Covers Scala 2.10

SAMPLE CHAPTER

Scala IN ACTION

Nilanjan Raychaudhuri

FOREWORD BY Chad Fowler

MANNING

Scala in Action

by Nilanjan Raychaudhuri

Chapter 1

brief contents

1 Why Scala?

This chapter covers

- What Scala is
- High-level features of the Scala language
- Why you should pick Scala as your next language

Scala is a general-purpose programming language that runs on Java Virtual Machine (JVM) and .NET platforms. But the recent explosion of programming languages on JVM, .NET, and other platforms raises a question that every developer faces today: which programming language to learn next? Which languages are ready for mainstream development? Among the heap of programming languages like Groovy, Ruby, Clojure, Erlang, and F#, why should you learn Scala?

Learning a new language is merely a beginning. To become a useful and productive developer, you also need to be familiar with all the toggles and gizmos that make up the language infrastructure.

Before I make the case for why Scala should be your next programming language, it's important to understand what Scala is. It's a feature-rich language that's used in various types of applications, starting with building a large messaging layer for social networking sites such as Twitter[1] to creating an application

[1] "Twitter on Scala: A Conversation with Steve Jenson, Alex Payne, and Robey Pointer," Scalazine, April 3, 2009, www.artima.com/scalazine/articles/twitter_on_scala.html.

build tool like SBT[2] (Simple Build Tool). Because of this *scala-bility*, the name of the language is *Scala*.

This chapter explores the high-level features of the language and shows how they compare to the programming languages you may be very familiar with. This will help you to choose Scala as your next programming language.

If you're an object-oriented programmer, you'll quickly get comfortable with the language; if you've used a functional programming language, Scala won't look much different because Scala supports both programming paradigms. Scala is one of those rare languages that successfully integrates both object-oriented and functional language features. This makes Scala powerful because it gives you more in your toolbox to solve programming problems. If you have existing Java applications and are looking for a language that will improve your productivity and at the same time reuse your existing Java codebase, you'll like Scala's Java integration and the fact that Scala runs on the JVM platform.

Now let's explore Scala a bit more.

1.1 What's Scala?

Scala is a general-purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional programming languages, enabling programmers to be more productive. Martin Odersky (the creator of Scala) and his team started development on Scala in 2001 in the programming methods laboratory at EPFL (École Polytechnique Fédérale de Lausanne). Scala made its public debut in January 2004 on the JVM platform and a few months later on the .NET platform.

Even though Scala is fairly new in the language space, it has gained the support of the programming community, which is growing every day. Scala is a rich language in terms of features available to programmers, so without wasting time let's dive into some of them.

> **SCALA ON .NET** At present Scala's support for .NET isn't stable. According to the Scala language website (www.scala-lang.org), the current Scala distribution can compile programs for the .NET platform, but a few libraries aren't supported. The main difficulty to overcome is that Scala programs make heavy use of the Java JDK, which is not available out of the box in the .Net platform. To overcome this issue, the current strategy is to use IKVM (www.ikvm.net), which allows Java programs to convert to MSIL and the .NET library.[3] In this book I mainly focus on Scala for the JVM. The examples in this book are tested only on a JVM.

1.1.1 Scala as an object-oriented language

The popularity of programming languages such as Java, C#, and Ruby has made object-oriented programming (OOP) widely acceptable to the majority of programmers. OOP,

[2] Mark Harrah, "SBT, a Build Tool for Scala," 2012, https://github.com/harrah/xsbt/.

[3] "Scala comes to .Net," July 22, 2011, www.scala-lang.org/node/10299.

as its name implies, is a programming paradigm that uses objects. Think of objects as data structures that consist of fields and methods. Object orientation helps to provide structure to your application using classes and objects. It also facilitates composition so you can create large applications from smaller building blocks. There are many OOP languages in the wild, but only a few are fit to be defined as pure object-oriented languages.

What makes a language purely object-oriented? Although the exact definition of the term depends on whom you ask, most will agree a pure object-oriented language should have the following characteristics:

- Encapsulation/information hiding.
- Inheritance.
- Polymorphism/dynamic binding.
- All predefined types are objects.
- All operations are performed by sending messages to objects.
- All user-defined types are objects.

