Chapter 1

A Scalable Language

The name Scala stands for "scalable language." The language is so named because it was designed to grow with the demands of its users. You can apply Scala to a wide range of programming tasks, from writing small scripts to building large systems.[1]

Scala is easy to get into. It runs on the standard Java platform and interoperates seamlessly with all Java libraries. It's quite a good language for writing scripts that pull together Java components. But it can apply its strengths even more when used for building large systems and frameworks of reusable components.

Technically, Scala is a blend of object-oriented and functional programming concepts in a statically typed language. The fusion of object-oriented and functional programming shows up in many different aspects of Scala; it is probably more pervasive than in any other widely used language. The two programming styles have complementary strengths when it comes to scalability. Scala's functional programming constructs make it easy to build interesting things quickly from simple parts. Its object-oriented constructs make it easy to structure larger systems and adapt them to new demands. The combination of both styles in Scala makes it possible to express new kinds of programming patterns and component abstractions. It also leads to a legible and concise programming style. And because it is so malleable, programming in Scala can be a lot of fun.

This initial chapter answers the question, "Why Scala?" It gives a high-level view of Scala's design and the reasoning behind it. After reading the chapter you should have a basic feel for what Scala is and what kinds of tasks it might help you accomplish. Although this book is a Scala tutorial, this chapter isn't really part of the tutorial. If you're eager to start writing some Scala code, you should jump ahead to Chapter 2.

1.1 A LANGUAGE THAT GROWS ON YOU

Programs of different sizes tend to require different programming constructs. Consider, for example, the following small Scala program:

```
var capital = Map("US" -> "Washington", "France" -> "Paris")

capital += ("Japan" -> "Tokyo")

println(capital("France"))
```

This program sets up a map from countries to their capitals, modifies the map by adding a new binding ("Japan" -> "Tokyo"), and prints the capital associated with the country France.[2]The notation in this example is high level, to the point, and not cluttered with extraneous semicolons or type annotations. Indeed, the feel is that of a modern "scripting" language like Perl, Python, or Ruby. One

common characteristic of these languages, which is relevant for the example above, is that they each support an "associative map" construct in the syntax of the language.

Associative maps are very useful because they help keep programs legible and concise, but sometimes you might not agree with their "one size fits all" philosophy because you need to control the properties of the maps you use in your program in a more fine-grained way. Scala gives you this fine-grained control if you need it, because maps in Scala are not language syntax. They are library abstractions that you can extend and adapt.

In the above program, you'll get a default Map implementation, but you can easily change that. You could for example specify a particular implementation, such as a HashMap or a TreeMap, or invoke the par method to obtain a ParMap that executes operations in parallel. You could specify a default value for the map, or you could override any other method of the map you create. In each case, you can use the same easy access syntax for maps as in the example above.

This example shows that Scala can give you both convenience and flexibility. Scala has a set of convenient constructs that help you get started quickly and let you program in a pleasantly concise style. At the same time, you have the assurance that you will not outgrow the language. You can always tailor the program to your requirements, because everything is based on library modules that you can select and adapt as needed.

Growing new types

Eric Raymond introduced the cathedral and bazaar as two metaphors of software development.[3] The cathedral is a near-perfect building that takes a long time to build. Once built, it stays unchanged for a long time. The bazaar, by contrast, is adapted and extended each day by the people working in it. In Raymond's work the bazaar is a metaphor for open-source software development. Guy Steele noted in a talk on "growing a language" that the same distinction can be applied to language design.[4] Scala is much more like a bazaar than a cathedral, in the sense that it is designed to be extended and adapted by the people programming in it. Instead of providing all constructs you might ever need in one "perfectly complete" language, Scala puts the tools for building such constructs into your hands.

Here's an example. Many applications need a type of integer that can become arbitrarily large without overflow or "wrap-around" of arithmetic operations. Scala defines such a type in library class scala.BigInt. Here is the definition of a method using that type, which calculates the factorial of a passed integer value:[5]

```
def factorial(x: BigInt): BigInt =
  if (x == 0) 1 else x * factorial(x - 1)
```

Now, if you call factorial(30) you would get:

```
265252859812191058636308480000000
```

BigInt looks like a built-in type because you can use integer literals and operators such as *and - with values of that type. Yet it is just a class that happens to be defined in Scala's standard library.[6] If the

class were missing, it would be straightforward for any Scala programmer to write an implementation, for instance, by wrapping Java's classjava.math.BigInteger (in fact that's how Scala's BigInt class is implemented).

