Denis Kalinin

Modern Web Development with Scala

Modern Web Development with Scala

A concise step-by-step guide to the Scala ecosystem

Denis Kalinin

This book is for sale at http://leanpub.com/modern-web-development-with-scala

This version was published on 2017-01-28

Contents

Preface

As a language, Scala emerged in 2004, but at that time it was known only to a limited circle of computer scientists and enthusiasts. A couple of years passed, and the books started to appear. These were books that concentrated mostly on Scala syntax and possibly the standard library. Then, in mid-2008, Twitter rewrote some of its core functionality in Scala. Foursquare, LinkedIn and Tumblr soon followed. All of a sudden, people realized that Scala is not a toy for enthusiasts, but a technology that can benefit today's businesses by enabling them to build high-quality and performant software.

Fast-forward to 2016. Lots of libraries, tools, frameworks written in Scala have appeared. People are not surprised anymore when they see another Scala success story on InfoQ. Recruiters and companies are actively looking for Scala developers, and many big data start-ups from Silicon Valley don't even question the choice of the language and select Scala by default.

And yet, the majority of public appears to be living in 2005. It sounds crazy, but people still perceive Scala as one of many niche languages and express doubts about its *future*. As someone who spent last two years actively using Play and Akka for building Web apps, I can say that this future is already here. In fact, it's been here at least since the Scala 2.9 release in 2011. While Java stays a high-performant but relatively low-level language, Scala allows you to use very high-level syntax constructs without compromising on performance.

I decided to write this book to show how Scala is *used*. It is not about syntax or functional programming per se but about applying Scala for building real-world Web applications. First, we will learn just enough basics to get started. Then, we will learn some functional programming concepts that are used in the rest of the book. Finally, we will look at build tools and start using Play framework for developing a simple Web app. I'm not going to overwhelm you with sophisticated one-page long code samples but rather show a little bit of everything. After reading this book, you will be familiar with most aspects of Web development in Scala from database access to user authentication. You will also get a pretty good understanding how to integrate Play with modern frontend tools such as Webpack and React.

This book is intended for people who only want to start writing Web apps in Scala, so I don't expect any prior Scala experience. However, I do expect that the reader is familiar with basic Web concepts such as HTML, CSS, JavaScript and JSON. I also assume that the reader knows well at least one modern programming language like Java, C#, Ruby or Python.

The book should be read from start to finish, because later more advanced topics are always based on simpler ones explained earlier. The reader is welcome to try any examples and follow the app development, but it should also be possible to follow the narrative without a computer.

Let's get started!

Before we begin

If you want to simply read a book in one go without trying to roll out an app of your own, you don't need any preparations. However, even if this is the case, I would still recommend flicking through this section because it provides a good overview of what tools you can use to write Scala code and how to set them up.

Setting up your environment

I assume that you are already familiar with your operating system and its basic commands. I personally prefer using Ubuntu for Web development, and if you want to give it a try, you may consider installing VirtualBox and then installing Ubuntu on top of it. If you want to follow this path, here are my suggestions:

- Download VirtualBox from https://www.virtualbox.org/[1] and install it
- Create a new virtual machine, give it about 2GB of RAM and attach 16GB of HDD (dynamically allocated storage)
- Download a lightweight Ubuntu distribution such as Xubuntu or Lubuntu. I wouldn't bother with 64bit images, plain i386 ones will work just fine. As for versions, 15.10 definitely works in latest versions of VirtualBox without problems, 15.04 probably will too
- After installing Ubuntu, install Guest Additions so that you will be able to use higher resolutions on a virtual machine

By the way, according to StackOverflow surveys[2] about 20% of developers consistently choose Linux as their primary desktop operating system.

Installing Java Development Kit

Scala compiles into Java bytecode, so you will need to install the Java Development Kit (JDK) to run Scala programs. Fortunately, installing JDK is straightforward, just follow these steps:

[1] https://www.virtualbox.org/wiki/Downloads

[2] http://stackoverflow.com/research/developer-survey-2016#technology-desktop-operating-system

- Go to http://www.oracle.com/technetwork/java/javase/downloads/index.html[3] and download the latest JDK 8 (the full name will look something like "8u51", which means "version 8, update 51"). If you are on Linux, I recommend downloading a tar.gz file
- On Windows, just follow the instructions of the installer. On Linux, extract the contents of the archive anywhere (for example, to `~/DevTools/java8`)
- Add the `bin` directory of JDK to your `PATH` so that the following command works from any directory on your machine:

```
$ java -version
java version "1.8.0_51"
Java(TM) SE Runtime Environment (build 1.8.0_51-b16)
Java HotSpot(TM) 64-Bit Server VM (build 25.51-b03, mixed mode)
```

- Create an environment variable `JAVA_HOME` and point it to the JDK directory. On Linux, you can accomplish both goals if you add to your `~/.bashrc` the following:

```
JAVA_HOME=/home/user/DevTools/jdk1.8.0_51
PATH=$JAVA_HOME/bin:$PATH
```

Strictly speaking, Scala 2.11 will work perfectly fine with Java 7. However, Play starting with version 2.4 supports only Java 8, so we will stick to this version.

Installing Scala

You don't need the Scala distribution in order to write Play applications, but you may want to install it to try some ideas in the interpreter. Just follow these steps:

- Get Scala binaries from http://www.scala-lang.org/download/[4]
- Extract the contents of the archive anywhere (for example, to `~/DevTools/scala`)
- Add the `bin` directory of the Scala distribution to your path:

[3] http://www.oracle.com/technetwork/java/javase/downloads/index.html
[4] http://www.scala-lang.org/download/

```
JAVA_HOME=/home/user/DevTools/jdk1.8.0_51
SCALA_HOME=/home/user/DevTools/scala-2.11.0
PATH=$JAVA_HOME/bin:$SCALA_HOME/bin:$PATH
```

- Start the interpreter in the interactive mode by typing `scala`

Using REPL

When you type `scala` without arguments in the command line, Scala will start the interpreter in the interactive mode and greet you with a REPL prompt. REPL stands for Read-Eval-Print loop, and it is a great way to learn new features or try out some ideas. You can start by typing

```
scala> println("Hello Scala")
```

and then you can press `Enter` and observe the output.

```
Hello Scala
```

Since the `println` function prints a message in `stdout`, and the REPL session is hooked to `stdout`, you see the message in the console. Another option is to simply type

```
scala> "Hello World"
```

The REPL will respond by printing

```
res1: String = Hello World
```

in the console. The REPL evaluates an expression, determines its type and, if you haven't assigned it to anything, assigns it to an automatically-generated value.

By the way, The Scala Worksheet plugin in Atom works by intercepting `stdout` of the REPL and presenting its output as a comment in the editor area.

Feel free to come back to REPL later, when you learn more about the language, but for now here is the list of commands that you may find particularly useful:

command	description
`:quit`	exits the REPL
`:reset`	resets the REPL to its initial state
`:cp`	adds a specified library to the classpath
`:paste`	enters the `paste` mode, which makes it possible to define multiple things at once
`:history`	shows the list of previously typed expressions
`:help`	shows the list of available commands

Using IntelliJ IDEA

IntelliJ IDEA[5] accompanied with the official Scala plugin is the best way to write Scala code in 2016. And the good news is that the Community edition, which is free and open-source, will work just fine.

When installing IntelliJ there are several not-so-obvious things worth mentioning.