Scala supports all these qualities and uses a pure object-oriented model similar to that of Smalltalk[4] (a pure object-oriented language created by Alan Kay around 1980), where every value is an object, and every operation is a message send. Here's a simple expression:

```
1 + 2
```

In Scala this expression is interpreted as `1.+(2)` by the Scala compiler. That means you're invoking a `+` operation on an integer object (in this case, `1`) by passing `2` as a parameter. Scala treats operator names like ordinary identifiers. An *identifier* in Scala is either a sequence of letters and digits starting with a letter or a sequence of operator characters. In addition to `+`, it's possible to define methods like `<=`, `-`, or `*`.

Along with the pure object-oriented features, Scala has made some innovations on OOP space:

- *Modular mixin composition*—This feature of Scala has traits in common with both Java interfaces and abstract classes. You can define contracts using one or more traits and provide implementations for some or all of the methods.
- *Self-type*—A mixin doesn't depend on any methods or fields of the class that it's mixed into, but sometimes it's useful to use fields or methods of the class it's mixed into, and this feature of Scala is called *self-type*.
- *Type abstraction*—There are two principle forms of abstraction in programming languages: parameterization and abstract members. Scala supports both forms of abstraction uniformly for types and values.

I cover these areas in detail in chapters 3 and 8.

---

[4] "Smalltalk," Wikipedia, http://en.wikipedia.org/wiki/Smalltalk.

DEFINITION A mixin is a class that provides certain functionality to be inherited by a subclass and isn't meant for instantiation by itself. A mixin could also be viewed as an interface with implemented methods.

1.1.2 Scala as a functional language

Before I describe Scala as a functional language, I'll define functional programming in case you're not familiar with it. *Functional programming* is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

Mutable vs. immutable data

An object is called *mutable* when you can alter the contents of the object if you have a reference to it. In the case of an *immutable* object, the contents of the object can't be altered if you have a reference to it.

It's easy to create a mutable object; all you have to do is provide access to the mutable state of the object. The disadvantage of mutable objects is keeping track of the changes. In a multithreaded environment you need lock/synchronization techniques to avoid concurrent access. For immutable objects, you don't have to worry about these situations.

Functional programming takes more of a mathematical view of the world, where programs are composed of functions that take certain input and produce values and possibly other functions. The building blocks of functional programming are neither objects nor procedures (C programming style) but functions. The simple definition of functional programming is programming with functions.

It's important to understand what is meant by *function* here. A function relates every value of the domain (the input) to exactly one value of the codomain (the output). Figure 1.1 depicts a function that maps values of type X to exactly one value of Y.

Another aspect of functional programming is that it doesn't have side effects or mutability. The benefits of not having mutability and side effects in functional programs are that the programs are much easier to understand (it has no side effects), reason about, and test because the activity of the function is completely local and it has no external effects. Another huge benefit of functional programming is ease of concurrent programming. Concurrency becomes a nonissue because there's no change (immutability) to coordinate

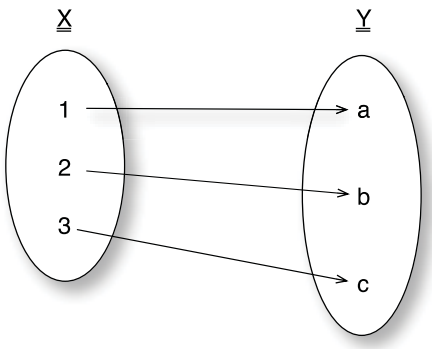

Figure 1.1 A pure function that maps values of X to exactly one value of Y

How do mathematical functions relate to functions in programming?

In mathematics, a function is a relation between a given set of elements called the *domain* (in programming we call this *input*) and a set of elements called the *codomain* (in programming we call this *output*). The function associates each element in the domain with exactly one element in the codomain. For example, `f(x) = y` could be interpreted as

```
x has a relationship f with y or x maps to y via f
```

If you write your functions keeping in mind the definition of the mathematical function, then for a given input your function should always return the same output.

Let's see this in a programming context. Say you have the following function that takes two input parameters and produces the sum of them:

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

For a given input set `(2, 3)` this function always returns `5`, but the following function `currentTime` doesn't fit the definition:

```
def currentTime(timezone: TimeZone) =
       Calendar.getInstance(timezone).getTime
```

For the given `timezone` GMT, it returns different results based on the time of day.

