

Scala

IN DEPTH

Joshua D. Suereth





**MEAP Edition
Manning Early Access Program
Scala in Depth Production Version**

Copyright 2012 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

1. Scala—a blended language
2. The core rules
3. Modicum of style—coding conventions
4. Utilizing object orientation
5. Using implicits to write expressive code
6. The type system
7. Using implicits and types together
8. Using the right collection
9. Actors
10. Integrating Scala with Java
11. Patterns in functional programming

Scala— *a blended language*

In this chapter

- Short introduction to Scala
- Insights into Scala's design

Scala was born from the mind of Martin Odersky, a man who had helped introduce generics into the Java programming language. Scala was an offshoot from the Funnel language, an attempt to combine functional programming and Petri nets. Scala was developed with the premise that you could mix together object orientation, functional programming, and a powerful type system and still keep elegant, succinct code. It was hoped that this blending of concepts would create something that real developers could use and that could be studied for new programming idioms. It was such a large success that industry has started adopting Scala as a viable and competitive language.

Understanding Scala requires understanding this mixture of concepts. Scala attempts to blend three dichotomies of thought into one language. These are:

- Functional programming and object-oriented programming
- Expressive syntax and static typing
- Advanced language features and rich Java integration

Functional programming is programming through the definition and composition of functions. Object-oriented programming is programming through the definition and composition of objects. In Scala, functions *are* objects. Programs can be constructed through both the definition and composition of objects or functions. This gives Scala the ability to focus on “nouns” or “verbs” in a program, depending on what is the most prominent.

Scala also blends expressive syntax with static typing. Mainstream statically typed languages tend to suffer from verbose type annotations and boilerplate syntax. Scala takes a few lessons from the ML programming language and offers static typing with a nice expressive syntax. Code written in Scala can look as expressive as dynamically typed languages, like Ruby, while retaining type safety.

Finally, Scala offers a lot of advanced language features that are not available in Java. But Scala runs on the Java virtual machine (JVM) and has tight integration with the Java language. This means that developers can make direct use of existing Java libraries and integrate Scala into their Java applications while also gaining the additional power of Scala. This integration makes Scala a practical choice for any JVM-based project.

Let’s take a deeper look at the blending of paradigms in Scala.

1.1 Functional programming meets object orientation

Functional programming and object-oriented programming are two different ways of looking at a problem. Functional programming puts special emphasis on the “verbs” of a program and ways to combine and manipulate them. Object-oriented programming puts special emphasis on “nouns” and attaches verbs to them. The two approaches are almost inverses of each other, with one being “top down” and the other “bottom up.”

Object-oriented programming is a top-down approach to code design. It approaches software by dividing code into nouns or objects. Each object has some form of identity (self/this), behavior (methods), and state (members). After identifying nouns and defining their behaviors, interactions between nouns are defined. The problem with implementing interactions is that the interactions need to live inside an object. Modern object-oriented designs tend to have *service classes*, which are a collection of methods that operate across several domain objects. Service classes, although objects, usually don’t have a notion of state or behavior independent of the objects on which they operate.

A good example is a program that implements the following story: “A cat catches a bird and eats it.” An object-oriented programmer would look at this sentence and see two nouns: cat and bird. The cat has two verbs associated with it: catch and eat. The following program is a more object-oriented approach:

```
class Bird
class Cat {
  def catch(b: Bird): Unit = ...
  def eat(): Unit = ...
```

```
}  
  
val cat = new Cat  
val bird = new Bird  
  
cat.catch(bird)  
cat.eat()
```

In the example, when a `Cat` catches a `Bird`, it converts the bird to a type of `Food`, which it can then eat. The code focuses on the nouns and their actions: `Cat.eat()`, `Cat.catch(...)`. In functional programming, the focus is on the verbs.