First, Windows versions come with an embedded JRE, while Linux versions use an already installed JDK, so if you have Java 8 installed, you may see messages like this when IDEA starts:

```
Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=350m; sup\
port was removed in 8.0
```

You can safely ignore these warnings for now. By the way, you can use the `idea.vmoptions` file that resides in the `bin` directory to tweak, well, VM options. Two properties which are often tweaked are:

```
-Xms128m
-Xmx2048m
```

Here `-Xms` specifies the size of the initial memory pool, and `-Xmx` specifies the maximum. And if you want to learn more about them, start with this question on StackOverflow[6].

Sometimes IntelliJ goes crazy and starts highlighting valid parts of your code as errors. When it happens, there are two ways to fix the IDE. The simplest is to deactivate "Scala type-aware highlighting" and then activate it again. This helps in many cases.

Scala type-aware highlighting toggle

A more radical solution is to click "File -> Invalidate Caches / Restart...". IDEA will have to reindex everything after restart, but sometimes it could be the only way to regain its senses.

[5] https://www.jetbrains.com/idea/

[6] http://stackoverflow.com/questions/14763079/

Using Atom Editor

Even though I strongly believe that using a full-blown IDE when working with Scala significantly increases your productivity, it's also worth mentioning a slightly more lightweight alternative - the Atom Editor. Atom is created by GitHub, based on such technologies as V8 and React, and at the moment, it's becoming more and more popular.

The official site is atom.io[7], but if you're using Ubuntu, you may want to install it from a PPA repository. Just add `ppa:webupd8team/atom` to the list of repositories via "Software & Updates" or, if you prefer the command line, simply type:

```
$ sudo add-apt-repository ppa:webupd8team/atom
$ sudo apt-get update
$ sudo apt-get install atom
```

There are many packages written for Atom, and for trying examples from this book you may want to install the following:

plugin	description
`language-scala`	adds syntax highlighting for Scala
`linter-scalac`	compiles your code behind the scenes using `scalac` to show compiler errors right inside the editor
`terminal-plus`	allows to use the terminal session without leaving the editor
`scala-worksheet-plus`	enables the worksheet mode, which executes your code in the Scala interpreter behind the scenes and shows results as comments

Just go to "Packages -> Settings View -> Install Packages/Themes".

Why use Atom Editor?

Atom is based on some relatively heavy technologies, so it's not as lightweight as, say, Notepad++[8]. Moreover, it doesn't help you much when it comes to adding package imports, and in this respect it is inferior to tools like IntelliJ IDEA, so why use it at all?

Well, there are several situations when you may actually prefer Atom to IntelliJ:

- When you browse the contents of someone else's project, often you want to get the overview of its directory structure and possibly quickly edit some files. In this case, you could simply type "`atom .`" and start hacking in a couple of seconds. IntelliJ would take at least half a

[7] http://atom.io
[8] https://notepad-plus-plus.org/

minute to load, and then it would require you to import the project, which may take a couple of minutes even on decent hardware.

- Everything related to frontend is usually top quality in Atom. At the same time, full support of JavaScript and CSS is considered a paid feature in IntelliJ. The Community version will paint HTML markup in different colors and even highlight JS files, but if you want support for JSX templates or CSS preprocessors, you should start looking at the Ultimate version or switch to Atom.
- Lastly, I found that when you're only starting with something new, learning the basics without using sophisticated tools provides a better understanding of technology and therefore, gives you more confidence later.

I suggest you use Atom for the first parts of the book and then switch to IntelliJ for working with Play. Please refer to Appendix B for additional instructions on importing Scala projects in IntelliJ.

Language fundamentals

In this section, I will demonstrate the basics of the Scala programming language. This includes things that are common to most programming languages as well as several features which are specific to Scala. I'm going to provide the output of the interpreter here, but feel free to start you own REPL session and try it yourself.

Defining values

There are three main keywords for defining everything in Scala:

keyword	description
`val`	defines a constant (or value)
`var`	defines a variable, very rarely used
`def`	defines a method

By the way, there are several ways to define a function in Scala, and using `def` is only one of them. Read on!

Defining a constant is usually as simple as typing

```
scala> val num = 42
num: Int = 42
```

Notice that Scala guessed the type of the constant by analyzing its value. This feature is called *type inference*.

Sometimes you want to specify the type explicitly. If, in the example, above you want to end up with a value of type `Short`, simply type

```
scala> val num: Short = 42
num: Short = 42
```

Variables are rarely needed in Scala, but they are supported by the language and could be useful sometimes:

```
scala> var hello = "Hello"
hello: String = Hello
```

Just as with types, you may explicitly specify the type or allow the type inferrer to do its work.

As we will see later, Scala mostly uses expressions instead of statements to alter the control flow. Since expressions always evaluate to some value, you can write code using `val`s and resort to `var`s only occasionally.

When defining methods, it is required to specify argument types. Specifying the return type is recommended but optional:

```
scala> def greet(name: String) = "Hello" + name
greet: (name: String)String
```

It's worth mentioning that the body doesn't require curly braces and could be placed on the same line as the signature. At the same time, relying on the type inferrer for guessing the return type is not recommended for a number of reasons:

- If you rely on the type inferrer, it basically means that the return type of your function depends on its implementation, which is dangerous as implementation is often changed
- The readers of your code don't want to spend time guessing the resulting type

These points are particularly important when you work on public APIs.

The return type can be specified after the parentheses:

```
scala> def greet(name: String): String = "Hello " + name
greet: (name: String)String
```

The value that is returned from the method is the last expression of its body, so in Scala you don't need to use the `return` keyword. Remember, however, that the assignment operator doesn't return a value.

If you want to create a procedure-like method that doesn't return anything and is only invoked for side effects, you should specify the return type as `Unit`. This is a Scala replacement for `void` (which, by the way, is not even a keyword in Scala):

```
scala> def greet2(): Unit = println("Hello World")
greet2: ()Unit
```

Not so long ago, the official documentation recommended omitting the equals sign when defining procedure-like methods. Not anymore! Starting with version 2.10, the recommendation is to use the equals sign, always. Check this StackOverflow question[9] or the docs[10].

It's also possible to define a method without parentheses, in which case it is called a *parameterless* method:

```
scala> def id = Math.random
id: Double

scala> id
res10: Double = 0.15449866168176285
```

This makes the invocation of such a method look like accessing a variable/constant and supports the uniform access principle[11]. If you want to define your method like this, you should ensure that it doesn't have side effects, otherwise it will be extremely confusing for users.

Functional types

It's possible to define a function and assign it to a variable (or constant). For example, the `greet` method from the example above could also be defined in the following way:

```
scala> var greetVar: String => String = (name) => "Hello " + name
greetVar: String => String = <function1>
```

As the REPL output shows, `String => String` is the type of our function. It can be read as *a function from String to String*. Again, you don't have to type the return type, but if you want the type inferrer to do the work, you'll have to specify the type of parameters:

```
scala> var greetVar = (name: String) => "Hello " + name
greetVar: String => String = <function1>
```