One other interesting property of a mathematical function is *referential transparency*, which means that an expression can be replaced with its result. In the case of `addFunction`, we could replace all the calls made to it with the output value, and the behavior of the program wouldn't change.

between processes or threads. You'll learn about the functional programming side of Scala throughout the book, particularly in chapter 10.

Now let's talk about functional programming languages. Functional programming languages that support this style of programming provide at least some of the following features:

- Higher-order functions (chapter 4)
- Lexical closures (chapter 3)
- Pattern matching (chapters 2 and 3)
- Single assignment (chapter 2)
- Lazy evaluation (chapter 2)
- Type inference (chapter 2)
- Tail call optimization (chapter 5)
- List comprehensions (chapters 2 and 4)
- Mondadic effects (chapter 5)

Some of these features are probably unfamiliar if you haven't done functional programming before.

Side effects

A function or expression is said to have a side effect if, in addition to producing a value, it modifies some state or has an observable interaction with calling functions or the outside world. A function might modify a global or a static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, or call other functions having side effects. In the presence of side effects, a program's behavior depends on its history of execution.

Is Scala a pure functional language?

Scala, put simply, is not a pure functional language. In a pure functional language modifications are excluded, and variables are used in a mathematical sense, with identifiers referring to immutable and persistent values. An example of a pure functional language is Haskell.

Scala supports both types of variables: single-assignment variables (also called values) that don't change their value throughout their lifetime and variables that point to a mutable state or could be reassigned to other objects. Even though you should use immutable objects whenever possible, Scala as a language doesn't provide any restrictions. The restriction is purely conventional. A good rule of thumb is to always default to `val` and use variables when it's absolutely necessary.

To me, the fundamental property of a functional language is treating functions as values, and Scala does that well.

Scala supports most of them, but to keep it simple Scala is a functional language in the sense that functions are *first-class values*. That means that in Scala, every function is a value (like some integer value `1` or some string value `"foo"`), and like any values, you can pass them as parameters and return them from other functions. In Scala you can assign a function `(x: Int) => x + 1` to a `val inc` and use that to invoke that function:

```
val inc = (x : Int) => x + 1
inc(1)
```

Here `val` represents a single assignment variable (like Java final variables) with a value that can't be changed after the assignment. The output of the function call is `2`.
In the following example you'll see how to pass functions as parameters to another function and get the result:

```
List(1, 2, 3).map((x: Int) => x + 1)
```

In this case you're passing an increment function to another function called `map`, and the output produced by the invocation of the `map` function will be `List(2, 3, 4)`. Based on the output you can see that `map` is invoking the given function for each element in the list. Don't worry about the syntax right now; you'll learn about it in detail in later chapters.

1.1.3 Scala as a multi-paradigm language

Scala is a multi-paradigm language because it supports both functional and OOP programming. Scala is the first to unify functional programming and OOP in a statically typed language for the JVM. The obvious question is why we need more than one style of programming.

The goal of multi-paradigm computing is to provide a number of problem-solving styles so a programmer can select the solution that best matches the characteristics of the problem to be solved. This provides a framework where you can work in a variety of

styles and mix the constructs from different ones. Functional programming makes it easy to build interesting things from simple parts (functions), and OOP makes it easy to adopt and extend complex systems using inheritance, classes, and so on.

According to researcher Timothy Budd,[5] "Research results from the psychology of programming indicate that expertise in programming is far more strongly related to the number of different programming styles understood by an individual than it is the number of years of experience in programming."

How can Scala combine these two different and almost opposite paradigms into one programming language? In the case of OOP, building blocks are objects, and in functional programming building blocks are functions. In Scala, functions are treated as objects.

FUNCTIONS AS OBJECTS

One of the benefits of combining functional programming with object-oriented programming in Scala is treating functions as objects.

Scala, being a functional language, treats functions as values, and you saw one example of assigning a function to a variable. Because all values in Scala are objects, it follows that functions are objects too. Look at the previous example again:

```
List(1, 2, 3).map((x: Int) => x + 1)
```

You're passing the function `(x:Int) => x + 1` to the method `map` as a parameter. When the compiler encounters such a call, it replaces the function parameter with an object, as in the following:

```
List(1, 2, 3).map(new Function1[Int, Int]{ def apply(x:Int): Int = x + 1})
```

What's going on here? Without diving in too deeply for now, when the Scala compiler encounters functions with one parameter, it replaces that call with an instance of class `scala.Function1`, which implements a method called `apply`. If you look carefully, you'll see that the body of the function is translated into the `apply` method. Likewise, Scala has `Function` objects for functions with more than one parameter.