Functional programming approaches software as the combination and application of functions. It tends to decompose software into behaviors, or actions that need to be performed, usually in a bottom-up fashion. Functions are viewed in a mathematical sense, purely operations on their input. All variables are considered immutable. This immutability aids concurrent programming. Functional programming attempts to defer all side effects in a program as long as possible. Removing side effects makes reasoning through a program simpler, in a formal sense. It also provides much more power in how things can be abstracted and combined.

In the story “A cat catches a bird and eats it,” a functional program would see the two verbs *catch* and *eat*. A program would create these two functions and compose them to create the program. The following program is a more functional approach:

```
trait Cat  
trait Bird  
trait Catch  
trait FullTummy  
  
def catch(hunter: Cat, prey: Bird): Cat with Catch  
def eat(consumer: Cat with Catch): Cat with FullTummy  
  
val story = (catch _) andThen (eat _)  
story(new Cat, new Bird)
```

In the example, the `catch` method takes a `Cat` and a `Bird` and returns a new value of type `Cat with Catch`. The `eat` method is defined as taking a `CatWithPrey` (a cat needs something to eat) and returns a `FullCat` (because it's no longer hungry). Functional programming makes more use of the type system to describe what a function is doing. The `catch` and `eat` methods use the type signatures to define the expected input and output states of the function. The `with` keyword is used to combine a type with another. In this example, the traits `Catch` and `FullTummy` are used to denote the current state of a `Cat`. The methods `eat` and `catch` return new instances of `Cat` attached to different state types. The `story` value is created by composing the functions `catch` and `eat`. This means that the `catch` method is called and the result is fed into the `eat` method. Finally, the `story` function is called with a `Cat` and a `Bird` and the result is the output of the story: a full cat.

Functional programming and object orientation offer unique views of software. It's these differences that make them useful to each other. Object orientation can deal with composing the nouns and functional programming can deal with composing

Table 1.1 Attributes commonly ascribed to object-oriented and functional programming

Object-oriented programming	Functional programming
Composition of objects (nouns)	Composition of functions (verbs)
Encapsulated stateful interaction	Deferred side effects
Iterative algorithms	Recursive algorithms and continuations
Imperative flow	Lazy evaluation
N/A	Pattern matching

verbs. In the example, the functional version was built by composing a set of functions that encompassed a story and then feeding the initial data into these functions. For the object-oriented version, a set of objects was created and their internal state was manipulated. Both approaches are useful in designing software. Object orientation can focus on the nouns of the system and functional programming can compose the verbs.

In fact, in recent years, many Java developers have started moving toward splitting nouns and verbs. The Enterprise JavaBeans (EJB) specification splits software into *Session beans*, which tend to contain behaviors, and *Entity beans*, which tend to model the nouns in the system. Stateless Session beans start looking more like collections of functional code (although missing most of the useful features of functional code).

This push of functional style has come along much further than the EJB specifications. The Spring Application Framework promotes a functional style with its Template classes, and the Google Collections library is very functional in design. Let's look at these common Java libraries and see how Scala's blend of functional programming with object orientation can enhance these Application Program Interfaces (APIs).

1.1.1 *Discovering existing functional concepts*

Many modern API designs have been incorporating functional ideas without ascribing them to functional programming. For Java, things such as Google Collections or the Spring Application Framework make popular functional concepts accessible to the Java developer. Scala takes this further and embeds them into the language. To illustrate, you'll do a simple translation of the methods on the popular Spring `JdbcTemplate` class and see what it starts to look like in Scala.

```
public interface JdbcTemplate {
    List query(PreparedStatementCreator psc,           ← Query for list of objects
              RowMapper rowMapper)
    ...
}
```

Now for a simple translation into Scala, you'll convert the interface into a trait having the same method(s):

```
trait JdbcTemplate {
    def query(psc: PreparedStatementCreator,
             rowMapper: RowMapper): List[_]
}
```

The simple translation makes a lot of sense but it's still designed with a distinct Java flair. Let's start digging deeper into this design. Specifically, let's look at the `PreparedStatementCreator` and the `RowMapper` interfaces.

```
public interface PreparedStatementCreator {
    PreparedStatement createPreparedStatement(Connection con)
        throws SQLException;
}
```