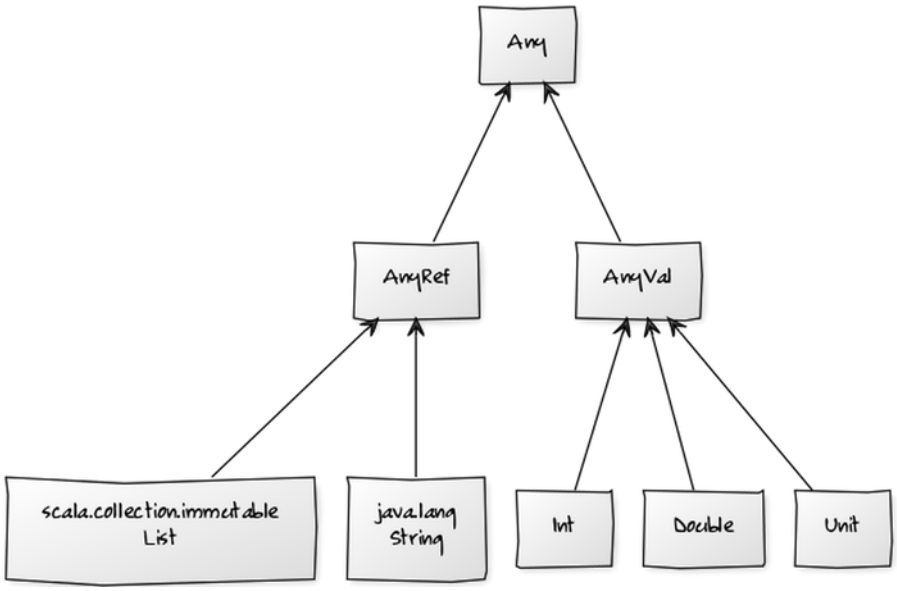
Assigning methods to variables is also possible, but it requires a special syntax to tell the compiler that you want to *reference* the function rather than *call* it:

[9] http://stackoverflow.com/questions/944111/
[10] http://docs.scala-lang.org/style/declarations.html#procedure_syntax
[11] http://stackoverflow.com/questions/7600910/

```
scala> val gr = greet _
gr: String => String = <function1>
```

Type hierarchy

Since we are on the topic of types, let's take a look at a very simplified type hierarchy:



Scala type hierarchy - top

Unlike Java, literally everything has its place here. Primitive types inherit from `AnyVal`, reference types (including user-defined classes) inherit from `AnyRef`. The absence of value has its own type `Unit`, which belongs to the primitives group. Even functions have their place in this hierarchy: for example, a function from String to String has type `Function1[String, String]`:

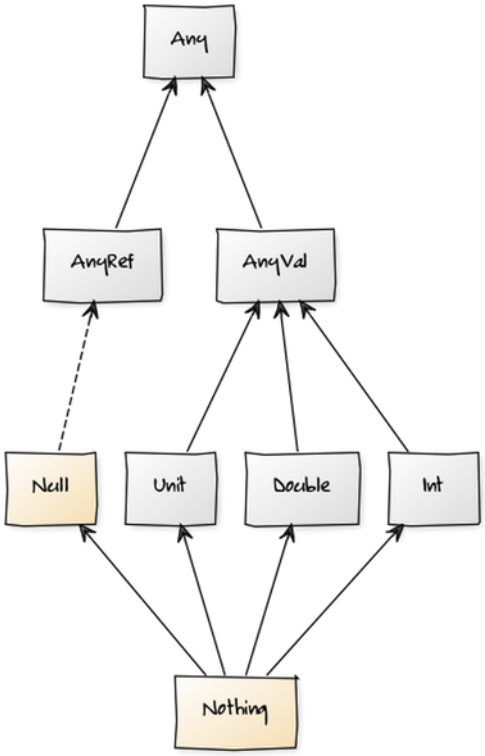
```
scala> val greetFn = (name: String) => "Hello " + name
greetFn: String => String = <function1>

scala> val fn: Function1[String, String] = greetVar
fn: String => String = <function1>
```

You can use `isInstanceOf` to check the type of a value:

```
scala> val str = "Hello"
str: String = Hello

scala> str.isInstanceOf[String]
res20: Boolean = true

scala> str.isInstanceOf[AnyRef]
res21: Boolean = true
```

By the way, another name of `scala.AnyRef` is `java.lang.Object`, so `str.isInstanceOf[Object]` returns `true`

Scala also introduces the concept of so-called *bottom types* which implicitly inherit from all other types:



Scala type hierarchy - bottom types

In the diagram above, `Null` implicitly inherits from all reference types (including user-defined ones, of course) and `Nothing` from all types including primitives and `Null` itself. You are unlikely to use bottom types directly in your programs, but they are useful to understand type inference. More on this later.

Collections

Unlike Java, Scala uses the same uniform syntax for creating and accessing collections. For example, you can use the following code to instantiate a new list and then get its first element:

```
scala> val list = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> list(0)
res24: Int = 1
```

For comparison, here is the same code that uses an array instead of a list:

```
scala> val array = Array(1,2,3,4)
array: Array[Int] = Array(1, 2, 3, 4)

scala> array(0)
res25: Int = 1
```

For performance reasons, Scala will map the latter collection to a Java array. It happens behind the scenes and is completely transparent for a developer.

The collections API was completely rewritten in Scala 2.8, which, in turn, was released in 2009. Even though many regard this rewrite as a significant improvement, some people were quite skeptical back then[12].

Another important point is that Scala collections always have a type. If the type wasn't specified, it will be inferred by analyzing provided elements. In the example above we ended up with the list and array of `Int`s because the elements of these collections looked like integers. If we wanted to have, say, a list of `Short`s, we would have to set the type explicitly like so:

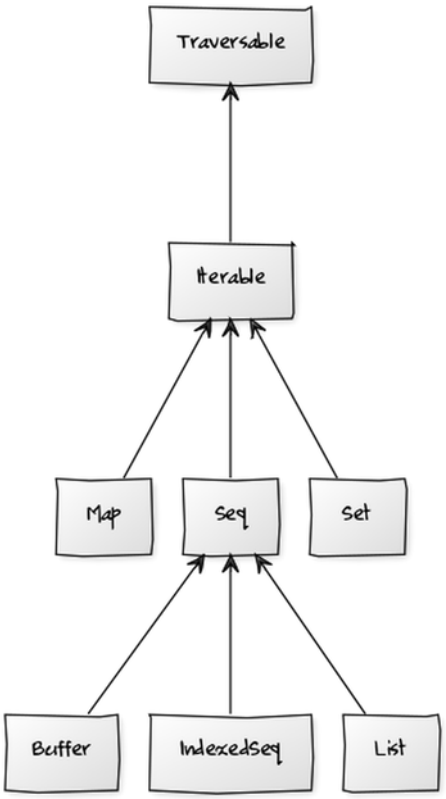
```
scala> val list = List[Short](1,2,3,4)
list: List[Short] = List(1, 2, 3, 4)
```

If the type inferrer cannot determine the type of the collection, compilation will fail:

[12] http://stackoverflow.com/questions/1722726/

```
scala> var list: List = null
<console>:7: error: type List takes type parameters
       var list: List = null
                 ^
```

A greatly simplified Collections hierarchy is shown below:



Collections Hierarchy

It's worth mentioning that most commonly used methods are already defined on very basic collection types. In practice, this means that when you need a collection, usually you can simply use `Seq`, and its API will probably be sufficient most of the time.

Collections also come in two flavours - mutable and immutable. Scala uses the immutable flavour by default, so when, in the previous example, we typed `List`, we actually ended up with an immutable `List`:

```
scala> list.isInstanceOf[scala.collection.immutable.List[Short]]
res29: Boolean = true
```

Immutable collections don't have methods for altering the original collection, but they have methods that return new collections. For example, the `:+` method returns a new list that contains all the elements of the original list and one new element:

```
scala> val list = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> list :+ 5
res33: List[Int] = List(1, 2, 3, 4, 5)
```

Scala imposes very few restrictions on function names, so it is perfectly acceptable to create a function called ":+". In addition, if the function has only one parameter, it is possible to omit the dot and parentheses and make a method invocation look like using an operator. This approach is widely used in the standard library, so when you use the mathematical plus operator to add two integers, you actually use a method called "+" defined on `Int`. Therefore, in Scala `2 + 2` can also be written as `2.+(2)` and vice versa.