As the popularity of multi-paradigm programming increases, the line between functional and object-oriented programming will fade away.[6] As we continue to explore Scala, you will see how we blend both functional programming and OOP to solve problems.

1.1.4 Scala as a scalable and extensible language

Scala stands for *scalable language*.[7] One of the design goals of Scala is to create a language that will grow and scale with your demand. Scala is suitable for use as a scripting language, as well as for large enterprise applications. Scala's component abstraction,

---

[5] Timothy A. Budd's personal web page, http://web.engr.oregonstate.edu/~budd/.

[6] "A Postfunctional Language," www.scala-lang.org/node/4960.

[7] "Scala: A Scalable Language" by Martin Odersky, Lex Spoon, and Bill Venners, Scalazine, May 6, 2008, www.artima.com/scalazine/articles/scalable-language.html.

succinct syntax, and support for both object-oriented and functional programming make the language scalable.

Scala also provides a unique combination of language mechanisms that makes it easy to add new language constructs in the form of libraries. You could use any method as an infix or postfix operator, and closures in Scala can be passed as "pass by name" arguments to other functions (see the next listing). These features make it easier for developers to define new constructs.

Let's create a new looping construct called `loopTill`, which is similar to the `while` loop in the following listing.

Listing 1.1 Creating the loop construct `loopTill` in Scala

```
def loopTill(cond: => Boolean)(body: => Unit): Unit = {
 if (cond) {
   body
   loopTill(cond)(body)
 }
}
var i = 10

loopTill (i > 0) {
   println(i)
   i -= 1
}
```

In this code you're creating a new `loopTill` construct by declaring a method called `loopTill` that takes two parameters. The first parameter is the condition (`i > 0`) and the second parameter is a closure. As long as the condition evaluates to `true`, the `loopTill` function will execute the given closure.

> **DEFINITION** *Closure* is a first-class function with free variables that are bound in the lexical environment. In the `loopTill` example, the free variable is `i`. Even though it's defined outside the closure, you could still use it inside. The second parameter in the `loopTill` example is a closure, and in Scala that's represented as an object of type `scala.Function0`.

Extending a language with a library is much easier than extending the language itself because you don't have to worry about backward compatibility. For example, Scala actor implementation (defined in section 1.2.2) is provided as a library and isn't part of the Scala language. When the first actor implementation didn't scale that well, another actor implementation was added to Scala without breaking anything.

1.1.5 Scala runs on the JVM

The best thing about Java is not the language but the JVM. A JVM is a fine piece of machinery, and the Hotspot team has done a good job in improving its performance over the years. Being a JVM language, Scala integrates well with Java and its ecosystem, including tools, libraries, and IDEs. Now most of the IDEs ship with the Scala plug-in so that you can build, run, and test Scala applications inside the IDE. To use Scala you

don't have to get rid of all the investments you've made in Java so far. Instead you can reuse them and keep your ROI coming.

Scala compiles to Java byte code, and at the byte-code level you can't distinguish between Java code and Scala code. They're the same. You could use the Java class file disassembler javap to disassemble Scala byte code (chapter 11 looks into this in more detail) as you could for Java classes.

Another advantage of running Scala on a JVM is that it can harness all the benefits of JVM-like performance and stability out of the box. And being a statically typed language, Scala programs run as fast as Java programs.

I go through all these features of Scala in more detail throughout the book, but I still haven't answered the question—why Scala?

1.2 The current crisis

An interesting phenomenon known as "Andy giveth, and Bill taketh away" comes from the fact that no matter how fast processors become, we software people find a way to use up that speed. There's a reason for that. With software you're solving more and more complex problems, and this trend will keep growing. The key question is whether processor manufacturers will be able to keep up with the demand for speed and processor power. When will this cycle end?