Of course, you could also use Java's class directly. But the result is not nearly as pleasant, because although Java allows you to create new types, those types don't feel much like native language support:

```
import java.math.BigInteger

def factorial(x: BigInteger): BigInteger =
  if (x == BigInteger.ZERO)
    BigInteger.ONE
  else
    x.multiply(factorial(x.subtract(BigInteger.ONE)))
```

BigInt is representative of many other number-like types—big decimals, complex numbers, rational numbers, confidence intervals, polynomials—the list goes on. Some programming languages implement some of these types natively. For instance, Lisp, Haskell, and Pythonimplement big integers; Fortran and Python implement complex numbers. But any language that attempted to implement all of these abstractions at the same time would simply become too big to be manageable. What's more, even if such a language were to exist, some applications would surely benefit from other number-like types that were not supplied. So the approach of attempting to provide everything in one language doesn't scale very well. Instead, Scala allows users to grow and adapt the language in the directions they need by defining easy-to-use libraries that feel like native language support.

Growing new control constructs

The previous example demonstrates that Scala lets you add new types that can be used as conveniently as built-in types. The same extension principle also applies to control structures. This kind of extensibility is illustrated by Akka, a Scala API for "actor-based" concurrent programming.

As multicore processors continue to proliferate in the coming years, achieving acceptable performance may increasingly require that you exploit more parallelism in your applications. Often, this will mean rewriting your code so that computations are distributed over several concurrent threads. Unfortunately, creating dependable multi-threaded applications has proven challenging in practice. Java's threading model is built around shared memory and locking, a model that is often difficult to reason about, especially as systems scale up in size and complexity. It is hard to be sure you don't have a race condition or deadlock lurking—something that didn't show up during testing, but might just show up in production. An arguably safer alternative is a message passing architecture, such as the "actors" approach used by the Erlang programming language.

Java comes with a rich, thread-based concurrency library. Scala programs can use it like any other Java API. However, Akka is an additional Scala library that implements an actor model similar to Erlang's.

Actors are concurrency abstractions that can be implemented on top of threads. They communicate by sending messages to each other. An actor can perform two basic operations, message send and receive.

The send operation, denoted by an exclamation point (!), sends a message to an actor. Here's an example in which the actor is named recipient:

```
recipient ! msg
```

A send is asynchronous; that is, the sending actor can proceed immediately, without waiting for the message to be received and processed. Every actor has a mailbox in which incoming messages are queued. An actor handles messages that have arrived in its mailbox via a receiveblock:

```
def receive = {
  case Msg1 => ... // handle Msg1
  case Msg2 => ... // handle Msg2
  // ...
}
```

A receive block consists of a number of cases that each query the mailbox with a message pattern. The first message in the mailbox that matches any of the cases is selected, and the corresponding action is performed on it. Once the mailbox does not contain any messages, the actor suspends and waits for further incoming messages.

As an example, here is a simple Akka actor implementing a checksum calculator service:

```
class ChecksumActor extends Actor {
  var sum = 0
  def receive = {
    case Data(byte) => sum += byte
    case GetChecksum(requester) =>
      val checksum = ~(sum & 0xFF) + 1
      requester ! checksum
  }
}
```

This actor first defines a local variable named sum with initial value zero. It defines a receiveblock that will handle messages. If it receives a Data message, it adds the contained byte to thesum variable. If it receives a GetChecksum message, it calculates a checksum from the current value of sum and sends the result back to the requester using the message send requester ! sum. The requester field is embedded in the GetChecksum message; it usually refers to the actor that made the request.

We don't expect you to fully understand the actor example at this point. Rather, what's significant about this example for the topic of scalability is that neither the receive block nor message send (!) are built-in operations in Scala. Even though the receive block may look and act very much like a built-in control construct, it is in fact a method defined in Akka's actors library. Likewise, even though `!' looks like a built-in operator, it too is just a method defined in the Akka actors library. Both of these constructs are completely independent of the Scala programming language.

