can get a feel for how Scala relates to Java.
- Inheritance, interfaces, abstract classes and mixins.
- Common control structures like `for` loops.
- Generics.

Flexibility

Scala is very flexible. There are generally several ways to achieve the same thing. I don't mean the difference between using a `while` loop or `for` loop; I mean that the language has different syntax options for expressing the same thing. This flexibility gives a lot of freedom but can be confusing when you're reading code from different authors.

An example is the infix notation we saw earlier. You can often drop the dots and brackets when calling methods. Scala is opinion-less; it's up to you if you want to use the dots or not.

Java, on the other hand, is very restrictive; there are generally very few ways to express the same things. It's often easier to recognise things at a glance. You might

have to work a little harder to recognise some of the more exotic syntax options in Scala.

This is true when it comes to the structure of your code too; you can create functions within functions, import statements in the middle of a class, or have a class live in a file with an unrelated name. It can all be a little disorienting when you're used to the rigidity of Java.

Immutable and Declarative

Because Scala favours immutability, you might also notice a different approach to solving problems. For example, you might notice a lack of looping over mutable variables. Scala programs usually favour more functional idioms to achieve the same thing.

This more *declarative* way of doing things says “tell me what to do, not how to do it”. You may be more used to the Java / imperative way of doing things that says “tell me exactly how to do it”. Ultimately, when you give up the micro-management style of imperative programming, you allow the language more freedom in how it goes about its business.

For example, a traditional imperative for loop in Java looks like this:

```
// java
for (int count = 0; count < 100; count++) {
    System.out.println(count);
}
```

It's a typical imperative loop. We're telling it explicitly to enumerate serially from zero to one hundred. If, on the other hand, we use a more declarative mechanism, like this:

```
// scala
(0 to 100).foreach(println(_))
```

...the enumeration is done within the `foreach` method, not by a language construct. We're saying, “for a range of numbers, perform some function on each”. Although

only subtly different, we're not saying *how* to enumerate the sequence. It means Scala is free to implement the enumeration however it likes. (For example, it may choose to do it in parallel.)

Interestingly, Oracle has adopted these ideas in Java 8. If you've been using that, you're probably already familiar with the concepts.

Classes and Fields

In this chapter, we'll have a look at:

1. Creating classes
2. How Scala makes things easier when defining fields
3. What happens behind the scenes when Scala creates methods for you

Creating Classes

Creating a class in Java means writing something like this:

```
// java
public class Customer {
}
```

It makes sense for us to have a name and address for a customer. So adding these as fields and initialising via the constructor would give us something like this:

```
// java
public class Customer {
    private final String name;
    private final String address;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
    }
}
```

We can instantiate an instance with the `new` keyword and create a new customer called Eric like this:

```
Customer eric = new Customer("Eric", "29 Acacia Road"); // java
```

In Scala, the syntax is much briefer; we can combine the class and constructor on a single line.

```
class Customer(val name: String, val address: String) // scala
```

We new it up in the same way, like this:

```
val eric = new Customer("Eric", "29 Acacia Road") // scala
```

Rather than define the fields as members within the class, the Scala version declares the variables as part of the class definition in what's known as the *primary constructor*. In one line, we've declared the class `Customer` and, in effect, declared a constructor with two arguments.

Derived Setters and Getters

The `val` keyword on the class definition tells the compiler to treat the arguments as fields. It will create the fields and accessor methods for them.

We can prove this by taking the generated class file and decompiling it into Java. Round-tripping like this is a great way to explore what Scala actually produces behind the scenes. I've used the excellent [CFR decompiler](http://www.benf.org/other/cfr/)⁵ by Lee Benfield here, but you could also use the `javadoc` program that ships with Java to get the basic information.