1.2.1 End of Moore's law

According to Moore's law, the number of transistors per square inch on a chip will double every 18 months. Unfortunately, Intel and other CPU manufacturers are finally hitting the wall[8] with Moore's law and instead are taking the route of multicore processors. The good news is that processors are going to continue to become more powerful, but the bad news is that our current applications and programming environments need to change to take advantage of multicore CPUs.

1.2.2 Programming for multicores

How can you take advantage of the new multicore processor revolution?

Concurrency. Concurrency will be, if it isn't already, the way we can write software to solve our large, distributed, complex enterprise problems if we want to exploit the CPU throughputs. Who doesn't want efficient and good performance from their applications? We all do.

A few people have been doing parallel and concurrent programming for a long time, but it still isn't mainstream or common among enterprise developers. One reason is that concurrent programming has its own set of challenges. In the traditional thread-based concurrency model, the execution of the program is split into multiple concurrently running tasks (threads), and each operates on shared memory. This leads to hard-to-find race conditions and deadlock issues that can take weeks and

[8] "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," by Herb Sutter, originally published in Dr. Dobb's Journal, March 2005, www.gotw.ca/publications/concurrency-ddj.htm.

months to isolate, reproduce, and fix. It's not the threads but the shared memory that's the root of all the concurrency problems. The current concurrency model is too hard for developers to grok, and we need a better concurrent programming model that will help developers easily write and maintain concurrent programs.

Scala takes a totally different approach to concurrency: the Actor model. An actor[9] is a mathematical model of concurrent computation that encapsulates data, code, and its own thread of control and communicates asynchronously using immutable (no side effects) message-passing techniques. The basic Actor architecture relies on a shared-nothing policy and is lightweight in nature. It's not analogous to a Java thread; it's more like an event object that gets scheduled and executed by a thread. The Scala Actor model is a better way to handle concurrency issues. Its shared-nothing architecture and asynchronous message-passing techniques make it an easy alternative to existing thread-based solutions.

History of the Actor model

The Actor model was first proposed by Carl Hewitt in 1973 in his paper "A Universal Modular ACTOR Formalism for Artificial Intelligence" and was later on improved by Gul Agha ("ACTORS: A Model of Concurrent Computation in Distributed Systems").

Erlang was the first programming language to implement the Actor model. Erlang is a general-purpose concurrent programming language with dynamic typing. After the success of the Erlang Actor model at Ericsson, Facebook, and Yahoo!, it became a good alternative for handling concurrency problems, and Scala inherited it. In Scala, actors are implemented as a library that allows developers to have their own implementation. In chapters 7 and 12 you'll look into various Scala actor implementations.

Traditionally, programming multicore processors is more complex than programming uniprocessors and it requires platform-specific knowledge. It's also harder to maintain and manage these codebases. To make parallel programming easier, Scala provides higher abstractions in the form of a parallel collections library that hides parallel algorithms. For example, to square up each element of a `List` in parallel, you can use parallel collections like the following:

```
List(1, 2, 3).par.map(x => x * x)
```

In this case the .par transforms the `List` into a parallel collection that implements the map method using a parallel algorithm. Behind the scenes a parallel collections library will fork threads necessary to execute the map method using all the cores available in a given host machine. The parallel collections library is a new addition to Scala and provides parallel versions of most collection types. I explore more about parallel collections in chapter 4.

---

[9] "Actor model," Wikipedia, http://en.wikipedia.org/wiki/Actor_model.

1.3 Transitioning from Java to Scala

> "If I were to pick a language to use today other than Java, it would be Scala."
>
> —James Gosling

When Java, released in May 1995 by Sun Microsystems, arrived on the programming language scene, it brought some good ideas, such as a platform-independent programming environment (write once, run anywhere), automated garbage collection, and OOP. Java made object-oriented programming easier for developers, compared with C/C++, and was quickly adopted into the industry.

Over the years Java has become bloated. Every new feature added to the language brings with it more boilerplate code for the programmer; even small programs can become bloated with annotations, templates, and type information. Java developers are always looking for new ways to improve productivity using third-party libraries and tools. But is that the answer to the problem? Why not have a more productive programming language?

1.3.1 Scala improves productivity

Adding libraries and tools to solve the productivity problem sometimes backfires, adding complexity to applications and reducing productivity. I'm not saying that you shouldn't rely on libraries; you should whenever it makes sense. But what if you had a language built from the ground up from ideas like flexibility, extensibility, scalability—a language that grows with you?