The receive block and send (!) syntax look in Scala much like they look in Erlang, but in Erlang, these constructs are built into the language. Akka also implements most of Erlang's other concurrent programming constructs, such as monitoring failed actors and time-outs. All in all, the actor model has

turned out to be a very pleasant means for expressing concurrent and distributed computations. Even though they must be defined in a library, actors can feel like an integral part of the Scala language.

This example illustrates that you can "grow" the Scala language in new directions even as specialized as concurrent programming. To be sure, you need good architects and programmers to do this. But the crucial thing is that it is feasible—you can design and implement abstractions in Scala that address radically new application domains, yet still feel like native language support when used.

1.2 WHAT MAKES SCALA SCALABLE?

Scalability is influenced by many factors, ranging from syntax details to component abstraction constructs. If we were forced to name just one aspect of Scala that helps scalability, though, we'd pick its combination of object-oriented and functional programming (well, we cheated, that's really two aspects, but they are intertwined).

Scala goes further than all other well-known languages in fusing object-oriented and functional programming into a uniform language design. For instance, where other languages might have objects and functions as two different concepts, in Scala a function value is an object. Function types are classes that can be inherited by subclasses. This might seem nothing more than an academic nicety, but it has deep consequences for scalability. In fact the actor concept shown previously could not have been implemented without this unification of functions and objects. This section gives an overview of Scala's way of blending object-oriented and functional concepts.

Scala is object-oriented

Object-oriented programming has been immensely successful. Starting from Simula in the mid-60s and Smalltalk in the 70s, it is now available in more languages than not. In some domains, objects have taken over completely. While there is not a precise definition of what object-oriented means, there is clearly something about objects that appeals to programmers.

In principle, the motivation for object-oriented programming is very simple: all but the most trivial programs need some sort of structure. The most straightforward way to do this is to put data and operations into some form of containers. The great idea of object-oriented programming is to make these containers fully general, so that they can contain operations as well as data, and that they are themselves values that can be stored in other containers, or passed as parameters to operations. Such containers are called objects. Alan Kay, the inventor of Smalltalk, remarked that in this way the simplest object has the same construction principle as a full computer: it combines data with operations under a formalized interface.[7] So objects have a lot to do with language scalability: the same techniques apply to the construction of small as well as large programs.

Even though object-oriented programming has been mainstream for a long time, there are relatively few languages that have followed Smalltalk in pushing this construction principle to its logical conclusion. For instance, many languages admit values that are not objects, such as the primitive values in Java. Or they allow static fields and methods that are not members of any object. These deviations

from the pure idea of object-oriented programming look quite harmless at first, but they have an annoying tendency to complicate things and limit scalability.

By contrast, Scala is an object-oriented language in pure form: every value is an object and every operation is a method call. For example, when you say 1 + 2 in Scala, you are actually invoking a method named + defined in class Int. You can define methods with operator-like names that clients of your API can then use in operator notation. This is how the designer of Akka's actors API enabled you to use expressions such as requester ! sum shown in the previous example: `!' is a method of the Actor class.

Scala is more advanced than most other languages when it comes to composing objects. An example is Scala's *trait*s. Traits are like interfaces in Java, but they can also have method implementations and even fields.[8] Objects are constructed by mixin composition, which takes the members of a class and adds the members of a number of traits to them. In this way, different aspects of classes can be encapsulated in different traits. This looks a bit like multiple inheritance, but differs when it comes to the details. Unlike a class, a trait can add some new functionality to an unspecified superclass. This makes traits more "pluggable" than classes. In particular, it avoids the classical "diamond inheritance" problems of multiple inheritance, which arise when the same class is inherited via several different paths.

Scala is functional

In addition to being a pure object-oriented language, Scala is also a full-blown functional language. The ideas of functional programming are older than (electronic) computers. Their foundation was laid in Alonzo Church's lambda calculus, which he developed in the 1930s. The first functional programming language was Lisp, which dates from the late 50s. Other popular functional languages are Scheme, SML, Erlang, Haskell, OCaml, and F#. For a long time, functional programming has been a bit on the sidelines—popular in academia, but not that widely used in industry. However, in recent years, there has been an increased interest in functional programming languages and techniques.