Mutable collections can alter themselves. A great example of a mutable list is `ListBuffer`, which is often used to accumulate values and then build an immutable list from these values:

```
scala> val buffer = scala.collection.mutable.ListBuffer.empty[Int]
buffer: scala.collection.mutable.ListBuffer[Int] = ListBuffer()

scala> buffer += 1
res35: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1)

scala> buffer += 2
res36: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2)

scala> buffer.toList
res37: List[Int] = List(1, 2)
```

This is similar to using `StringBuilder` for constructing strings in Java.

Note how array and list elements are accessed by using parentheses. Note also that both collections were created without using the `new` keyword. This trick is not collection-specific, and we will return to this topic after discussing classes and objects.

Packages and imports

Scala classes are organized into packages similarly to Java or C#. For example, the `ListBuffer` type was defined in a package called `scala.collection.mutable` so in order to use it in your program you need to either always type a so-called fully qualified name[13] (FQN for short) or import the class:

[13] https://en.wikipedia.org/wiki/Fully_qualified_name

```
scala> import scala.collection.mutable.ListBuffer
import scala.collection.mutable.ListBuffer

scala> val buffer = ListBuffer.empty[Int]
buffer: scala.collection.mutable.ListBuffer[Int] = ListBuffer()
```

Just like in Java or C#, we can import all classes from a particular package using a wildcard. The difference is that Scala uses _ instead of *:

```
scala> import scala.collection.mutable._
import scala.collection.mutable._

scala> val set = HashSet[Int]()
set: scala.collection.mutable.HashSet[Int] = Set()
```

In addition to the imports defined by a developer, Scala automatically imports the following:

import	description
`java.lang._`	which includes `String`, `Exception` etc.
`scala._`	which includes `Option`, `Any`, `AnyRef`, `Array` etc.
`scala.Predef`	which conveniently introduces aliases for commonly used types and functions such as `List`, `Seq`, `println` etc.

Imports in Scala are not restricted to the beginning of the file and can appear inside functions or classes.

Defining classes

Users can define their own types using the omnipresent `class` keyword:

```
scala> class Person {}
defined class Person
```

Needless to say, the `Person` class defined above is completely useless. You can create an instance of this class, but that's about it:

```
scala> val person = new Person
person: Person = Person@794eeaf8
```

Note that the REPL showed `Person@794eeaf8` as a value of this instance. Why is that? Well, in Scala all user-defined classes implicitly extend `AnyRef`, which is the same as `java.lang.Object`. The `java.lang.Object` class defines a number of methods including `toString`:

```
def toString(): String
```

By convention, the `toString` method is used to create a String representation of the object, so it makes sense that REPL uses this method here. The base implementation of the `toString` method simply returns the name of the class followed by an object hashcode.

By the way, if you don't have anything inside the body of your class, you don't need curly braces, so `class Person` is a perfectly acceptable class declaration.

A slightly better version of this class can be defined as follows:

```
scala> class Person(name: String)
defined class Person
```

By putting `name: String` inside the parentheses we defined a constructor with a parameter of type `String`. Although it is possible to use `name` inside the class body, the class still doesn't have a field to store it. Fortunately, it's easily fixable:

```
scala> class Person(val name: String) {}
defined class Person
```

Much better! Adding `val` in front of the parameter name defines an immutable field. Similarly `var` defines a mutable field. In any case, this field will be initialized with the value passed to the constructor when the object is instantiated. What's interesting, the class defined above is roughly equivalent of the following Java code:

```
1  class Person {
2    private final String mName;
3    public Person(String name) {
4      mName = name;
5    }
6    public String name() {
7      return mName;
8    }
9  }
```

To define a method, simply use the `def` keyword inside a class:

```
class Person(val name: String) {
  override def toString = "Person(" + name + ")"
  def apply(): String = name
  def sayHello(): String = "Hi! I'm " + name
}
```

When working with classes and objects in REPL, it is often necessary to type several lines of code at once. Remember that REPL provides the paste mode, which can be activated by the `:paste` command.

In the example above we defined two methods - `apply` and `sayHello` and overrode (thus keyword `override`) existing method `toString`. Of these three, the simplest one is `sayHello` as it only prints a phrase to `stdout`:

```
scala> val joe = new Person("Joe")
joe: Person = Person(Joe)

scala> joe.sayHello()
res49: String = Hi! I'm Joe
```

Notice that REPL used our `toString` implementation to print information about the instance to `stdout`. As for the `apply` method, there are two ways to invoke it.

```
scala> joe.apply()
res50: String = Joe

scala> joe()
res51: String = Joe
```

Yep, using parentheses on the instance of a class actually calls the `apply` method defined on this class. This approach is widely used in the standard library as well as in third-party libraries.

Defining objects

Scala doesn't have the `static` keyword, but it does have syntax for defining *singletons*[14]. If you need to define methods or values that can be accessed on a type rather than an instance, use the `object` keyword:

[14] https://en.wikipedia.org/wiki/Singleton_pattern

```
object RandomUtils {
  def random100 = Math.round(Math.random * 100)
}
```

After `RandomUtils` is defined this way, you will be able to use method `random100` without creating any instances of the class:

```
scala> RandomUtils.random100
res62: Long = 67
```

You can define the `apply` method on an object and then call it using the name of the object followed by parentheses. One common pattern is to define an object that has the same name as the original class and define the `apply` method with the same constructor parameters as the original class:

```
class Person(val name: String) {
  override def toString = "Person(" + name + ")"
}
object Person {
  def apply(name: String): Person = new Person(name)
}
```

If an object has the same name as a class, then it's called a *companion object*. Companion objects are often used in Scala for defining additional methods and implicit values. You will see a concrete example when we get to serializing objects into JSON.

Now you can use the `apply` method on the object to create instances of class Person:

```
scala> val person = Person("Joe")
person: Person = Person(Joe)
```

Essentially, this eliminates the need for using the `new` keyword. Again, this approach is widely used in the standard library and in third-party libraries. In later chapters, I usually refer to this syntax as calling the *constructor* even though it is actually calling the `apply` method on a companion object.

Type parametrization

Classes can be parametrized, which makes them more generic and possibly more useful:

```
class Cell[T](val contents: T) {
  def get: T = contents
}
```

We defined a class called `Cell` and specified one field `contents`. The type of this field is not yet known. Scala doesn't allow *raw types*, so you cannot simply create a new `Cell`, you must create a `Cell` *of something*. Fortunately, the type inferrer can help with that:

```
scala> new Cell(1)
res71: Cell[Int] = Cell@1c0d3eb6
```

Here we passed `1` as an argument, and it was enough for the type inferrer to decide on the type of this instance. We could also specify the type ourselves:

```
scala> new Cell[Short](1)
res73: Cell[Short] = Cell@751fa7a3
```

Collections syntax revisited

Now it's becoming more and more clear that there is absolutely nothing in Scala syntax that is specific to collections. In fact, now we can explain what happens behind the scenes when we are creating and accessing `Lists`:

```
scala> val list = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> list(0)
res24: Int = 1
```

First, we can conclude that `List` is a parametrized class that has the `apply` method. This method accepts an index as an argument and returns the element that has this index. Second, there is also an *object* called `List` which has the `apply` method. This method probably calls the `List` constructor to instantiate the actual object. Finally, the `toString` method is overridden to return the word List and its elements in parentheses.

This is actually a recurring idea in Scala design. Rather than introduce a new feature to support a particular use case, Scala always tries to solve a particular problem using a more universal or generic approach.

