1
1

Why Scala?

This chapter covers

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

Because of the recent explosion of programming languages in JVM, .NET, and other platforms, a common question every developer faces today is which programming language to learn next. Which of these languages are ready for mainstream development? Among the heap of programming languages like Groovy, Ruby, Erlang, and F#, why should you learn Scala? Learning a new language is just 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 for you 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 like Twitter1 to creating an application build tool like SBT2 (Simple Build Tool). Because of this scalability, the name of the language is Scala.

In this chapter we'll explore the high-level features of the language and show how they compare to the programming languages you're comfortable with. If you're an object-oriented programmer, then you'll quickly get comfortable with the language; if you've used a functional programming language, then 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. If you have existing Java applications and are looking for a language that will improve your productivity and at the same reuse your existing Java codebase, then you'll like Scala's Java integration and the fact that Scala runs on JVM. Now let's dive in and look at what Scala is.

1.1.1 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 few months later on the .NET platform.

1 http://www.artima.com/scalazine/articles/twitter_on_scala.html
2 http://code.google.com/p/simple-build-tool/

Even though Scala is fairly new in the language space, it has already gained popularity and the support of the programming community, which is growing everyday. 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.scalalang.
org), the current Scala distribution can compile programs for the .NET platform, but a few libraries aren't supported. Scala's structural types don't work on the .NET platform yet. In this book I'll mainly focus on Scala for JVM. The examples in this book are tested only on JVM.

1.1.2 Scala as object-oriented language

The popularity of programming languages like Java, C#, and Ruby made object-oriented programming widely acceptable to the majority of programmers. Object orientation helps us to provide structure to our
application using classes and objects. It also facilitates composition so that we can create large applications from smaller building blocks. Even though there are many object-oriented programming languages in the wild, 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 is highly variable depending on
whom you ask, most will agree on several qualities. 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 of these qualities and uses a pure object-oriented model similar to that of Smalltalk3 (Smalltalk is a pure object-oriented language created by Alan Kay about 1980), where every value is an object and every operation is a message send. Let's look at a simple expression:

3 http://en.wikipedia.org/wiki/Smalltalk

```
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 object-oriented programming space:

 Modular mixin composition—This feature of Scala provides a mechanism for composing classes for designing reusable components without the problems of multiple inheritance. Mixin in Scala is similar to both Java interfaces and abstract classes. On the one hand, you could define contracts using it and have multiples of them (such as interfaces). On the other hand, you could have concrete method implementations.
 Self-type—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 styles of abstraction uniformly for types and values.

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. *

We'll cover these areas in detail in chapters 3 and 6.

1.1.3 Scala as functional language

Before I describe Scala as functional language, you need to understand a bit of functional programming.
Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

Mutable versus immutable data *

An object is called mutable when you can alter the contents of the object if you have a reference to it. Whereas 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. But the disadvantages of mutable objects are that you have to keep track of the changes, and in a multithreaded environment you have to provide some sort of locking/synchronization for 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.

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) = returns you the current time for the given
timezone.
```

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

One other interesting property of mathematical function is referential transparency. It 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 behavior of the program wouldn't change.

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 the programs are much easier
to understand (you don't have to worry about 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 between processes or threads. You'll learn about the functional programming side of Scala throughout the book and especially in chapter 10.

Side effects *

A function or expression is said to have a side effect if, in addition to producing a value, it also 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 side-effecting functions. In the presence of side effects, a program's behavior depends on its history of execution.

With the definition of functional programming out of the way, it's time to 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 10)
 List comprehensions (chapters 2 and 4)
 Mondadic effects (chapter 10)

Some of these features will be unknown to you if you haven't done functional programming before. Scala supports most of the features mentioned, but to keep it simple, Scala is a functional language in the
sense that functions are first-class values. This means that in Scala, every function is a value (like some integer value 1 or some string value "foo"), and like any value, 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 value type inc and use that value to invoke that function:

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

inc(1)
```

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

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

In this case we'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.

Is Scala a pure functional language? *

The simple answer is no. 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 pure functional language is Haskell.

Scala supports both types of variables: single-assignment variables that won't change their value throughout their lifetime and variables that point to a mutable state and could be reassigned to other objects. Even though you should use immutable objects whenever possible, Scala as a language doesn't provide any sort of restrictions.

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

1.1.4 Scala as multi-paradigm language

Scala is multi-paradigm language because it supports both functional and object-oriented programming. Scala is the first to unify functional programming (FP) and object-oriented programming (OOP) in a statically typed language. 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 that a programmer can select the solution technique 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 styles. Functional programming makes it easy to build interesting things from simple parts (functions), and object-oriented programming makes it easy to adopt and extend complex systems using
inheritance, classes, and so on.

According to researcher Timothy Budd,4 "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 individual than it is the number of years of experience in programming."

4 http://web.engr.oregonstate.edu/~budd/

5 "A Postfunctional Language," http://www.scala-lang.org/node/4960

The biggest question that arises after what we've discussed so far is how Scala could combine these two different and almost opposite paradigms into one programming language.

In the case of object-oriented programming, building blocks are objects, and in functional programming building, blocks are functions. In Scala, functions are treated as objects.

1.1.5 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 value, and you saw one example of assigning a function to a variable. Because all values in Scala are objects, then it follows that functions are objects too. Let's look our previous example again:

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

In this example we'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 too deeply into detail for now, when the Scala compiler encounter
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 our 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 objectoriented
programming will fade away.5

1.1.6 Scala as scalable and extensible language

Scala stands for scalable language.6 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, and you could also use

6 http://www.artima.com/scalazine/articles/scalable-language.html

Scala for large enterprise applications. Scala's component abstraction, succinct syntax, and support for both object-oriented and functional programming all helped to make the language more 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 would automatically target typed dependent. 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; see listing 1.1.

Listing 1.1 Creating the new 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 we'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. You'll see more techniques and examples in the DSL chapter (chapter 14) on how to extend Scala.

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 implement
didn't scale quite that well, another Actor implementation was added to Scala without breaking anything.

1.1.7 Scala runs on JVM

The best thing about Java is not the language but the Java Virtual Machine. JVM is a fine piece of machinery, and the Hotspot team has done a good job in improving performance of JVM over the years.
Being a JVM language, Scala integrates well with Java and its ecosystem, like 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 made in Java so far. Instead, you could 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 javap (the Java class file disassembler) to disassemble Scala byte code (we'll look into this in more detail in chapter 11) as you could for Java classes.

Another advantage of running Scala on JVM is that it could 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.

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

1.2 The current crisis

An interesting phenomenon is known as “Andy giveth, and Bill taketh away.” 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 we’re solving more and more complex problems, and this trend will keep growing in the foreseeable future. The key question is whether processor manufacturers will be able to keep up with our
demand for speed and processor power. When this will 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. But unfortunately, Intel and other CPU manufactures are hitting the wall7 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.

7 http://www.gotw.ca/publications/concurrency-ddj.htm

8 http://en.wikipedia.org/wiki/Actor_model

1.2.2 Programming for multicores

How you can 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 to
have an efficient and good performance from their applications? We all do.

Few of us have been doing parallel and concurrent programming for a long time, but it still isn’t mainstream or common among enterprise developers. One of the reasons behind this 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 of them

operates on shared memory. This leads to hard-to-find race conditions and deadlock issues that take weeks and 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 Actor8 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 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 makes 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 famous 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 is 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 Ericssion, 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 we'll look into various Scala Actor implementations.

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

Java was first released in May 1995 by Sun Microsystems. Since then, Java has come a long way as a programming language. When Java came to the programming language scene, it brought some good ideas like a platform-independent programming environment (write once, run anywhere), automated garbage collection, and object-oriented programming. Java made object-oriented programming easier for
developers when compared with C/C++ and got quickly adopted into the industry.

Over the years Java is becoming a bloated language. 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. As Java developers, we're always looking out for new ways
to improve our productivity using third-party libraries and tools. But is that the answer to our problem? Why not have a more productive programming language?

1.3.1 Scala improves productivity

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

Today developers' needs are much different than they used to be. In the world of Web 2.0 and agile development, we understand how important it is to have flexibility and extensibility in our programming
environment. We need a language that can scale and grow with us. 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 demonstrate the succinctness of Scala, we have to dive into the code. Let's pick up a simple example

of finding an uppercase character in a given string and compare Scala and Java code; see listing 1.2.

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).

Listing 1.3 Finding an uppercase character in a string using Scala

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

In Scala you could 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 programmer's hands. In this case we'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 _. Here I wanted to demonstrate the brevity of the Scala language without losing the readability, but now let's look at another example (listing 1.4), where we 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 we could
create a similar class in one line, as in listing 1.5.

Listing 1.5 Defining a Programmer class in Scala