⁵<http://www.benf.org/other/cfr/>

To run the decompiler on the Scala generated class file for Customer, you do something like the following:

```
java -jar cfr_0_99.jar target/scala-2.11/classes/scala/demo/Customer.class
```

It produces the following:

```
1  // decompiled from scala to java
2  public class Customer {
3      private final String name;
4      private final String address;
5
6      public String name() {
7          return this.name;
8      }
9
10     public String address() {
11         return this.address;
12     }
13
14     public Customer(String name, String address) {
15         this.name = name;
16         this.address = address;
17     }
18 }
```

What's important to notice is that Scala has generated accessor methods at lines 6 and 10, and a constructor at line 14. The accessors aren't using the Java getter convention, but we've got the equivalent of `getName` and `getAddress`.

You might also want to define fields but not have them set via the constructor. For example, in Java, we might want to add an `id` to the customer to be set later with a setter method. This is a common pattern for tools like Hibernate when populating an object from the database.

```
// java
public class Customer {
    private final String name;
    private final String address;

    private String id;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
    }

    public void setId(String id) {
        this.id = id;
    }
}
```

Is Scala, you do pretty much the same thing.

```
// scala
class Customer(val name: String, val address: String) {
    var id = ""
}
```

You define a field, in this case a `var`, and magically Scala will create a setter method for you. The setter method it creates is called `id_ =` rather than the usual `setId`. If we round-trip this through the decompiler, we see the following:

```
1  // decompiled from scala to java
2  public class Customer {
3      private final String name;
4      private final String address;
5      private String id;
6
7      public static Customer apply() {
8          return Customer$.MODULE$.apply();
9      }
10     public String name() {
11         return this.name;
12     }
13     public String address() {
14         return this.address;
15     }
16     public String id() {                // notice it's public
17         return this.id;
18     }
19     public void id_$eq(String x$1) {    // notice it's public
20         this.id = x$1;
21     }
22     public Customer(String name, String address) {
23         this.name = name;
24         this.address = address;
25         this.id = null;
26     }
27 }
```

Notice it has created a method called `id_$eq` on line 19 rather than `id_`; that's because the equals symbol isn't allowed in a method name on the JVM, so Scala has escaped it and will translate it as required. You can call the setter method directly like this:

```
new Customer("Bob", "10 Downing Street").id_=("000001")
```

Scala offers a shorthand, however; you can just use regular assignment and Scala will call the auto-generated `id_$eq` setter method under the covers:

```
new Customer("Bob", "10 Downing Street").id = "000001"
```

If there are no modifiers in front of a field, it means it's public. So as well as being able to call the auto-generated setter, clients could also work directly on the field, potentially breaking encapsulation. We'd like to be able to make the field private and allow updates only from within the Customer class.

To do this, just use the private keyword with the field.

```
class Customer(val name: String, val address: String) {  
    private var id = ""  
}
```

The decompiler shows that the setter and getter methods are now private.

```
1  // decompiled from scala to java  
2  public class Customer {  
3      private final String name;  
4      private final String address;  
5      private String id;  
6  
7      public String name() {  
8          return this.name;  
9      }  
10  
11     public String address() {  
12         return this.address;  
13     }  
14  
15     private String id() {                                // now it's private  
16         return this.id;  
17     }  
18  
19     private void id_$eq(String x$1) {                    // now it's private  
20         this.id = x$1;  
21     }  
22
```

```
23     public Customer(String name, String address) {
24         this.name = name;
25         this.address = address;
26         this.id = "";
27     }
28 }
```

Redefining Setters and Getters

The advantage of using setters to set values is that we can use the method to preserve invariants or perform special processing. In Java, it's straightforward: you create the setter method in the first place. It's more laborious for Scala, as the compiler is generating the methods.