The `PreparedStatementCreator` interface contains only one method: `createPreparedStatement`. This method takes a JDBC connection and returns a `PreparedStatement`. The `RowMapper` interface looks similar:

```
public interface RowMapper {
    Object mapRow(ResultSet rs, int rowNum)
        throws SQLException;
}
```

Scala provides first-class functions. This feature lets us change the `JdbcTemplate` query method so that it takes functions instead of interfaces. These functions should have the same signature as the sole method defined on the interface. In this case, the `PreparedStatementCreator` argument can be replaced by a function that takes a connection and returns a `PreparedStatement`. The `RowMapper` argument can be replaced by a function that takes a `ResultSet` and an integer and returns some type of object. The updated Scala version of the `JdbcTemplate` interface would look as follows:

```
trait JdbcTemplate {
    def query(psc: Connection => PreparedStatement,
             rowMapper: (ResultSet, Int) => AnyRef
             ): List[AnyRef]
}
```

← Use first-class functions

The query method is now more functional. It's using a technique known as the *loaner pattern*. This technique involves some controlling entity (the `JdbcTemplate`) creating a resource and delegating the use of it to another function. In this case, there are two functions and three resources. Also, as the name implies, `JdbcTemplate` is part of a template method in which pieces of the behavior were deferred for the user to implement. In pure object-orientation, this is usually done via inheritance. In a more functional approach, these behavioral pieces become arguments to the controlling function. This provides more flexibility by allowing mixing/matching arguments without having to continually use subclasses.

You may be wondering why you're using `AnyRef` for the second argument's return value. `AnyRef` is equivalent in Scala to `java.lang.Object`. Because Scala has supported generics, even when compiling for 1.4 JVMs, we should modify this interface further to remove the `AnyRef` and allow users to return specific types.

```
trait JdbcTemplate {
    def query[ResultItem](psc: Connection => PreparedStatement,
                        rowMapper: (ResultSet, Int) => ResultItem
                        ): List[ResultItem]
}
```

← Typed return list

With a few simple transformations, you’ve created an interface that works directly against functions. This is a more functional approach because Scala’s function traits allow composition. By the time you’re finished reading this book, you’ll be able to approach the design of this interface completely differently.

Functional programming also shines when used in a collections library. The Ruby and Python programming languages support some functional aspects directly in their standard library collections. For Java users, the Google Collections library bring practices from functional programming.

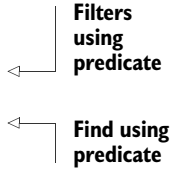
1.1.2 **Examining functional concepts in Google Collections**

The Google Collections API adds a lot of power to the standard Java collections. Primarily it brings a nice set of efficient immutable data structures, and some functional ways of interacting with your collections, primarily the `Function` interface and the `Predicate` interface. These interfaces are used primarily from the `Iterables` and `Iterators` classes. Let’s look at the `Predicate` interface and its uses.

```
interface Predicate<T> {
    public boolean apply(T input);
    public boolean equals(Object other);
}
```

The `Predicate` interface is simple. Besides equality, it contains an `apply` method that returns true or false against its argument. This is used in an `Iterators/Iterables-filter` method. The `filter` method takes a collection and a predicate. It returns a new collection containing only elements that pass the predicate `apply` method. Predicates are also used in the `find` method. The `find` method looks in a collection for the first element passing a `Predicate` and returns it. The `filter` and `find` method signatures are shown in the following code.

```
class Iterables {
    public static <T> Iterable<T> filter(Iterable<T> unfiltered,
        Predicate<? super T> predicate) {...}
    public static <T> T find(Iterable<T> iterable,
        Predicate<? super T> predicate) {...}
    ...
}
```



There also exists a `Predicates` class that contains static methods for combining predicates (ANDs/ORs) and standard predicates for use, such as “not null.” This simple interface creates some powerful functionality through the potential combinations that can be achieved with terse code. Also, because the predicate itself is passed into the `filter` function, the function can determine the best way or time to execute the filter. The data structure may be amenable to lazily evaluating the predicate, making the iterable returned a “view” of the original collection. It might also determine that it could best optimize the creation of the new iterable through some form of parallelism. This has been abstracted away, so the library could improve over time with no code changes on our part.