Functional programming is guided by two main ideas. The first idea is that functions are first-class values. In a functional language, a function is a value of the same status as, say, an integer or a string. You can pass functions as arguments to other functions, return them as results from functions, or store them in variables. You can also define a function inside another function, just as you can define an integer value inside a function. And you can define functions without giving them a name, sprinkling your code with function literals as easily as you might write integer literals like 42.

Functions that are first-class values provide a convenient means for abstracting over operations and creating new control structures. This generalization of functions provides great expressiveness, which often leads to very legible and concise programs. It also plays an important role for scalability. As an example, the ScalaTest testing library offers an eventuallyconstruct that takes a function as an argument. It is used like this:

```
val xs = 1 to 3
val it = xs.iterator
```

```
eventually { it.next() shouldBe 3 }
```

The code inside eventually—the assertion, it.next() shouldBe 3—is wrapped in a function that is passed unexecuted to the eventually method. For a configured amount of time, eventually will repeatedly execute the function until the assertion succeeds.

In most traditional languages, by contrast, functions are not values. Languages that do have function values often relegate them to second-class status. For example, the function pointers of C and C++ do not have the same status as non-functional values in those languages: Function pointers can only refer to global functions, they do not allow you to define first-class nested functions that refer to some values in their environment. Nor do they allow you to define unnamed function literals.

The second main idea of functional programming is that the operations of a program should map input values to output values rather than change data in place. To see the difference, consider the implementation of strings in Ruby and Java. In Ruby, a string is an array of characters. Characters in a string can be changed individually. For instance you can change a semicolon character in a string to a period inside the same string object. In Java and Scala, on the other hand, a string is a sequence of characters in the mathematical sense. Replacing a character in a string using an expression like s.replace(';', '.') yields a new string object, which is different from s. Another way of expressing this is that strings are immutable in Java whereas they are mutable in Ruby. So looking at just strings, Java is a functional language, whereas Ruby is not. Immutable data structures are one of the cornerstones of functional programming. The Scala libraries define many more immutable data types on top of those found in the Java APIs. For instance, Scala has immutable lists, tuples, maps, and sets.

Another way of stating this second idea of functional programming is that methods should not have any side effects. They should communicate with their environment only by taking arguments and returning results. For instance, the replace method in Java's String class fits this description. It takes a string and two characters and yields a new string where all occurrences of one character are replaced by the other. There is no other effect of callingreplace. Methods like replace are called *referentially transparent*, which means that for any given input the method call could be replaced by its result without affecting the program's semantics.

Functional languages encourage immutable data structures and referentially transparent methods. Some functional languages even require them. Scala gives you a choice. When you want to, you can write in an imperative style, which is what programming with mutable data and side effects is called. But Scala generally makes it easy to avoid imperative constructs when you want because good functional alternatives exist.

1.3 WHY SCALA?

Is Scala for you? You will have to see and decide for yourself. We have found that there are actually many reasons besides scalability to like programming in Scala. Four of the most important aspects will be discussed in this section: compatibility, brevity, high-level abstractions, and advanced static typing.

Scala is compatible

Scala doesn't require you to leap backwards off the Java platform to step forward from the Java language. It allows you to add value to existing code—to build on what you already have—because it was designed for seamless interoperability with Java.[9] Scala programs compile to JVM bytecodes. Their run-time performance is usually on par with Java programs. Scala code can call Java methods, access Java fields, inherit from Java classes, and implement Java interfaces. None of this requires special syntax, explicit interface descriptions, or glue code. In fact, almost all Scala code makes heavy use of Java libraries, often without programmers being aware of this fact.

Another aspect of full interoperability is that Scala heavily re-uses Java types. Scala's Ints are represented as Java primitive integers of type int, Floats are represented as floats, Booleans asbooleans, and so on. Scala arrays are mapped to Java arrays. Scala also re-uses many of the standard Java library types. For instance, the type of a string literal "abc" in Scala isjava.lang.String, and a thrown exception must be a subclass of java.lang.Throwable.