For example, once the `id` has been set, we might want to prevent it from being updated. In Java, we could do something like this:

```
// java
public void setId(String id) {
    if (id.isEmpty())
        this.id = id;
}
```

Scala, on the other hand, creates the setter method automatically, so how do we redefine it? If we try to just replace the setter directly in the Scala code, we'd get a compiler error:

```
// scala doesn't compile
class Customer(val name: String, val address: String) {
  private var id = ""

  def id_=(value: String) {
    if (id.isEmpty)
      this.id = value
  }
}
```

Scala can't know to replace the method so it creates a *second method* of the same name, and the compiler fails when it sees the duplicate:

```
ambiguous reference to overloaded definition,
both method id_= in class Customer of type (value: String)Unit
and method id_= in class Customer of type (x$1: String)Unit
match argument types (String)
this.id = value
```

method id_= is defined twice

conflicting symbols both originated in file 'Customer.scala'

```
def id_=(value: String) {
  ^          ^
```

To redefine the method, we have to jump through some hoops. Firstly, we have to rename the field (say to `_id`), making it private so as to make the getter and setters private. Then we create a new getter method called `id` and setter method called `id_=` that are public and are used to access the renamed private field.


```
class Customer(val name: String, val address: String) {  
  private var _id: String = ""  
  
  def id = _id  
  
  def id_=(value: String) {  
    if (_id.isEmpty)  
      _id = value  
  }  
}
```

We've hidden the real field `_id` behind the `private` modifier and exposed a method called `id_` to act as a setter. As there is no field called `id` any more, Scala won't try to generate the duplicate method, and things compile.

```
// REPL session  
scala> val bob = new Customer("Bob", "32 Bread Street")  
bob: Customer = Customer@e955027  
  
scala> bob.id = "001"  
bob.id: String = 001  
  
scala> println(bob.id)  
001  
  
scala> bob.id = "002"  
bob.id: String = 001  
scala> println(bob.id)  
001
```

Looking at the decompiled version, you can see how to redefine the method. We've hidden the real field and exposed public methods to synthesize access to it under the guise of the field name.

```
1  // decompiled from scala to java
2  public class Customer {
3      private final String name;
4      private final String address;
5      private String _id;
6
7      public String name() {
8          return this.name;
9      }
10
11     public String address() {
12         return this.address;
13     }
14
15     private String _id() {                // private
16         return this._id;
17     }
18
19     private void _id_$eq(String x$1) {    // private
20         this._id = x$1;
21     }
22
23     public String id() {                  // public
24         return this._id();
25     }
26
27     public void id_$eq(String value) {    // public
28         if (!this._id().isEmpty()) return;
29         this._id_$eq(value);
30     }
31
32     public Customer(String name, String address) {
33         this.name = name;
34         this.address = address;
35         this._id = "";
36     }
37 }
```

Why the Getter?

You might be wondering why we created the getter method `def id()`. Scala won't allow us to use the shorthand assignment syntax to set a value unless the class has both the setter (`id_=`) and getter methods defined.

Summary

Creating classes is straightforward with Scala. You can add fields to the class simply by adding parameters to the class definition, and the equivalent Java constructor, getters and setters are generated for you by the compiler.

All fields in the class file are generated as private but have associated accessor methods generated. These generated methods are affected by the presence of `val` or `var` in the class definition.

- If `val` is used, a public getter is created but no setter is created. The value can only be set by the constructor.
- If `var` is used, a public getter and setter is created. The value can be set via the setter or the constructor.
- If neither `val` or `var` is used, no methods are generated and the value can only be used within the scope of the primary constructor; it's not really a field in this case.
- Prefixing the class definition with `private` won't change these rules, but will make any generated methods private.

This is summarised in the following table:

class Foo(? x)	val x	var x	x	private val x	private var x
Getter created (x())	Y (public)	Y (public)	N	Y (private)	Y (private)
Setter created (x_=(y))	N	Y (public)	N	N	Y (private)
Generated constructor includes x	Y	Y	N	Y	Y

If you need to override the generated methods, you have to rename the field and mark it as private. You then recreate the getter and setter methods with the original name. In practice, it's not something you'll have to do very often.

Classes and Objects

In this chapter we'll look at:

- How you can define fields within the class body rather than on the class definition line and how this affects the generated methods.
- How you create additional constructors.
- Scala's singleton objects defined with the `object` keyword.
- *Companion objects*, a special type of singleton object.