The Predicate interface is rather interesting, because it looks like a simple function. This function takes some type `T` and returns a `Boolean`. In Scala this would be represented `T => Boolean`. Let's rewrite the `filter`/`find` methods in Scala and see what their signatures would look like:

```
object Iterables {
  def filter[T](unfiltered: Iterable[T],
               predicate: T => Boolean): Iterable[T] = {...}
  def find[T](iterable: Iterable[T],
              predicate: T => Boolean): T = {...}
  ...
}
```

← No need
for ?

You'll immediately notice that in Scala we aren't using any explicit `? super T` type annotations. This is because Scala defines type variance at declaration time. For this example, that means that the variance annotation is defined on the `Function1` class rather than requiring it on every method that used the class.

What about combining predicates in Scala? We can accomplish a few of these quickly using some functional composition. Let's make a new `Predicates` module in Scala that takes in function predicates and provides commonly used function predicates. The input type of these combination functions should be `T => Boolean` and the output should also be `T => Boolean`. The predefined predicates should also have a type `T => Boolean`.

```
object Predicates {
  def or[T](f1: T => Boolean, f2: T => Boolean) =
    (t: T) => f1(t) || f2(t)
  def and[T](f1: T => Boolean, f2: T => Boolean) =
    (t: T) => f1(t) && f2(t)
  val notNull[T]: T => Boolean = _ != null
}
```

← Explicit
anonymous
function

← Placeholder
function syntax

We've now started to delve into the realm of functional programming. We're defining first-class functions and combining them to perform new behaviors. You'll notice the `or` method take two predicates, `f1` and `f2`. It then creates a new anonymous function that takes an argument `t` and ORs the results of `f1` and `f2`. Playing with functions also makes more extensive use of generics and the type system. Scala has put forth a lot of effort to reduce the overhead for generics in daily usage.

Functional programming is more than combining functions with other functions. The essence of functional programming is delaying side effects as long as possible. This predicate object defines a simple mechanism to combine predicates. The predicate isn't used to cause side effects until passed to the `Iterables` object. This distinction is important. Complex predicates can be built from simple predicates using the helper methods defined on the object `predicates`.

Functional programming grants the means to defer state manipulation in a program until a later time. It provides a mechanism to construct verbs that delay side effects. These verbs can be combined in a fashion that makes reasoning through a program simpler. Eventually the verbs are applied against the nouns of the system. In

traditional FP, side effects are delayed as long as possible. In blended OO-FP, the idioms merge.

1.2 **Static typing and expressiveness**

The Scala type system allows expressive code. A common misconception among developers is that static typing leads to verbose code. This myth exists because many of the languages derived from C, where types must be explicitly specified in many different places. As software has improved, along with compiler theory, this is no longer true. Scala uses some of these advances to reduce boilerplate in code and keep things concise.

Scala made a few simple design decisions that help make it expressive:

- Changing sides of type annotation
- Type inference
- Scalable syntax
- User-defined implicits

Let's look at how Scala changes the sides of type annotations.

1.2.1 **Changing sides**

Scala places type annotations on the right-hand side of variables. In some statically typed languages, like Java or C++, it's common to have to express the types of variables, return values, and arguments. When specifying variables or parameters, the convention, drawn from C, is to place type indicators on the left-hand side of the variable name. For method arguments and return values, this is acceptable, but causes some confusion when creating different styles of variables. C++ is the best example of this, as it has a rich set of variable styles, such as `volatile`, `const`, pointers, and references. Table 1.2 shows a comparison of C++ variables and Scala variables.