Developers' needs today are much different than they used to be. In the world of Web 2.0 and agile development, flexibility and extensibility in the programming environment are important. Developers need a language that can scale and grow with them. If you're from Java, then Scala is that language. It will make you productive, and it will allow you to do more with less code and without the boilerplate code.

1.3.2 Scala does more with less code

To see the succinctness of Scala, you have to dive into the code. The next two listings provide a simple example of finding an uppercase character in a given string, comparing Scala and Java code.

Listing 1.2 Finding an uppercase character in a string using Java

```
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
        break;
    }
}
```

In this code you're iterating through each character in the given string name and checking whether the character is uppercase. If it's uppercase, you set the `hasUpperCase` flag to `true` and exit the loop. Now let's see how we could do it in Scala.

Listing 1.3 Finding an uppercase character in a string using Scala

```
val hasUpperCase = name.exists(_.isUpper)
```

In Scala you can solve this problem with one line of code. Even though it's doing the same amount of work, most of the boilerplate code is taken out of the programmer's hands. In this case you're calling a function called `exists` on `name`, which is a string, by passing a predicate that checks whether the character is true, and that character is represented by `_`. This demonstrates the brevity of the Scala language and its readability. Now let's look at the following listing, where you create a class called `Programmer` with the properties `name`, `language`, and `favDrink`.

Listing 1.4 Defining a `Programmer` class in Java

```
public class Programmer {

    private String name;
    private String language;
    private String favDrink;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getLanguage() {
        return language;
    }
    public void setLanguage(String language) {
        this.language = language;
    }
    public String getFavDrink() {
        return favDrink;
    }
    public void setFavDrink(String favDrink) {
        this.favDrink = favDrink;
    }
}
```

This is a simple POJO (plain old Java object) with three properties—nothing much to it. In Scala you could create a similar class in one line, as in the following listing.

Listing 1.5 Defining a `Programmer` class in Scala

```
class Programmer(var name:String,var language:String,var favDrink:String
```

In this example you're creating a similar class called `Programmer` in Scala but with something called a *primary constructor* (similar to a *default constructor* in Java) that takes three arguments. Yes, you can define a constructor along with the class declaration—another example of succinctness in Scala. The `var` prefix to each parameter makes the Scala compiler generate a getter and setter for each field in the class. That's impressive, right? You'll look into more interesting examples throughout the book

when you go deeper into Scala. For now, it's clear that with Scala you can do more with fewer lines of code. You could argue that the IDE will automatically generate some of this boilerplate code, and that's not a problem. But I'd argue that you'd still have to maintain the generated code. Scala's succinctness will be more apparent when you look into much more involved examples. In Java and Scala code comparisons, the same feature requires 3 to 10 times more lines in Java than Scala.

1.4 Coming from a dynamic language

It's hard to find developers these days who haven't heard of or played with Ruby, Groovy, or Python. The biggest complaint from the dynamic language camp about statically typed languages is that they don't help the productivity of the programmer and they reduce productivity by forcing programmers to write boilerplate code. And when dynamically typed languages are compared with Java, obvious things like closures and extensibility of the language are cited everywhere. The obvious question here is how Scala is different.

Before going into the issue of static versus dynamically typed languages, let's look into Scala's support for closures and mixin. The following listing shows how to count the number of lines in a given file in Ruby.

Listing 1.6 Counting the number of lines in a file in Ruby

```
count = 0
File.open "someFile.txt" do |file|
   file.each { |line| count += 1 }
end
```

You're opening the file someFile.txt and for each line incrementing the count with 1. Simple! The following listing shows how you can do this in Scala.

Listing 1.7 Counting the number of lines in a file in Scala

```
val src = scala.io.Source.fromFile("someFile.txt")
val count = src.getLines().map(x => 1).sum
```

The Scala code looks similar to the Ruby code. You could solve this in many ways in Scala; here you're using the `map` method to return 1 for each line, then using the `sum` method to calculate the total count.