Classes Without Constructor Arguments

Let's begin by looking at how we create fields within classes without defining them on the class definition line. If you were to create a class in Scala with no fields defined on the class definition, like this:

```
// scala  
class Counter
```

...the Scala compiler would still generate a primary constructor with no arguments, a lot like Java's default constructor. So the Java equivalent would look like this:

```
// java  
public class Counter {  
    public Counter() {  
    }  
}
```

In Java you might initialise a variable and create some methods.

```
// java
public class Counter {

    private int count = 0;

    public Counter() {

    }

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

You can do the same in Scala.

```
// scala
class Counter {
    private var count = 0

    def increment() { // brackets to denote this is a "mutator"
        count += 1
    }

    def getCount = count
}
```

Within the primary constructor (i.e., not in the class definition but immediately afterwards in the class body), the `val` and `var` keywords will affect the bytecode like this:

Declared in primary constructor	val x	var x	x	private val x	private var x
Getter (x())	Y (public)	Y (public)	N/A	Y (private)	Y (private)
Setter (x_=(y))	N	Y (public)	N/A	N	Y (private)

As you can see, this is consistent with the table we saw earlier. Getters are generated by default for `val` and `var` types and will all be public. Adding `private` to the field declaration will make the generated fields private and setters are only generated for vars (which are, again, public by default).

Additional Constructors

Let's create an alternative Java version of our `Customer` class, this time with additional constructors.

```
// java
public class Customer {

    private final String fullname;

    public Customer(String forename, String initial, String surname) {
        this.fullname =
            String.format("%s %s. %s", forename, initial, surname);
    }

    public Customer(String forename, String surname) {
        this(forename, "", surname);
    }
}
```

We've defaulted the customer's initial and allowed clients to choose if they want to supply it.

We should probably tidy up the main constructor to reflect the fact that the variable could come through as an empty string. We'll add an if-condition and format the string depending on the result.

```
// java
public class Customer {
    private final String fullname;

    public Customer(String forename, String initial, String surname) {
        if (initial != null && !initial.isEmpty())
            this.fullname =
                String.format("%s %s. %s", forename, initial, surname);
        else
            this.fullname = String.format("%s %s", forename, surname);
    }

    public Customer(String forename, String surname) {
        this(forename, "", surname);
    }

    public static void main(String... args) {
        System.out.println(new Customer("Bob", "J", "Smith").fullname);
        System.out.println(new Customer("Bob", "Smith").fullname);
    }
}
```

Creating additional or *auxiliary constructors* in Scala is just a matter of creating methods called `this`. The one constraint is that each auxiliary constructor must call another constructor using `this` on its first line. That way, constructors will always be chained, all the way to the top.

Scala has the notion of a *primary constructor*; it's the code in the class body. Any parameters passed in from the class definition are available to it and if you don't write any auxiliary constructors, the class will still have a constructor; it's the implicit primary constructor.

```
// scala
class Customer(forename: String, initial: String, surname: String) {
    // primary constructor
}
```

So, if we create a field within the primary constructor and assign it some value,


```
// scala
class Customer(forename: String, initial: String, surname: String) {
  val fullname = String.format("%s %s. %s", forename, initial, surname)
}
```

...it would be equivalent to the following Java:

```
// java
public class Customer {
  private final String fullname;

  public Customer(String forename, String initial, String surname) {
    this.fullname =
      String.format("%s %s. %s", forename, initial, surname);
  }
}
```

If we can add an another auxiliary constructor to the Scala version, we can refer to this to chain the call to the primary constructor.

```
// scala
class Customer(forename: String, initial: String, surname: String) {
  val fullname = String.format("%s %s. %s", forename, initial, surname)

  def this(forename: String, surname: String) {
    this(forename, "", surname)
  }
}
```

Using Default Values

Scala has language support for default values on method signatures, so we could have written this using just parameters on the class definition, and avoided the extra constructor. We'd just default the value for `initial` to be an empty string. To make the implementation handle empty strings better, we can put some logic in the primary constructor like before.