Scala not only re-uses Java's types, but also "dresses them up" to make them nicer. For instance, Scala's strings support methods like toInt or toFloat, which convert the string to an integer or floating-point number. So you can write str.toInt instead of Integer.parseInt(str). How can this be achieved without breaking interoperability? Java's String class certainly has no toInt method! In fact, Scala has a very general solution to solve this tension between advanced library design and interoperability. Scala lets you define implicit conversions,which are always applied when types would not normally match up, or when non-existing members are selected. In the case above, when looking for a toInt method on a string, the Scala compiler will find no such member of class String, but it will find an implicit conversion that converts a Java String to an instance of the Scala class StringOps, which does define such a member. The conversion will then be applied implicitly before performing the toIntoperation.

Scala code can also be invoked from Java code. This is sometimes a bit more subtle, because Scala is a richer language than Java, so some of Scala's more advanced features need to be encoded before they can be mapped to Java. Chapter 31 explains the details.

Scala is concise

Scala programs tend to be short. Scala programmers have reported reductions in number of lines of up to a factor of ten compared to Java. These might be extreme cases. A more conservative estimate would be that a typical Scala program should have about half the number of lines of the same program written in Java. Fewer lines of code mean not only less typing, but also less effort at reading and understanding programs and fewer possibilities of defects. There are several factors that contribute to this reduction in lines of code.

First, Scala's syntax avoids some of the boilerplate that burdens Java programs. For instance, semicolons are optional in Scala and are usually left out. There are also several other areas where Scala's syntax is less noisy. As an example, compare how you write classes and constructors in Java and Scala. In Java, a class with a constructor often looks like this:

```
// this is Java
```

```
class MyClass {

    private int index;
    private String name;

    public MyClass(int index, String name) {
        this.index = index;
        this.name = name;
    }
}
```

In Scala, you would likely write this instead:

```
class MyClass(index: Int, name: String)
```

Given this code, the Scala compiler will produce a class that has two private instance variables, an Int named index and a String named name, and a constructor that takes initial values for those variables as parameters. The code of this constructor will initialize the two instance variables with the values passed as parameters. In short, you get essentially the same functionality as the more verbose Java version.[10] The Scala class is quicker to write, easier to read, and most importantly, less error prone than the Java class.

Scala's type inference is another factor that contributes to its conciseness. Repetitive type information can be left out, so programs become less cluttered and more readable.

But probably the most important key to compact code is code you don't have to write because it is done in a library for you. Scala gives you many tools to define powerful libraries that let you capture and factor out common behavior. For instance, different aspects of library classes can be separated out into traits, which can then be mixed together in flexible ways. Or, library methods can be parameterized with operations, which lets you define constructs that are, in effect, your own control structures. Together, these constructs allow the definition of libraries that are both high-level and flexible to use.

Scala is high-level

Programmers are constantly grappling with complexity. To program productively, you must understand the code on which you are working. Overly complex code has been the downfall of many a software project. Unfortunately, important software usually has complex requirements. Such complexity can't be avoided; it must instead be managed.

Scala helps you manage complexity by letting you raise the level of abstraction in the interfaces you design and use. As an example, imagine you have a String variable name, and you want to find out whether or not that String contains an upper case character. Prior to Java 8, you might have written a loop, like this:

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

```
}
```

Whereas in Scala, you could write this:

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

The Java code treats strings as low-level entities that are stepped through character by character in a loop. The Scala code treats the same strings as higher-level sequences of characters that can be queried with *predicates*. Clearly the Scala code is much shorter and—for trained eyes—easier to understand than the Java code. So the Scala code weighs less heavily on the total complexity budget. It also gives you less opportunity to make mistakes.

The predicate _.isUpper is an example of a function literal in Scala.[11] It describes a function that takes a character argument (represented by the underscore character) and tests whether it is an upper case letter.[12]

Java 8 introduced support for lambdas and streams, which enable you to perform a similar operation in Java. Here's what it might look like:

```
boolean nameHasUpperCase =   // This is Java 8
    name.chars().anyMatch(
        (int ch) -> Character.isUpperCase((char) ch)
    );
```

Although a great improvement over earlier versions of Java, the Java 8 code is still more verbose than the equivalent Scala code. This extra "heaviness" of Java code, as well as Java's long tradition of loops, may encourage many Java programmers in need of new methods likeexists to just write out loops and live with the increased complexity in their code.