Functions and implicits

A parameter of a function can have a default value. If this is the case, users can call the function without providing the value for the argument:

```
scala> def greet(name: String = "User") = "Hello " + name
greet: (name: String)String

scala> greet()
res77: String = Hello User
```

Compiler *sees* that the argument is absent, but it also *knows* that there is a default value, so it takes this value and then invokes the function as usual.

We can go even further and declare the `name` parameter as *implicit*:

```
scala> def greet(implicit name: String) = "Hello " + name
greet: (implicit name: String)String
```

Here we're not setting any default values in the function signature. As a result, in the absence of the argument, Scala will look for a value of type `String` marked as `implicit` and defined somewhere in scope. So, if we try to call this function immediately, it will not work:

```
scala> greet
<console>:9: error: could not find implicit value for parameter name: String
               greet
               ^
```

However, after defining an implicit value with a type of `String`, it will:

```
scala> implicit val n = "User"
n: String = User

scala> greet
res79: String = Hello User
```

This code works because there is exactly one value of type `String` marked as `implicit` and defined in the scope. If there were several `implicit Strings`, the compilation would fail due to ambiguity.

Implicits and companion objects

You can define your own `implicit` values or introduce already defined `implicits` into scope by means of `imports`. However, there is one more place where Scala will look for an `implicit` value when it needs one. This is the companion object of the parameter type. For example, when we are defining a new class called `Person`, we may decide to create a default value and put it into the companion object:

```
class Person(val name: String)

object Person {
  implicit val person: Person = new Person("User")
}
```

Let's also create a method with one parameter of type `Person` and mark it as `implicit`:

```
def sayHello(implicit person: Person): String = "Hello " + person.name
```

Even if we don't create any instances of class `Person` in the scope, we will still be able to use the `sayHello` method without providing any arguments:

```
scala> sayHello
res0: String = Hello User
```

This works because companion objects are another place where the compiler looks for implicits. The point here is that the type of the method parameter is the same as the name of the companion object.

Of course, we can always pass a regular argument *explicitly*:

```
scala> sayHello(new Person("Joe"))
res1: String = Hello Joe
```

In this case, the explicitly passed parameter takes precedence over whatever implicit parameter was defined. Remember that the compiler only starts looking for implicits if it cannot validate code using regular arguments.

Some people argue that implicits make code unmaintainable. In my experience, however, it has never been the case. In fact, most professional Scala developers agree that implicits actually make code clearer.

Loops and conditionals

Unlike Java with its *if-statements*, in Scala *if-expressions* always result in a value. In this respect, they are similar to Java's ternary operator `?:`.

Let's define a function that uses the if expression to determine whether the argument is an even number:

```
1 def isEven(num: Int) = {
2   if (num % 2 == 0)
3     true
4   else
5     false
6 }
```

When this method is defined in REPL, the interpreter responds with `isEven: (num: Int)Boolean`. Even though we haven't specified the return type, the type inferrer determined it as the type of the last (and only) expression, which is the *if expression*. How did it do that? Well, by analyzing the types of both branches:

expression	type
if branch	`Boolean`
else branch	`Boolean`
whole expression	`Boolean`

If an argument is an even number, the *if branch* is chosen and the result type has type `Boolean`. If an argument is an odd number, the *else branch* is chosen but result still has type `Boolean`. So, the whole expression has type `Boolean` regardless of the "winning" branch, no rocket science here.

But what if branch types are different, for example `Int` and `Double`? In this case the nearest common supertype will be chosen. For `Int` and `Double` this supertype will be `AnyVal`, so `AnyVal` will be the type of whole expression.

If you give it some thought, it makes sense, because the result type must be able to hold values of both branches. After all, we don't know which one will be chosen until runtime.

Bottom types revisited

Now it's time to recall two types which reside at the bottom of the Scala type hierarchy - `Null` and `Nothing`.

There is only one object of type `Null` - `null` and its meaning is the same as it was in Java or C#: the object is not initialized. Any reference object can be `null`, so it's only logical that any reference type (i.e `AnyRef`) is a supertype of `Null`. This brings us to the following conclusion:

expression	type
if branch	`AnyRef`
else branch	`Null`
whole expression	`AnyRef`

In other words, if the *else branch* returns `null`, the *if branch* determines the result of the whole expression.

OK, what if the else branch never returns and instead, throws an exception? In this case, the type of the *else branch* is considered to be `Nothing`, and the whole expression will have the type of the *if branch*.

expression	type
if branch	`Any`
else branch	`Nothing`
whole expression	`Any`

There's not much you can do with bottom types, but they are included in Scala hierarchy, and they make the rules that the type inferrer uses more clear.

while loops

The `while` loop is almost a carbon copy of its Java counterpart, so in Scala it looks like an outcast. Instead of returning a value, it's called for a side effect. Moreover, it almost always utilizes a `var` for iterations:

```
var it = 5
while (it > 0) {
  print(it)
  it -= 1
}
// prints 54321
```

It feels almost *awkward* to use it in Scala, and, as a matter of fact, you almost never need to. We will look at some alternatives when we get to the Functional programming section, but for now, here is more Scala-like code that does essentially the same thing:

```
1.to(5).reverse.foreach { num => print(num) }
// prints 54321
```

String interpolation

String interpolation was one of the features introduced in Scala 2.10. It allows to execute Scala code inside string literals. In order to use it, simply put `s` in front of a string literal and `$` in front of any variable or value you want to interpolate:

```
val name = "Joe"
val greeting = s"Hello $name"
println(greeting)
// prints Hello Joe
```

If an interpolated expression is more complex, e.g contains dots or operators, it needs to be taken in curly braces:

```
val person = new Person("Joe")
val greeting = s"Hello ${person.name}"
println(greeting)
// prints Hello Joe
```

If you need to spread your string literal across multiple lines or include a lot of special characters without escaping you can use triple-quote to create raw strings:

```
val json = """
  {
    firstName: "Joe",
    lastName: "Black"
  }
"""
```

Note that the usual backslash escaping doesn't work there:

```
val str = """First line\nStill first line"""
println(str)
// prints First line\nStill first line
```

This makes sense because the whole idea of raw string is to treat characters for what they are.

Traits

Traits in Scala are similar to mixins in Ruby in a sense that you can use them to add functionality to your classes. Traits can contain both abstract (not implemented) and concrete (implemented) members. If you mix in traits with abstract members, you must either implement them or mark your class as `abstract`, so the Java rule about implementing interfaces still holds.

In Java, a class can implement many interfaces, but if you want to make your class concrete (i.e. allow to instantiate it), you need to provide implementations for all methods defined by all interfaces.

Unlike Java interfaces, though, Scala traits can and often do have concrete methods.

Let's look at a rather simplistic example:

```
trait A { def a(): Unit = println("a") }

trait B { def b(): Unit }
```

We defined two traits so that trait `A` has one concrete method, and trait `B` has one abstract method (the absence of a body that usually comes after the equals sign means exactly that). If we want to create a new class `C` that inherits functionality from both traits, we will have to implement the `b` method:

```
class C extends A with B { def b(): Unit = println("b") }
```

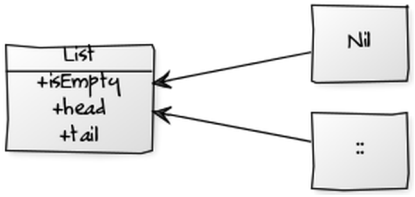
If we don't implement `b`, we will have to make the `C` class `abstract` or get an error.

Functional programming

In this section, we will examine a number of features which support functional programming and are available in Scala. As we will see, all of them are actively used by library designers and developers in real-world applications.