```
class Customer(forename: String, initial: String = "", surname: String) {  
    val fullname = if (initial != null && !initial.isEmpty)  
        forename + " " + initial + ". " + surname  
    else  
        forename + " " + surname  
}
```

When calling it, we may need to name default values; for example:

```
new Customer("Bob", "J", "Smith")
```

"Bob", "J", "Smith" is ok, but if we skip the `initial` variable, we'd need to name the `surname` variable like this:

```
new Customer("Bob", surname = "Smith")
```

Singleton Objects

In Java you can enforce a single instance of a class using the singleton pattern. Scala has made this idea as a feature of the language: as well as classes, you can define (singleton) *objects*.

The downside is that when we talk about “objects” in Scala, we’re overloading the term. We might mean an instance of a class (for example, a new `ShoppingCart()`, of which there could be many) or we might mean the one and only instance of a class; that is, a singleton object.

A typical use-case for a singleton in Java is if we need to use a single logger instance across an entire application.

```
// java  
Logger.getLogger("example").log(INFO, "Everything is fine.");
```

We might implement the singleton like this:

```
// java
public final class Logger {

    private static final Logger INSTANCE = new Logger();

    private Logger() { }

    public static Logger getLogger() {
        return INSTANCE;
    }

    public void log(Level level, String string) {
        System.out.printf("%s %s%n", level, string);
    }
}
```

We create a `Logger` class, and a single static instance of it. We prevent anyone else creating one by using a private constructor. We then create an accessor to the static instance, and finally give it a rudimentary log method. We'd call it like this:

```
// java
Logger.getLogger().log(INFO, "Singleton loggers say YEAH!");
```

A more concise way to achieve the same thing in Java would be to use an enum.

```
// java
public enum LoggerEnum {

    LOGGER;

    public void log(Level level, String string) {
        System.out.printf("%s %s%n", level, string);
    }
}
```

We don't need to use an accessor method; Java ensures a single instance is used and we'd call it like this:

```
// java
LOGGER.log(INFO, "An alternative example using an enum");
```

Either way, they prevent clients newing up an instance of the class and provide a single, global instance for use.

The Scala equivalent would look like this:

```
// scala
object Logger {
    def log(level: Level, string: String) {
        printf("%s %s%n", level, string)
    }
}
```

The thing to notice here is that the singleton instance is denoted by the `object` keyword rather than `class`. So we're saying "define a single object called `Logger`" rather than "define a class".

Under the covers, Scala is creating basically the same Java code as our singleton pattern example. You can see this when we decompile it.

```
1 // decompiled from scala to java
2 public final class Logger$ {
3     public static final Logger$ MODULE$;
4
5     public static {
6         new scala.demo.singleton.Logger$();
7     }
8
9     public void log(Level level, String string) {
10         Predef.MODULE$.printf("%s %s\n", (Seq)Predef.MODULE$
11             .genericWrapArray((Object)new Object[]{level, string}));
12     }
13
14     private Logger$() {
15         Logger$.MODULE$ = this;
16     }
17 }
```

There are some oddities in the `log` method, but that's the decompiler struggling to decompile the bytecode, and generally how Scala goes about things. In essence though, it's equivalent; there's a private constructor like the Java version, and a single static instance of the object. The class itself is even `final`.

There's no need to new up a new `Logger`; `Logger` is already an object, so we can refer to it directly. In fact, you couldn't new one up if you wanted to, because there's no class definition and so no class to new up.

Incidentally, you replicate Java's static `main` method by adding a `main` method to a Scala singleton object, not a class.

Companion Objects

You can combine objects and classes in Scala. When you create a class and an object with the same name in the same source file, the *object* is known as a *companion object*.

Scala doesn't have a `static` keyword but members of singleton objects are effectively static. Remember that a Scala singleton object is just that, a singleton. Any members

it contains will therefore be reused by all clients using the object; they're globally available just like statics.

You use companion objects where you would mix statics and non-statics in Java.

The Java version of `Customer` has fields for the customer's name and address, and an ID to identify the customer uniquely.