On the other hand, function literals in Scala are really lightweight, so they are used frequently. As you get to know Scala better you'll find more and more opportunities to define and use your own control abstractions. You'll find that this helps avoid code duplication and thus keeps your programs shorter and clearer.

Scala's functional programming style also offers high-level reasoning principles for programming. The key idea is that functions are referentially transparent—a function application is characterized only by its result. You can, therefore, freely exchange a function application with the function's right hand side (*i.e.*, its body, which follows the equals sign) without worrying about any hidden side effects. This principle gives many useful laws that you can employ to better understand or to refactor your code. As an example, take once more the exists method described above. This method should satisfy the following law: for every sequence s and for every pair of predicates p and q it should hold that

```
s.exists(p) || s.exists(q) == s.exists(x => p(x) || q(x))
```

That is, querying the same sequence with two predicates p and q and or-ing the results is the same as querying with a single predicate that tests at the same time for p or for q. A law like this is clearly useful for writing and refactoring programs. However, if exists had side effects, it would in general not

be correct to assume such a law because the left hand side executesexists twice for each sequence element whereas the right hand side executes it only once per element. So this is an example where purely functional code leads to more laws that are useful for understanding and refactoring your code.

The functional programming style also eliminates aliasing problems encountered in imperative programming. Aliasing happens when multiple variables refer to the same object. It gives rise to some thorny questions and complications. For instance, does changing a fieldr.x also affect s.x? It does if r and s refer to the same object. In practice it is often very difficult to trace such aliases. Immutable data, on the other hand, can be shared freely, because a copy is indistinguishable from a shared reference. This advantage is particularly crucial when writing concurrent code. (This is why Java has immutable strings.)

Scala is statically typed

A static type system classifies variables and expressions according to the kinds of values they hold and compute. Scala stands out as a language with a very advanced static type system. Starting from a system of nested class types much like Java's, it allows you to parameterize types with generics, to combine types using intersections, and to hide details of types usingabstract types.[13] These give a strong foundation for building and composing your own types, so that you can design interfaces that are at the same time safe and flexible to use.

If you like dynamic languages, such as Perl, Python, Ruby, or Groovy, you might find it a bit strange that Scala's static type system is listed as one of its strong points. After all, the absence of a static type system has been cited by some as a major advantage of dynamic languages. The most common arguments against static types are that they make programs too verbose, prevent programmers from expressing themselves as they wish, and make impossible certain patterns of dynamic modifications of software systems. However, often these arguments do not go against the idea of static types in general, but against specific type systems, which are perceived to be too verbose or too inflexible. For instance, Alan Kay, the inventor of the Smalltalk language, once remarked: "I'm not against types, but I don't know of any type systems that aren't a complete pain, so I still like dynamic typing."[14]

We hope to convince you in this book that Scala's type system is far from being a "complete pain." In fact, it addresses nicely two of the usual concerns about static typing: Verbosity is avoided through type inference, and flexibility is gained through pattern matching and several new ways to write and compose types. With these impediments out of the way, the classical benefits of static type systems can be better appreciated. Among the most important of these benefits are verifiable properties of program abstractions, safe refactorings, and better documentation.

Verifiable properties. Static type systems can prove the absence of certain run-time errors. For instance, they can prove properties like: Booleans are never added to integers; private variables are not accessed from outside their class; functions are applied to the right number of arguments; only strings are ever added to a set of strings.

Other kinds of errors are not detected by today's static type systems. For instance, they will usually not detect non-terminating functions, array bounds violations, or divisions by zero. They will also not

detect that your program does not conform to its specification (assuming there is a spec, that is!). Static type systems have therefore been dismissed by some as not being very useful. The argument goes that since such type systems can only detect simple errors, whereas unit tests provide more extensive coverage, why bother with static types at all? We believe that these arguments miss the point. Although a static type system certainly cannot *replace* unit testing, it can reduce the number of unit tests needed by taking care of some properties that would otherwise need to be tested. Likewise, unit testing cannot replace static typing. After all, as Edsger Dijkstra said, testing can only prove the presence of errors, never their absence.[15] So the guarantees that static typing gives may be simple, but they are real guarantees of a form no amount of testing can deliver.