Algebraic data types

Even though it sounds like rocket science, *algebraic data types* (ADT) are actually pretty simple. An ADT is a type formed by combining other types. For example, `List` is an ADT, because it's formed by combining two other types - empty list (represented by singleton object `Nil`) and non-empty list (represented by `::` and pronounced *cons*):



List as an ADT

The important thing here is that the list may be either empty or non-empty, i.e `Nil` and `::` form disjoint sets. In addition to using the `List` object's `apply` method, you can use the following approach to construct a new list:

```
val list = 1 :: 2 :: 3 :: Nil
```

The `::` method is called `prepend` and it's available on all lists including the empty one. In Scala, if a method's name ends with a colon (`:`), the argument of this method (when the method invocation is written in the *operator style*, i.e, without dots and parentheses) must be placed on the left. With this little trick in mind, we can rewrite the previous example like so:

```
val list = Nil.::(3).::(2).::(1)
```

Basically, we start with the empty list `Nil` and then prepend the numbers so that the resulting list will be `List(1, 2, 3)`.

In order to determine the type of the list, we could use the `isInstanceOf` method available on all types:

```
list.isInstanceOf[List[Any]]    // true
list.isInstanceOf[Nil.type]     // false
list.isInstanceOf[::[Int]]      // true
```

Note that since Scala doesn't allow raw types, we had to specify `Any` and `Int` even though we weren't interested in such details. On the other hand, `Nil` is defined as an object (singleton) and represents the empty list regardless of the element type. Since `Nil` is a value, we need to add the `type` suffix to get the type back.

As Martin Odersky usually puts it, the `instanceOf` method is made verbose intentionally to discourage people from using it too often. A more Scala-like approach is to use *pattern matching*.

Pattern matching

You can think of pattern matching as Java's `switch` statement on steroids. The main differences are that pattern matching in Scala always returns a value and it is *much* more powerful.

```
list match {
  case Nil => println("empty")
  case ::(head, tail) => println("non-empty")
}
```

The first case block simply checks whether the `list` object is the `Nil` singleton object. The second one is more interesting, though. Not only does it check that the `list` has type `::`, it also extracts `head` (the first element) and `tail` (all elements except the first one).

Scala also allows to use *infix notation* for destructuring constructor arguments, so the previous example can also be written as follows:

```
list match {
  case Nil => println("empty")
  case head :: tail => println("non-empty")
}
```

Since it's not really our goal here, we're not going to get into mechanics that make pattern matching possible. However, it's worth mentioning that if you want your class to have similar capabilities, you must provide the `unapply` method on the companion object. Alternatively, you can simply turn your class into a `case class`, and then, the `unapply` method will be created automatically.

Case classes

Remember our `Person` class from the section about objects? Let's revisit that code here:

```
class Person(val name: String) {
  override def toString = "Person(" + name + ")"
}
object Person {
  def apply(name: String): Person = new Person(name)
}
```

Here we're creating a class, defining a field, overriding the `toString` method, defining a companion object. All of this could be achieved by the following line:

```
case class Person(name: String)
```

On top of the goodies mentioned above, this definition also does the following:

- overrides the `hashcode` method used by some collections from the standard library
- overrides the `equals` method, which makes two objects with the same set of values equal, so `new Person("Joe") == new Person("Joe")` will return `true`
- defines the `unapply` method used in pattern matching

If we create an instance of the `Person` class, we can pattern match against it:

```
val person = Person("Joe")

person match {
  case Person(name) => s"Hello $name"
  case _ => "Not a person!"
}
```

The _ symbol defines a so-called *catch-all* case that will be chosen if all previous cases failed to match the expression.

The `Person` type also received well-behaved `toString` and `equals` methods:

```
val person = Person("Joe")
println(person)
// prints Person(Joe)

val personClone = Person("Joe")
println(personClone == person)
// prints true
```

As a result, many developers use case classes to quickly get hassle-free types with `hashcode`, `equals` and `apply` methods even if they don't intend to use their classes in pattern matching.

Higher-order functions

Functions that accept other functions as their arguments are called *higher-order functions*[15]. They are supported by virtually any modern programming language including JavaScript, Ruby, Python, PHP, Java, C# and so on. Since they are such a common concept, which is probably already familiar to you, we're not going to spend much time discussing the theory behind higher-order functions. Instead, we will look at some Scala specifics.

Curly braces instead of parentheses

If your function accepts only one argument, then you are free to call your function using curly braces instead of parentheses:

```
scala> def greet(name: String) = "Hello " + name
greet: (name: String)String

scala> greet{"Joe"}
res74: String = Hello Joe
```

It may not be that useful when the only argument is a `String`, but what if the function accepts another function as the argument? If we had to always use parentheses, there would be too many of them:

```
scala> def invokeBlock(block: () => Unit): Unit = block()
invokeBlock: (block: () => Unit)Unit

scala> invokeBlock( () => println("Inside the block!") )
Inside the block!
```

But since the `invokeBlock` method accepts only one argument, we can use the following syntax:

```
1  invokeBlock { () =>
2    println("Inside the block!")
3  }
```

The block spreads to several lines, and using curly braces makes the code actually look like a DSL (Domain-Specific Language).

map, filter, foreach

Let's create a simple list that we will use to demonstrate common higher-order functions from the Collections API:

[15] https://en.wikipedia.org/wiki/Higher-order_function

```
val list = List(1, 2, 3, 4)
```

Arguably, the most used method in the entire Collection API is `map`. It accepts a function that will be applied to each collection element, one by one. For example:

```
val squares = list.map { el => el * el }
println(squares)
// prints List(1, 4, 9, 16)
```

Here each element is multiplied by itself. Note that the original list stays untouched because instead of changing it, `map` returns a new list with updated values.

Another method is `filter`, which allows to keep only those elements that satisfy the condition:

```
val even = list.filter { el => el % 2 == 0 }
println(even)
// prints List(2, 4)
```

Here we created a new collection containing only even numbers.

If we don't need a new collection, but instead we want to perform some action on each element, we can use `foreach`:

```
list.foreach { el => print(el) }
// prints 1234
```

Here we're simply printing each element's value.

flatMap

Suppose that we have two lists:

```
val list = List(1, 2, 3, 4)
val list2 = List("a", "b")
```

What if we wanted to combine each element of the first list with each element of the second list?

We could try to solve this task by nesting two `maps`:

```
val result list.map { el1 =>
  list2.map { el2 =>
    el1 + el2
  }
}
```

The resulting collection will have the following elements:

```
println(result)
// prints List(List(1a, 1b), List(2a, 2b), List(3a, 3b), List(4a, 4b))
```

The `result` has a type of `List[List[String]]`, so we ended up with a list of lists. Not quite what we expected, right? The problem here is the external `map`. If you look at the Scala documentation, you will see that the `map` method accepts a function that takes one element and returns another element. On the other hand, we want to pass a function that takes an element and returns a list. Is there such a function defined on `List`? It turns out that yes, and it's called `flatMap`. Using `flatMap` the problem can be solved easily:

```
val result = list.flatMap { el1 =>
  list2.map { el2 =>
    el1 + el2
  }
}
```

The result is exactly what we needed:

```
println(result)
// List(1a, 1b, 2a, 2b, 3a, 3b, 4a, 4b)
```

Note that the `result` has a type of `List[String]`, so it's a *flattened* version of the list we received previously.

for comprehensions

In order to simplify writing expressions that use `map`, `flatMap`, `filter` and `foreach`, Scala provides so-called *for comprehensions*. Often *for comprehensions* make your code clearer and easier to understand.