The more complicated a variable type, the more annotations are required directly on the type of the variable. In C++, this is maximized in the usage of a pointer, because a pointer can be constant. Scala defines three variable types on the left-hand side, like `var`, `val`, and `lazy val`. These leave the type of the variable clean. In all instances, the type of the name `x` is `Int`.

Table 1.2 Variable definition in C++ versus Scala

Variable type	C++	Java	Scala
Mutable integer variable	<code>int x</code>	<code>int x</code>	<code>var x: Int</code>
Immutable integer value	<code>const int x</code>	<code>final int x</code>	<code>val x: Int</code>
Constant pointer to a volatile integer	<code>volatile int * const x</code>	N/A	N/A
Lazily evaluated integer value	N/A	N/A	<code>lazy val x: Int</code>

In addition to separating the concerns of how a variable behaves from the variable type, the placement of types on the right allows type inference to determine the type of the variables.

1.2.2 Type inference

Scala performs type inference wherever possible. Type inference is when the compiler determines what the type annotation should be, rather than forcing the user to specify one. The user can always provide a type annotation, but has the option to let the compiler do the work.

```
val x: Int = 5
val y = 5
```

This feature can drastically reduce the clutter found in some other typed languages. Scala takes this even further to do some level of inference on arguments passed into methods, specifically with first-class functions.

```
def myMethod(functionLiteral: A => B): Unit
myMethod({ arg: A => new B })
myMethod({ arg => new B })
```

If a method is known to take a function argument, the compiler can infer the types used in a function are literal.

1.2.3 Dropping verbose syntax

Scala syntax takes the general approach that when the meaning of a line of code is straightforward, the verbose syntax can be dropped. This feature can confuse users first using Scala but can be rather powerful when used wisely. Let's show a simple refactoring from the full glory of Scala syntax into the simplistic code that's seen in idiomatic usage. Here is a function for Quicksort in Scala.

```
def qsort[T <% Ordered[T]](list:List[T]):List[T] = { ← <% means "view"
  list.match({
    case Nil => Nil;
    case x::xs =>
      val (before,after) = xs.partition({ i => i.<(x) });
      qsort(before).++(qsort(after).::(x)); ← ++ and ::
  });                                       mean aggregate
}
```

This code accepts a list whose type, `T`, is able to be implicitly converted into a variable of type `Ordered[T]` (`T <% Ordered[T]`). We'll discuss type parameters and constraints in great detail in chapter 6, so don't focus too much on these. We're requiring that the list contain elements that we have some notion of ordering for, specifically a less than function (`<`). We then examine the list. If it's empty, or `Nil`, then we return a `Nil` list. If it encounters a list, we extract the head (`x`) and tail (`xs`) of the list. We use the head element of the list to partition the tail into two lists. We then recursively call the Quicksort method on each partition. In the same line, we combine the sorted partitions and the head element into a complete list.

You may be thinking, “Wow, Scala looks ugly.” In this case you would be right. The code is cluttered and difficult to read. There’s a lot of syntactic noise preventing the meaning of the code from being clear. There’s also a lot of type information after `qsort`. Let’s pull out our surgical knife and start cutting out cruft. First we’ll start with Scala’s semicolon inference. The compiler will assume that the end of a line is the end of an expression, unless you leave some piece of syntax hanging, like the `.` before a method call.

But removing semicolons isn’t quite enough to reduce the clutter. We should also use an *operator notation*. This is the name Scala gives to its ability to treat methods as operators. A method of no arguments can be treated as a postfix operator. A method of one argument can be treated as an infix operator. There’s also the special rule for certain characters (for example, `:`) at the end of a method name that reverses the order of a method call. These rules are demonstrated as follows:

```
x.foo(); /*is the same as*/ x foo
x.foo(y); /*is the same as*/ x foo y
x.:(y); /*is the same as*/ y :: x
```

Scala also provides placeholder notation when defining anonymous functions (aka, lambdas). This syntax uses the `_` keyword as a placeholder for a function argument. If more than one placeholder is used, each consecutive placeholder refers to consecutive arguments to the function literal. This notation is usually reserved for simple functions, such as the less-than (`<`) comparison in our Quicksort.