Scala supports mixin composition with something called *traits*, which are similar to an abstract class with partial implementation. For example, you can create a new type of collection which allows users to access file contents as iterable, by mixing the Scala `Iterable` trait. The only contract is to implement an iterator method:

```
class FileAsIterable {
  def iterator = scala.io.Source.fromFile("someFile.txt").getLines()
}
```

Now if you mix in the Scala `Iterable`, your new `FileAsIterable` will become a Scala `Iterable` and will start supporting all the `Iterable` methods:

```
val newIterator = new FileAsIterable with Iterable[String]
newIterator.foreach { line => println(line) }
```

In this case you're using the `foreach` method defined in the `Iterable` trait and printing each line in the file.

Scala version 2.10 adds support for a `Dynamic`[10] type. Using this feature you can dynamically add methods and fields to a type at runtime. This is very similar to the `method_missing` feature of Ruby and is quite useful if you're building a domain-specific language (DSL). For example, Scala map is a collection of key value pairs and if you want to access the value associated with a key you can do something like the following:

```
val someMap = Map("foo" -> 1, "bar" -> 2)
someMap.get("foo")
```

Here `someMap` is a collection of two key value pairs and `someMap.get("foo")` will return 1. Using `Dynamic` we can easily change that so that we can access the keys as if they were part of a type:

```
class MyMap extends Dynamic {
  ...
  def selectDynamic(fieldName: String) = map.get(fieldName)
  private val map = Map("foo" -> "1", "bar" -> 2)
}

val someMap = new MyMap
someMap.foo
someMap.bar
```

The magic ingredient in this case is the `selectDynamic` method. (Scala methods are defined using the `def` keyword.) When the Scala compiler checks that `foo` is not part of the type it doesn't give up immediately. If the type is a subtype of `Dynamic` it looks for the `selectDynamic` method and invokes it. If the method is not provided, you will get a compilation error.

Scala also supports something called *implicit conversion*, which is similar to Ruby open classes but scoped and compile time checked. Examples of implicit conversions are available throughout the book.

1.4.1 Case for static typing, the right way

With all that said and done, Scala is still a statically typed language. But if you've gone through the examples in the previous section, you've probably already figured out that Scala's static typing doesn't get in your face, and it almost feels like a dynamically typed language. But still, why should you care about static typing?

> **DEFINITION** *Static typing* is a typing system where the values and the variables have types. A number variable can't hold anything other than a number. Types are determined and enforced at compile time or declaration time.

[10] "SIP-17 Type Dynamic," http://docs.scala-lang.org/sips/pending/type-dynamic.html.

DEFINITION *Dynamic typing* is a typing system where values have types but the variables don't. It's possible to successively put a number and a string inside the same variable.

The size and the complexity of the software you're building are growing every day, and having a compiler do the type checking for you is great. It reduces the time you need to spend fixing and debugging type errors. In a statically typed language like Scala, if you try to invoke a length method on a number field, the Scala compiler will give you a compilation error. In a dynamically typed language you'll get a runtime error.

Another benefit of a statically typed language is that it allows you to have powerful tools like refactoring and IDEs. Having an IDE might not interest you because of powerful editing tools like Emacs and TextMate, but having refactoring support is great when working on large codebases.

All these benefits do come with a price. Statically typed languages are more constraining than dynamically typed languages, and some force you to provide additional type information when you declare or call a function. But having constraints is useful when building a large application because they allow you to enforce a certain set of rules across the codebase. Scala, being a type-inferred language, takes care of most of the boilerplate code for the programmer (that's what compilers are good for, right?) and takes you close to a dynamically typed language, but with all the benefits of a statically typed language.

DEFINITION *Type inference* is a technique by which the compiler determines the type of a variable or function without the help of a programmer. The compiler can deduce that the variable `s` in `s="Hello"` will have the type string because `"hello"` is a string. The type inference ensures the absence of any runtime type errors without putting a declaration burden on the programmer.

To demonstrate how type inference works, create an array of maps in Scala:

```
val computers = Array(
                    Map("name" -> "Macbook", "color" -> "white"),
                    Map("name" -> "HP Pavillion", "color" -> "black")
                )
```

If you run this Scala code in the Scala REPL, you'll see the following output:

```
computers:
     Array[scala.collection.immutable.Map[java.lang.String,java.lang.String]]
     = Array(Map(name -> Macbook, color -> white), Map(name -> HP Pavillion,
     color -> black))
```

Even though you only specified an array of maps with key and value, the Scala compiler was smart enough to deduce the type of the array and the map. And the best part is that now if you try to assign the value of `name` to some integer type variable somewhere in your codebase, the compiler will complain about the type mismatch, saying that you can't assign `String` to an integer-type variable.