Let's rewrite the previous examples with `for` expressions:

```
for { el <- list } yield el * el              // List(1, 4, 9, 16)
for { el <- list if el % 2 == 0 } yield el    // List(2,4)
for { el <- list } println(el)                // prints 1234
```

The `yield` keyword is used when the whole expression returns a value. With `foreach`, however, we are simply printing values on the screen, thus there's no `yield` keyword.

The `flatMap` example translates into the following:

```
for {
  el1 <- list
  el2 <- list2
} yield el1 + el2
```

In my opinion, the last snippet can be easily understood even by people who don't know what `flatMap` is and even that this function exists. Another interesting feature of this syntax is that it removes nesting: no matter how many lists we want to combine, we can put all of them into a single for expression.

For comprehensions are the central part of the Scala programming language. They are used in many situations, and you will see a lot of examples with the `for` keyword throughout this book.

If you want to use *your* classes in for comprehensions, you must provide at least two methods - `map` and `flatMap`. The other two are necessary only if you plan on filtering and iterating with `for`.

Currying

A *curried function* is a function that has each of the parameters in its own pair of parentheses. The process of transforming a regular function into a curried one is called *currying*[16]. A simple function that sums two integers

```
def sum(a: Int, b: Int): Int = a + b
```

can be transformed into the following curried function:

```
def sum(a: Int)(b: Int): Int = a + b
```

If the function is curried, it must be called with each argument in its own pair of parentheses:

[16] https://en.wikipedia.org/wiki/Currying

```
val result = sum(2)(2)
println(result)
// prints 4
```

foldLeft

The `foldLeft` method from the Collections API is a curried method that takes two parameters. First parameter - the initial value - is used as the staring point of a computation, and the second parameter is a function that describes how to compute the next value using the previous one and the accumulator. It may sound terribly complicated, but in reality `foldLeft` is absolutely straightforward to use.

For example, we can use `foldLeft` to calculate the sum of elements in our list - `List(1,2,3,4)`:

```
val sum = list.foldLeft(0) { (acc, next) =>
  acc + next
}
println(sum)
// prints 10
```

Here we replaced parentheses around the second argument with curly braces to make it look like a proper function.

Another (more advanced) reason for making `foldLeft` curried is type inference. By looking at the first argument, the type inferrer gets more insight into the types used in the second argument. Check this question on StackOverflow[17] if you want more details.

Laziness

By default, all values in Scala are initialized *eagerly* at the moment of declaration. This approach is widely known and in many programming languages there is no other way. In Scala, however, values can be initialized *lazily*, which means they remain uninitialized until the first use.

Let's create a fictional service that we will use for demonstration:

```
class ConnectionService {
  def connect = println("Connected")
}
```

The `ConnectionService` has only one method. Obviously, the service must be instantiated somewhere before it's used. The following code reminds us that the order of initialization matters:

[17] http://stackoverflow.com/questions/4915027/

```
1  var service: Service = null
2  val serviceRef = service
3  service = new Service
4  serviceRef.connect
```

The code will fail at runtime with `NullPointerException` because when we declared the `serviceRef`, the service itself hadn't been initialized. Ironically, when we first tried to use it, the service was up and running, so the problem was the order of initialization, not the service being uninitialized. If only we could postpone the initialization of `serviceRef`! It turns out that in Scala we can do exactly that by putting the `lazy` keyword in front of the `val` declaration (line 2).

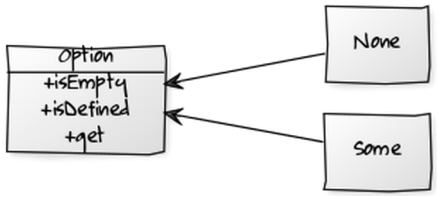
```
2  lazy val serviceRef = service
```

In this case, the `serviceRef` is not initialized until `serviceRef` first referenced. By this time, the service is already up and running, so everything works fine.

Lazy values are sometimes used in Scala for configuration and *dependency injection* (DI). In Java, building a dependency graph is a task performed by various DI frameworks. In Scala, this task can be delegated to the compiler.

Option

`Option` is a container that may or may not contain a value. It is another example of an algebraic data type, because it is formed by two subtypes: `None` and `Some`.



Option as ADT

Options are a great way to show that a particular value may be absent. Essentially, this allows Scala developers to always avoid `NullPointerExceptions` if they play by the rules. Options are used everywhere in the standard library, in third-party libraries and they should be used in user code as well.

The safest way to create a value of type `Option` is to specify the type and then pass the value as the only argument:

```
val opt = Option[String]("Joe")
println(opt)
// prints Some(Joe)
```

This way, you can guarantee that the type will be `Option[String]` and you never end up with the `null` values inside. Even if you pass `null`, you will get `None`:

```
val opt = Option[String](null)
println(opt)
// prints None
```

In many situations, you can also pass a value in `Some`, but this approach doesn't guarantee that the type will be what you expect:

```
val opt = Some(1)
// opt: Some[Int] = Some(1)
```

In the example above, we expected to get a more generic `Option[Int]` but instead, received a more narrow `Some[Int]`. Moreover, we may end up with `null`s inside, which is a very dangerous situation and brings back the possibility of `NullPointerExceptions`:

```
val opt = Some(null)
// opt: Some[Null] = Some(null)
```

There are lots and lots of methods on the `Option` class, which makes `Options` such a pleasure to work with. Here are only some of them:

method	description
`isDefined`	checks whether the value is present
`isEmpty`	checks whether the value is absent
`getOrElse`	returns the value if it's there or the provided default value if it's not
`get`	returns the value if it's there or throws an exception if it is not; very rarely used
`map`	allows to transform one option into another
`foreach`	allows to use existing value without returning anything

Let's look at one concrete example to see how options are used. Suppose that we have a map that may contain information about the user.

We haven't discussed maps yet, but they are absolutely straightforward to use. In essence, maps are collections of key-value pairs. In Scala, they are represented by `Map[K, V]` where `K` stands for *keys*, and `V` stands for *values*.

Here we're interested in two particular keys `firstName` and `lastName` and we want to use them to construct a value for full name.

```
val data = Map(
 "firstName" -> "Joe",
 "lastName" -> "Black"
)
// data: Map[String, String]
```

There are several ways to initialize a map and this is only one of them. Note, that using `->` for creating key-value pairs is a neat trick that is sometimes used in Scala. If you want to learn more about this particular use case, check out this question on StackOverflow[18].

The `apply` method on `Map` returns the value associated with the requested key if the key is present. If it's not, `apply` throws an exception. So, if we know for sure that this map contains both keys, constructing the full name is trivial:

```
val fullName = data("firstName") + " " + data("lastName")
```

If we don't know what information is inside the map, then we need to use the `get` method, which returns an `Option`.

```
val maybeFirstName = data.get("firstName")
// maybeFirstName: Option[String]
val maybeLastName = data.get("lastName")
// maybeLastName: Option[String]
```

We cannot concatenate two `Options` as we would `Strings`, but we can obtain `Strings` by using the `get` method on `Option`:

[18] http://stackoverflow.com/questions/4980515/

```
val maybeFullName = if (maybeFirstName.isDefined &&
  maybeLastName.isDefined) {
  Some(maybeFirstName.get + " " + maybeLastName.get)
} else None
```

When working with `Options` in this way, we must first check whether the value actually exists and then use the `get` method. This approach, however, feel cumbersome, and we can definitely do better than that.

A slightly more Scala-like approach would involve pattern matching:

```
val maybeFullName = (maybeFirstName, maybeLastName) match {
  case (Some(firstName), Some(lastName)) =>
    Some(firstName + " " + lastName)
  case _ => None
}
```

Here we're grouping two values together by enclosing them in parentheses. Then we match the newly created pair against two `case` blocks. The first block works if both `Options` have type `Some` and therefore contain some values inside. The second *catch-all* case is used if either `Option` is actually `None`.