We can apply this notation paired with operator notation to achieve the following on our quick sort algorithm:

```
def qsort[T <% Ordered[T]](list:List[T]):List[T] = list match {
  case Nil => Nil
  case x :: xs =>
    val (before, after) = xs partition ( _ < x )
    qsort(before) ++ ( x :: qsort(after));
}
```

Placeholder notation used instead of =>

Scala offers syntactic shortcuts for simple cases, and it provides a mechanism to bend the type system via implicit conversions and implicit arguments.

1.2.4 *Implicits are an old concept*

Scala implicits are a new take on an old concept. The first time I was ever introduced to the concept of implicit conversions was with primitive types in C++. C++ allows primitive types to be automatically converted as long as there is no loss of precision. For example, we can use an `int` literal when declaring a `long` value. The types *double*, *float*, *int*, and *long* are different to the compiler. It does try to be intelligent and “do the right thing” when mixing these values. Scala provides this same mechanism, but using a language feature that’s available for anyone.

The `scala.Predef` object is automatically imported into scope by Scala. This places its members available to all programs. It’s a handy mechanism for providing convenience functions to users, like directly writing `println` instead of `Console`

.println or System.out.println. Predef also provides what it calls *primitive widenings*. These are a set of implicit conversions that automatically migrate from lower-precision types to higher precision types. The following listing shows the set of methods defined for the Byte type.

Listing 1.1 Byte conversions in scala.Predef object

```
implicit def byte2short(x: Byte): Short = x.toShort
implicit def byte2int(x: Byte): Int = x.toInt
implicit def byte2long(x: Byte): Long = x.toLong
implicit def byte2float(x: Byte): Float = x.toFloat
implicit def byte2double(x: Byte): Double = x.toDouble
```

These methods are calls to the runtime-conversion methods. The implicit before the method means the compiler may attempt to apply this method to a type Byte, if it's required for correct compilation. This means if we attempt to pass a Byte to a method requiring a Short, it will use the implicit conversion defined as byte2short. Scala also takes this one step further and looks for methods via implicit conversions if the current type doesn't have the called method. This comes in handy for more than just primitive conversions.

Scala also uses the implicit conversion mechanism as a means of extending Java's base classes (Integer, String, Double, and so on). This allows Scala to make direct use of Java classes, for ease of integration, and provide richer methods that make use of Scala's more advanced features. Implicits are a powerful feature and are mistrusted by some. The key to implicits in Scala are knowing how and when to use them.

1.2.5 Using Scala's implicit keyword

Utilizing implicits is key to manipulating Scala's type system. They're primarily used to automatically convert from one type to another as needed, but can also be used to limited forms of compiler time metaprogramming. To use, implicits must be associated with a lexical scope. This can be done via companion objects or by explicitly importing them.

The implicit keyword is used in two different ways in Scala. First it's used to identify and create arguments that are automatically passed when found in the scope. This can be used to lexically scope certain features of an API. As implicits also have a lookup policy, the inheritance linearization, they can be used to change the return type of methods. This allows some advanced APIs and type-system tricks such as that used in the Scala collections API. These techniques are covered in detail in chapter 7.

The implicit keyword can also be used to convert from one type to another. This occurs in two places, the first when passing a parameter to a function. If Scala detects that a different type is needed, it will check the type hierarchy and then look for an implicit conversion to apply to the parameter. An implicit conversion is a method, marked implicit, that takes one argument and returns something. The second place where Scala will perform an implicit conversion is when a method is called against a particular type. If the compiler can't find the desired method, it will apply implicit

conversations against the variable until it either finds one that contains the method or it runs out of conversions. This is used in Scala’s “pimp my library” pattern, described in chapter 7.