Safe refactorings. A static type system provides a safety net that lets you make changes to a codebase with a high degree of confidence. Consider for instance a refactoring that adds an additional parameter to a method. In a statically typed language you can do the change, re-compile your system, and simply fix all lines that cause a type error. Once you have finished with this, you are sure to have found all places that need to be changed. The same holds for many other simple refactorings, like changing a method name or moving methods from one class to another. In all cases a static type check will provide enough assurance that the new system works just like the old.

Documentation. Static types are program documentation that is checked by the compiler for correctness. Unlike a normal comment, a type annotation can never be out of date (at least not if the source file that contains it has recently passed a compiler). Furthermore, compilers and integrated development environments (IDEs) can make use of type annotations to provide better context help. For instance, an IDE can display all the members available for a selection by determining the static type of the expression on which the selection is made and looking up all members of that type.

Even though static types are generally useful for program documentation, they can sometimes be annoying when they clutter the program. Typically, useful documentation is what readers of a program cannot easily derive by themselves. In a method definition like:

```
def f(x: String) = ...
```

it's useful to know that f's argument should be a String. On the other hand, at least one of the two annotations in the following example is annoying:

```
val x: HashMap[Int, String] = new HashMap[Int, String]()
```

Clearly, it should be enough to say just once that x is a HashMap with Ints as keys and Strings as values; there's no need to repeat the same phrase twice.

Scala has a very sophisticated type inference system that lets you omit almost all type information that's usually considered annoying. In the previous example, the following two less annoying alternatives would work just as well:

```
val x = new HashMap[Int, String]()
val x: Map[Int, String] = new HashMap()
```

Type inference in Scala can go quite far. In fact, it's not uncommon for user code to have no explicit types at all. Therefore, Scala programs often look a bit like programs written in a dynamically typed scripting language. This holds particularly for client application code, which glues together pre-written library components. It's less true for the library components themselves, because these often employ fairly sophisticated types to allow flexible usage patterns. This is only natural. After all, the type signatures of the members that make up the interface of a reusable component should be explicitly given, because they constitute an essential part of the contract between the component and its clients.

1.4 SCALA'S ROOTS

Scala's design has been influenced by many programming languages and ideas in programming language research. In fact, only a few features of Scala are genuinely new; most have been already applied in some form in other languages. Scala's innovations come primarily from how its constructs are put together. In this section, we list the main influences on Scala's design. The list cannot be exhaustive—there are simply too many smart ideas around in programming language design to enumerate them all here.

At the surface level, Scala adopts a large part of the syntax of Java and C#, which in turn borrowed most of their syntactic conventions from C and C++. Expressions, statements, and blocks are mostly as in Java, as is the syntax of classes, packages and imports.[16] Besides syntax, Scala adopts other elements of Java, such as its basic types, its class libraries, and its execution model.

Scala also owes much to other languages. Its uniform object model was pioneered by Smalltalk and taken up subsequently by Ruby. Its idea of universal nesting (almost every construct in Scala can be nested inside any other construct) is also present in Algol, Simula, and, more recently, in Beta and gbeta. Its uniform access principle for method invocation and field selection comes from Eiffel. Its approach to functional programming is quite similar in spirit to the ML family of languages, which has SML, OCaml, and F# as prominent members. Many higher-order functions in Scala's standard library are also present in ML or Haskell. Scala's implicit parameters were motivated by Haskell's type classes; they achieve analogous results in a more classical object-oriented setting. Scala's main actor-based concurrency library, Akka, was heavily inspired by Erlang.