We can also achieve the same result by using `map`/`flatMap` combination:

```
val maybeFullName = maybeFirstName.flatMap { firstName =>
  maybeLastName.map { lastName =>
    firstName + " " + lastName
  }
}
```

Doesn't it look familiar? It certainly does! When we were working with lists, we found out that this construct can be written as a for comprehension:

```
val maybeFullName = for {
  firstName <- maybeFirstName
  lastName <- maybeLastName
} yield firstName + " " + lastName

println(maybeFullName)
// prints Joe Black
```

Great, isn't it? This looks almost as straightforward as `String` concatenation. Please, memorize this construct, because as we will see later, it is used in Scala for doing a great deal of seemingly unrelated things like working with dangerous or asynchronous code.

Try

Let's write a fictitious service that works roughly 60% of the time:

```
object DangerousService {
  def queryNextNumber: Long = {
    val source = Math.round(Math.random * 100)
    if (source <= 60)
      source
    else throw new Exception("The generated number is too big!")
  }
}
```

If we start working with this service directly without taking any precautions, sooner or later our program will blow up:

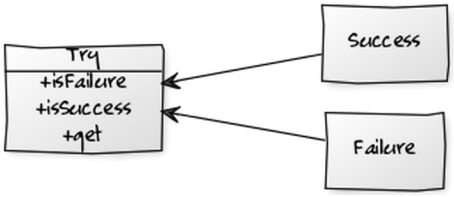
```
scala> DangerousService.queryNextNumber
res118: Long = 27

scala> DangerousService.queryNextNumber
java.lang.Exception: The generated number is too big!
  at DangerousService$.queryNextNumber(<console>:14)
  ... 32 elided
```

To work with dangerous code in Scala you can use `try-catch-finally` blocks. The syntax slightly differs from Java/C#, but the main idea is the same: you wrap dangerous code in the `try` block, and then, if an exception occurs, you can do something about it in the `catch` block.

```
val number = try {
  DangerousService.queryNextNumber
} catch { case e: Exception =>
  e.printStackTrace
  60
}
```

The good news about `try` blocks in Scala is that they return a value, so you don't need to define a `var` before the block and then assign it to some value inside (this approach is extremely common in Java). The bad news is that even if you come up with some reasonable value to return in case of emergency, this will essentially swallow the exception without letting anyone know what has happened. In theory, we could return an `Option`, but the problem here is that `None` will not be able to store information about the exception.

A better alternative is to use `Try` from the standard library. The `Try` type is an algebraic data type that is formed by two subtypes - `Success` and `Failure`.



Try as ADT

The `Try` class provides many utility methods that make working with it very convenient:

method	description
`isSuccess`	checks whether the value was successfully calculated
`isFailure`	checks whether the exception occurred
`get`	returns the value if it is a `Success` or throws an exception if it is a `Failure`; very rarely used
`toOption`	returns `Some` if it is a `Success` or `None` if it is a `Failure`
`map`	allows to transform one `Try` into another
`foreach`	allows to use existing value without returning anything

The `Try` type is defined in `scala.util`, so you need to use `import scala.util.Try` if you want to use `Try` without typing its fully qualified name.

When working with `Try`, the dangerous code can be simply put in the `Try` constructor, which will catch all exceptions if any occur:

```
1  val number1T = Try { DangerousService.queryNextNumber }
2  val number2T = Try { DangerousService.queryNextNumber }
```

If we want to sum two values wrapped in `Try`s, we can use the already familiar `flatMap`/`map` combination:

```
1  val sumT = number1T.flatMap { number1 =>
2    number2T.map { number2 =>
3      number1 + number2
4    }
5  }
```

As always, we can write it as a for comprehension:

```
val sumT = for {
  number1 <- number1T
  number2 <- number2T
} yield number1 + number2
```

Note that a for comprehension *yields* the same container type (in our case `Try`) that was used initially. This happens because `maps` and `flatMaps` can merely transform the value inside the container, but they cannot change the type of the container itself. If, at some point, you need to convert a `Try` into something else, you will need to use utility methods but not `maps` or `flatMaps`.

Future

The final piece of the standard library that we are going to examine here is the `Future` class. `Futures` allow you to work with asynchronous code in a type-safe and straightforward manner without resorting to concurrent primitives like threads or semaphores.

Many methods defined on the `Future` class are curried and accept an `implicit` argument. For example, take a look at the `map` signature:

```
def map[S](f: (T) => S)(implicit executor: ExecutionContext): Future[S]
```

By defining the `map` method this way, the creators of the standard library basically allowed users to completely ignore the second parameter in most cases. Now we can simply declare (or import) an `ExecutionContext` and then work with `Futures` as if they were regular Scala containers like `Options` or `Trys`. In this case, necessary implicit resolution is happens behind the scenes and handled by the compiler.

The global `ExecutionContext` is already defined in the standard library and the only thing that's left is to import it:

```
import scala.concurrent.ExecutionContext.Implicits.global
```

Let's add a pause to our service to make it slower and, therefore, simulate working with something very remote:

```
object DangerousAndSlowService {
  def queryNextNumber: Long = {
    Thread.sleep(2000)
    val source = Math.round(Math.random * 100)
    if (source <= 60)
      source
    else throw new Exception("The generated number is too big!")
  }
}
```

If we try to use this service synchronously, *our* program, which uses this service, will be slowed down. In order to avoid it, we need to wrap our uses of the `DangerousAndSlowService` in a `Future`:

```
val number1F = Future { DangerousAndSlowService.queryNextNumber }
val number2F = Future { DangerousAndSlowService.queryNextNumber }
```

The important thing here is that both `number1F` and `number2F` will be initialized immediately. However, we will not be able to use the actual numbers until they are returned from the service.

The `Future` class provides many methods that you can use:

method	description
`onComplete`	applies the provided callback when the `Future` completes, successfully or not
`onFailure`	applies the provided callback when the `Future` completes with an exception
`onSuccess`	applies the provided callback when the `Future` completes successfully
`mapTo`	casts the type of the `Future` to the given type
`map`	allows to transform one `Future` into another
`foreach`	allows to use existing value without returning anything

In theory, it is possible to work with `Futures` relying solely on callbacks:

```
number1F.onSuccess { case number1 =>
  number2F.onSuccess { case number2 =>
    println(number1 + number2)
  }
}
```

However, this approach makes code very complicated extremely quickly by introducing so-called *callback hell*. Besides, even after your callback is executed it's usually not obvious how to communicate the result (or lack thereof) to the rest of the program.

A better solution is to utilize `map` and `flatMap`:

```
val sumF = number1F.flatMap { number1 =>
  number2F.map { number2 =>
    number1 + number2
  }
}
```

And again, the same code could also be written as a for comprehension:

```
val sumF = for {
  number1 <- number1F
  number2 <- number2F
} yield number1 + number2
```

Usually, if you have only one container (`Option`, `Try`, `Future`, `Seq` and so on), it's easier to simply use `map`. If you have several containers, for comprehensions will be a better choice.

You've probably noticed that in several examples we used functional blocks that start with the `case` keyword. These functional blocks are called *partial functions*[19]. A function is called *partial* if it is defined only for a limited set of arguments. In practice, just remember that if you want to use a higher-order function that accepts a parameter of type `PartialFunction[T, R]`, you are expected to provide a functional block that starts with `case`.

[19] https://en.wikipedia.org/wiki/Partial_function