These features combine an expressive syntax with Scala, despite its advanced type system. Creating expressive libraries requires a deep understanding of the type system, as well as thorough knowledge of implicit conversions. The type system will be covered more fully in chapter 6. The type system also interoperates well with Java, which is a critical design for Scala.

1.3 **Transparently working with the JVM**

One of Scala’s draws is its seamless integration with Java and the JVM. Scala provides a rich compatibility with Java, such that Java classes can be mapped directly to Scala classes. The tightness of this interaction makes migrating from Java to Scala rather simple, but caution should be used with some of Scala’s advanced feature sets. Scala has some advanced features not available in Java, and care was taken in the design so that seamless Java interaction can be achieved. For the most part, libraries written in Java can be imported into Scala as is.

1.3.1 **Java in Scala**

Using Java libraries from Scala is seamless because Java idioms map directly into Scala idioms. Java classes become Scala classes; Java interfaces become abstract Scala traits. Java static members get added to a pseudo-Scala object. This combined with Scala’s package import mechanism and method access make Java libraries feel like natural Scala libraries, albeit with more simplistic designs. In general, this kind of interaction just works. For example, the following listing shows a Java class that has a constructor, a method, and a static helper method.

Listing 1.2 Simple Java object

```
class SimpleJavaClass {
  private String name;
  public SimpleJavaClass(String name) {           ← Constructor
    this.name = name;
  }
  public String getName() {                       ← Class method
    return name;
  }
  public static SimpleJavaClass create(String name) { ← Static class helper
    return new SimpleJavaClass(name);
  }
}
```

Now, let’s use this in Scala.

```
val x = SimpleJavaClass.create("Test")           ← Calling Java static methods
x.getName()                                       ← Calling Java methods
val y = new SimpleJavaClass("Test")             ← Using Java constructor
```

This mapping is rather natural and makes using Java libraries a seamless part of using Scala. Even with the tight integration, Java libraries usually have a form of thin Scala wrapper that provides some of the more advanced features a Java API could not provide. These features are apparent when trying to use Scala libraries inside Java.

1.3.2 Scala in Java

Scala attempts to map its features to Java in the simplest possible fashion. For the most part, simple Scala features map almost one-to-one with Java features (for example, classes, abstract classes, methods). Scala has some rather advanced features that don't map easily into Java. These include things like objects, first-class functions, and implicits.

SCALA OBJECTS IN JAVA

Although Java statics map to Scala objects, Scala objects are instances of a singleton class. This class name is compiled as the name of the object with a \$ appended to the end. A `MODULE$` static field on this class is designed to be the sole instance. All methods and fields can be accessed via this `MODULE$` instance. Scala also provides *forwarding* static methods when it can; these exist on the companion class (that is, a class with the same name as the object). Although the static methods are unused in Scala, they provide a convenient syntax when called from Java.

```
object ScalaUtils {
  def log(msg : String) : Unit = Console.println(msg)    ← Simple Scala method
  val MAX_LOG_SIZE = 1056                               ← Simple Scala field
}
ScalaUtils.log("Hello!");                               ← Acts like static call
ScalaUtils$.MODULE$.log("Hello!");                     ← Use the singleton instance
System.out.println(ScalaUtils$.MODULE$.MAX_LOG_SIZE()); ← Variables become
System.out.println(ScalaUtils.MAX_LOG_SIZE());         ← Static forwarder
```

SCALA FUNCTIONS IN JAVA

Scala promotes the use of function as object, or first-class functions. As of Java 1.6, there is no such concept in the Java language (or the JVM). Therefore, Scala creates the notion of *function traits*. These are a set of 23 traits that represent functions of arity 0 through 22. When the compiler encounters the need for passing a method as a function object, it creates an anonymous subclass of an appropriate function trait. As traits don't map into Java, the passing of first-class functions from Java into Scala is also inhibited but not impossible.

```
object FunctionUtil {
  def testFunction(f : Int => Int) : Int = f(5)
}
abstract class AbstractFunctionIntIntForJava extends
  (Int => Int) {
}
Special abstract class to use from Java
```

We’ve created an abstract class in Scala that Java can implement more easily than a function trait. Although this eases the implementation in Java, it doesn’t make things 100% simple. There’s still a mismatch between Java’s type system and Scala’s encoding of types that requires us to coerce the type of the function when making the Scala call, as you can see in the following listing.