1.5 For the programming language enthusiast

One of the main design goals for Scala was to integrate functional and OOP into one language (see section 1.1.4 for details). Scala is the first statically typed language to fuse functional and OOP into one language for the JVM. Scala has made some innovations in OOP (mentioned previously) so that you can create better component abstractions.

Scala inherits lots of ideas from various programming languages of the past and present. To start with, Scala adopts its syntax from Java/C# and supports both JVM and Common Language Runtime (CLR). Some would argue that Scala's syntax is more dissimilar than similar to that of Java/C#. You saw some Scala code in previous sections, so you can be the judge of that. In Scala every value is an object, and every operation is a method call. Smalltalk influences this pure object-oriented model. Scala also supports universal nesting and uniform access principles (see the following listing), and these are borrowed from Algol/Simula and Eiffel, respectively. In Scala variables and functions without parameters are accessed the same way.

Listing 1.8 Universal access principles in Scala

```
class UAPExample {
  val someField = "hi"
  def someMethod = "there"
}

val o = new UAPExample
o.someField
o.someMethod
```

Here you're accessing a field and a method of the instance of the `UAPExample` class, and to the caller of the class it's transparent.

Scala's functional programming constructs are similar to those of the metalanguage (ML) family of languages, and Scala's Actor library is influenced by Erlang's Actor model.

> **COMPILE MACROS** The Scala 2.10 release adds experimental support for compile-time macros.[11] This allows programmers to write macro defs: functions that are transparently loaded by the compiler and executed during compilation. This realizes the notion of compile-time metaprogramming for Scala.

Based on this list you may realize that Scala is a rich language in terms of features and functionality. You won't be disappointed by Scala and will enjoy learning this language.

1.6 Summary

In this chapter I quickly covered many concepts, but don't worry because I'm going to reiterate these concepts throughout the book with plenty of examples so that you can relate them to real-world problems.

---

[11] Eugene Burmako, "Def Macros," http://docs.scala-lang.org/overviews/macros/overview.html.

You learned what Scala is and why you should consider learning Scala as your next programming language. Scala's extensible and scalable features make it a language that you can use for small to large programming problems. Its multi-paradigm model provides programmers with the power of abstractions from both functional and OOP models. Functional programming and actors will make your concurrent programming easy and maintainable. Scala's type inference takes care of the pain of boilerplate code so that you can focus on solving problems.

In the next chapter you'll set up your development environment and get your hands dirty with Scala code and syntax.

SCALA/JAVA

Scala IN ACTION

Nilanjan Raychaudhuri

Free eBook
SEE INSERT

Scala runs on the JVM and combines object-orientation with functional programming. It's designed to produce succinct, type-safe code, which is crucial for enterprise applications. Scala implements Actor-based concurrency through the amazing Akka framework, so you can avoid Java's messy threading while interacting seamlessly with Java.

Scala in Action is a comprehensive tutorial that introduces the language through clear explanations and numerous hands-on examples. It takes a "how-to" approach, explaining language concepts as you explore familiar programming tasks. You'll tackle concurrent programming in Akka, learn to work with Scala and Spring, and learn how to build DSLs and other productivity tools. You'll learn both the language and how to use it.

What's Inside

- A Scala tutorial
- How to use Java and Scala open source libraries
- How to use SBT
- Test-driven development
- Debugging

Experience with Java is helpful but not required. Ruby and Python programmers will also find this book accessible.

Nilanjan Raychaudhuri is a skilled developer, speaker, and an avid polyglot programmer who works with Scala on production systems.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/ScalainAction

"A great way to get started on building real-world applications using Scala and its tools and frameworks."
—Martin Odersky, Creator of Scala

"Makes one wonder why functional programming isn't more widely used in the industry!"
—Alexandre Alves, Oracle Corp.

"A must for any forward-looking Java developer."
—Michael Smolyak
Next Century Corp.

"Like having an experienced mentor."
—From the Foreword by Chad Fowler, Author, Speaker, and Engineer

ISBN 13: 978-1-935182-75-7
ISBN 10: 1-935182-75-7
54499
9 781935 182757

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]