Scala is not the first language to emphasize scalability and extensibility. The historic root of extensible languages that can span different application areas is Peter Landin's 1966 paper,"The Next 700 Programming Languages."[17] (The language described in this paper, Iswim, stands beside Lisp as one of the pioneering functional languages.) The specific idea of treating an infix operator as a function can be traced back to Iswim and Smalltalk. Another important idea is to permit a function literal (or block) as a parameter, which enables libraries to define control structures. Again, this goes back to Iswim and Smalltalk. Smalltalk and Lisp both have a flexible syntax that has been applied extensively for building internal domain-specific languages. C++ is another scalable language that can be adapted and extended through operator overloading and its template system; compared to Scala it is built on a lower-level, more systems-oriented core. Scala is also not the first language to integrate functional and object-oriented programming, although it probably goes furthest in this direction. Other languages that have

integrated some elements of functional programming into object-oriented programming (OOP) include Ruby, Smalltalk, and Python. On the Java platform, Pizza, Nice, Multi-Java—and Java 8 itself—have all extended a Java-like core with functional ideas. There are also primarily functional languages that have acquired an object system; examples are OCaml, F#, and PLT-Scheme.

Scala has also contributed some innovations to the field of programming languages. For instance, its abstract types provide a more object-oriented alternative to generic types, its traits allow for flexible component assembly, and its extractors provide a representation-independent way to do pattern matching. These innovations have been presented in papers at programming language conferences in recent years.[18]

1.5 CONCLUSION

In this chapter, we gave you a glimpse of what Scala is and how it might help you in your programming. To be sure, Scala is not a silver bullet that will magically make you more productive. To advance, you will need to apply Scala artfully, and that will require some learning and practice. If you're coming to Scala from Java, the most challenging aspects of learning Scala may involve Scala's type system (which is richer than Java's) and its support for functional programming. The goal of this book is to guide you gently up Scala's learning curve, one step at a time. We think you'll find it a rewarding intellectual experience that will expand your horizons and make you think differently about program design. Hopefully, you will also gain pleasure and inspiration from programming in Scala.

In the next chapter, we'll get you started writing some Scala code.

Footnotes for Chapter 1:

[1] Scala is pronounced *skah-lah*.

[2] Please bear with us if you don't understand all the details of this program. They will be explained in the next two chapters.

[3] Raymond, *The Cathedral and the Bazaar*. [Ray99]

[4] Steele, "Growing a language." [Ste99]

[5] factorial(x), or x! in mathematical notation, is the result of computing 1 * 2 * ... * x, with 0!defined to be 1.

[6] Scala comes with a standard library, some of which will be covered in this book. For more information, you can also consult the library's Scaladoc documentation, which is available in the distribution and online at http://www.scala-lang.org.

[7] Kay, "The Early History of Smalltalk." [Kay96]

[8] Starting with Java 8, interfaces can have default method implementations, but these do not offer all the features of Scala's traits.

[9] Originally, there was an implementation of Scala that ran on the .NET platform, but it is no longer active. More recently, an implementation of Scala that runs on JavaScript, Scala.js,has become increasingly popular.

[10] The only real difference is that the instance variables produced in the Scala case will be final. You'll learn how to make them non-final in Section 10.6.

[11] A function literal can be called a *predicate* if its result type is Boolean.

[12] This use of the underscore as a placeholder for arguments is described in Section 8.5.

[13] Generics are discussed in Chapter 19; intersections (*e.g.*, A with B with C) in Chapter 12; and abstract types in Chapter 20.

[14] Kay, in an email on the meaning of object-oriented programming. [Kay03]

[15] Dijkstra, "Notes on Structured Programming." [Dij70]

[16] The major deviation from Java concerns the syntax for type annotations: it's "variable: Type" instead of "Type variable" in Java. Scala's postfix type syntax resembles Pascal, Modula-2, or Eiffel. The main reason for this deviation has to do with type inference, which often lets you omit the type of a variable or the return type of a method. Using the "variable: Type" syntax is easy—just leave out the colon and the type. But in C-style "Type variable" syntax you cannot simply leave off the type; there would be no marker to start the definition anymore. You'd need some alternative keyword to be a placeholder for a missing type (C# 3.0, which does some type inference, uses var for this purpose). Such an alternative keyword feels more ad-hoc and less regular than Scala's approach.

[17] Landin, "The Next 700 Programming Languages." [Lan66]

[18] For more information, see [Ode03], [Ode05], and [Emi07] in the bibliography.