Listing 1.3 Implementing a first-class function in Java

```
class JavaFunction {
    public static void main(String[] args) {
        System.out.println(FunctionUtil.testFunction(
            (scala.Function1<Integer, Integer>)
                new AbstractFunctionIntIntForJava() {
                    public Integer apply(Integer argument) {
                        return argument + 5;
                    }
                }
        ));
    }
}
```

← Coerce types
 ← First-class function
 ← Function logic

It’s possible to use first-class functions and with them a more functional approach when combining Scala and Java. But other alternatives exist to make this work. A more detailed discussion of this tweak, along with other Java–Scala related issues can be found in chapter 10. As you can see, Scala can integrate well with existing Java programs and be used side by side with existing Java code. Java–Scala interaction isn’t the only benefit of having Scala run inside the JVM; the JVM itself provides a huge benefit.

1.3.3 The benefits of a JVM

As alluded to earlier, the JVM provides many of the benefits associated with Java. Through bytecode, libraries become distributable to many differing platforms on an as is basis. The JVM has also been well tested in many environments and is used for large-scale enterprise deployments. It has also been a big focus on performance of the Java platform. The HotSpot compiler can perform various optimizations on code at runtime. This also enables users to upgrade their JVM and immediately see performance improvements, without patches or recompiling.

HOTSPOTING

The primary benefit of Scala running on the JVM is the HotSpot runtime optimizer. This allows runtime profiling of programs, with automatic optimizations applied against the JVM bytecode. Scala acquires these optimization “for free” by nature of running against the JVM. Every release of the JVM improves the HotSpot compiler, and this improves the performance of Scala. The HotSpot compiler does this through various techniques. Including the following:

- Method inlining
- On Stack Replacement (OSR)
- Escape Analysis
- Dynamic deoptimization

Method inlining is HotSpot's ability to determine when it can inline a small method directly at a call-spot. This was a favorite technique of mine in C++, and HotSpot will dynamically determine when this is optimal. *On Stack Replacement* refers to HotSpot's ability to determine that a variable could be allocated on the stack versus the heap. I remember in C++ the big question when declaring a variable was whether to place it on the stack or the heap. Now HotSpot can answer that for me. HotSpot performs *escape analysis* to determine if various things "escape" a certain scope. This is primarily used to reduce locking overhead when synchronized method calls are limited to some scope, but it can be applied to other situations. *Dynamic deoptimization* is the key feature of HotSpot. It's the ability to determine whether an optimization did *not* improve performance and undo that optimization, allowing others to be applied. These features combine into a pretty compelling picture of why new and old languages (for example, Ruby) desire to run on the JVM.

1.4 Summary

In this chapter, you've learned a bit about the philosophy of Scala. Scala was designed with the idea of blending various concepts from other languages. Scala blends functional and object-oriented programming, although this has been done in Java as well. Scala made choices about syntax that drastically reduced the verbosity of the language and enabled some powerful features to be elegantly expressed, such as type inference. Finally, Scala has tight integration with Java and runs on top of the Java virtual machine, which is perhaps the single most important aspect to make Scala relevant to us. It can be utilized in our day-to-day jobs with little cost.

As Scala blends various concepts, users of Scala will find themselves striking a balance among functional programming techniques, object orientation, integration with existing Java applications, expressive library APIs, and enforcing requirements through the type system. Often the best course of action is determined by the requirements at hand. It's the intersection of competing ideas where Scala thrives and also where the greatest care must be taken. This book will help guide when to use each of these techniques.

Let's start looking at a few key concepts every Scala developer needs to know when coding Scala.