```
class Programmer(
```

```
 var name:String,

 var language:String,

 var favDrink:String

 )
```

In this example we'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, in Scala you could define a constructor along with the class declaration, another example of succinctness in Scala.
The var prefix to each parameter will make the Scala compiler generate a getter and setter for each field in the class. That's impressive, right? We'll look into more interesting examples through out the book when we'll take a deep dive into Scala. But for now, it's clear that with Scala we could do more with fewer
lines of code. You could argue that IDE will automatically generate some of this boilerplate code, and that's not a problem. But I'd argue that you'll still have to maintain the generated code. Scala's succinctness will be more apparent when we'll look into much more involved examples. In Java and Scala
code comparisons, the same feature requires 3 to 10 times fewer lines in Scala than Java.

1.4 Coming from a dynamic language

It's hard to find developers these days who haven't heard 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, then obvious things like
closures and extensibility of the language are cited everywhere. The obvious question is how Scala is different.

Before going into the issue of statically versus dynamically typed languages, let's look into Scala's support for closures and mixin. Listing 1.6 shows how to count number of lines in a given file in Ruby.

Listing 1.6 Count number of lines in a file in Ruby

```
count = 0

File.open "someFile.txt" do |file|

 file.each { |line| count += 1 }

end
```

We're opening the file someFile.txt and for each line incrementing the count with 1. Simple! Now let'

see
how we could do this in Scala (listing 1.7).

Listing 1.7 Count number of lines in a file in Scala

```
 val src = scala.io.Source.fromFile(new java.io.File("someFile.txt"))

 val count = src.getLines(System.getProperty("line.separator")).foldLeft(0) {

 (i, line) => i + 1

 }
```

The Scala code looks similar to the Ruby code. We're calling foldLeft (similar to inject) by passing a closure that takes the current running count and the line from a file. It's not much different from the Ruby
implementation.

Scala supports mixin with something called traits that's similar to an abstract class with partial implementation. For example, if you want to create a new type of collection that will allow users to access
file contents as iterable, you can do that by mixing the Scala Iterable trait. The only contract is to implement an iterator method.

```
class FileAsIterable {

 val src = Source.fromFile(new File("someFile.txt"))

 def iterator = src.getLines(getProperty("line.separator"))

}
```

Now while creating the instance of the class, 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 we're using the foreach method defined in the Iterable trait and printing each line in the file.

Scala also supports something called implicits that are similar to Ruby open classes but a little more manageable. Scala implicits are a great type-safe way to create DSLs in Scala, and we'll look into that in
chapter 14.

1.4.1 Case for static typing, the right way

With all said and done so far, Scala is still a statically typed language. But if you've gone through the examples in the previous section, then 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 we care about static typing?

DEFINITION Static typing—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.

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

The size and the complexity of the software we're building are growing every day, and having a compiler do the type checking for us is great. It reduces the time we 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. But for dynamically typed languages 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 some of you because of powerful editing tools like
Emacs and TextMate, but having refactoring support is great, especially when working on a large codebases.

All these benefits do come with a price. The price is that statically typed languages are more constraining compared to dynamically typed languages, and some of them force you to provide additional
type information when you declare or call a function. But sometimes having constraints is useful when building a large application because that will 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—A technique by which the compiler determines the type of a variable or function without the help of a programmer. For example, 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 the type inference works, let's 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 with a Scala interpreter, you'll see the following output:

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

Even though we 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, then 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 object-oriented programming into
one language (look at section 1.1.3 for the details). Scala is the first statically typed language to fuse functional and object-oriented programming into one language. Scala has made some innovations in object-oriented programming (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 is supported for both JVM (Java Virtual Machine) and CLR (Common Language Runtime). Some would argue that Scala's syntax is more dissimilar than similar to that of Java/C#. You saw some Scala code already in previous sections, so I'll let you 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, and these are
borrowed from Algol/Simula and Eiffel, respectively. For example, in Scala variables and functions

without
parameters are accessed 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 we'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 ML (metalanguage) family of languages, and Scala's Actor library is influenced by Erlang's Actor model.

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

1.6 Summary

In this chapter we quickly went over lots of interesting concepts, but don't worry because we're going to
reiterate these concepts throughout the book with plenty of examples so that you can relate them to realworld
problems. You learned what is Scala and why you should consider learning Scala as your next programming language. Scala's extensible and scalable features make it a language that you could fit into
small to large programming problems. Its multi-paradigm model provides programmers with the power of

abstractions from both functional and object-oriented programming 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 we'll set up
our development environment and get our hands dirty with Scala code and syntax.