```
// java
public class Customer {

    private final String name;
    private final String address;

    private Integer id;

    public Customer(String name, String address) {
        this.name = name;
        this.address = address;
    }
}
```

Now we may want to create a helper method to create the next ID in a sequence. To do that globally, we create a static field to capture a value for the ID and a method to return and increment it. We can then just call the method on construction of a new instance, assigning its ID to the freshly incremented global ID.

```
// java
public class Customer {

    private static Integer sequenceOfIds;

    private final String name;
    private final String address;

    private Integer id;

    public Customer(String name, String address) {
```

```
        this.name = name;
        this.address = address;
        this.id = Customer.nextId();
    }

    private static Integer nextId() {
        return sequenceOfIds++;
    }
}
```

It's static because we want to share its implementation among all instances to create unique IDs for each.

In Scala, we'd separate the static from non-static members and put the statics in the singleton object and the rest in the class. The singleton object is the *companion object* to Customer.

We create our class with the two required fields and in the singleton object, create the nextId method. Next we create a private var to capture the current value, assigning it the value of zero so Scala can infer the type as an Integer. Adding a val here means no setter will be generated, and adding the private modifier means the generated getter will be private. We finish off by implementing the increment in the nextId method and calling it from the primary constructor.

```
// scala
class Customer(val name: String, val address: String) {
    private val id = Customer.nextId()
}

object Customer {
    private var sequenceOfIds = 0

    private def nextId(): Integer = {
        sequenceOfIds += 1
        sequenceOfIds
    }
}
```

The singleton object is a *companion object* because it has the same name and lives in the same source file as its class. This means the two have a special relationship and can access each other's private members. That's how the `Customer` object can define the `nextId` method as private but the `Customer` class can still access it.

If you were to name the object differently, you wouldn't have this special relationship and wouldn't be able to call the method. For example, the class `CustomerX` object below is not a companion object to `Customer` and so can't see the private `nextId` method.

```
// scala
class Customer(val name: String, val address: String) {
  private val id = CustomerX.nextId()           // compiler failure
}

object CustomerX {
  private var sequenceOfIds = 0

  private def nextId(): Integer = {
    sequenceOfIds += 1
    sequenceOfIds
  }
}
```

Other Uses for Companion Objects

When methods don't depend on any of the fields in a class, you can more accurately think of them as functions. Functions generally belong in a singleton object rather than a class, so one example of when to use companion objects is when you want to distinguish between functions and methods, but keep related functions close to the class they relate to.

Another reason to use a companion object is for factory-style methods — methods that create instances of the class companion. For example, you might want to create a factory method that creates an instance of your class but with less noise. If we want to create a factory for `Customer`, we can do so like this:


```
// scala
class Customer(val name: String, val address: String) {
    val id = Customer.nextId()
}

object Customer {
    def apply(name: String, address: String) = new Customer(name, address)
}
```

The apply method affords a shorthand notation for a class or object. It's kind of like the default method for a class, so if you don't call a method directly on an instance, but instead match the arguments of an apply method, it'll call it for you. For example, you can call:

```
Customer.apply("Bob Fossil", "1 London Road")
```

...or you can drop the apply and Scala will look for an apply method that matches your argument. The two are identical.

```
Customer("Bob Fossil", "1 London Road")
```

You can still construct a class using the primary constructor and new, but implementing the companion class apply method as a factory means you can be more concise if you have to create a lot of objects.

You can even force clients to use your factory method rather than the constructor by making the primary constructor private.

```
class Customer private (val name: String, val address: String) {
    val id = Customer.nextId()
}
```

The Java analog would have a static factory method, for example createCustomer, and a private constructor ensuring everyone is forced to use the factory method.

```
// java
public class Customer {

    private static Integer sequenceOfIds;

    private final String name;
    private final String address;

    private Integer id;

    public static Customer createCustomer(String name, String address) {
        return new Customer(name, address);
    }

    private Customer(String name, String address) {
        this.name = name;
        this.address = address;
        this.id = Customer.nextId();
    }

    private static Integer nextId() {
        return sequenceOfIds++;
    }

}
```