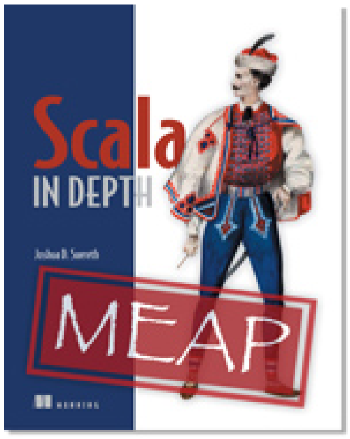
MANNING PUBLICATIONS



Scala—a Blended Language

Scala is a language that blends various techniques seen from other programming languages. Scala attempts to leap between worlds by offering its developers object-oriented programming features, functional programming features, a very expressive syntax, statically enforced typing, and rich generics—all on top of the Java Virtual Machine. Scala allows developers to focus on familiar features, but its true power comes from forming a balance amongst the sometimes competing ideas and features. The truly freeing power in Scala is the ability to dive into whatever paradigm is most useful for the task at hand. If an imperative design is best suited for the task, then there is nothing preventing the developer from writing imperative code. If a functional approach with immutability is desired, then the programmer is able to make it happen. Even more importantly, when the problem domain has different requirements, more than one approach can be used for various pieces of the solution.

The philosophy of Scala

To understand Scala, we need to understand the environment that produced Scala: The Java Ecosystem. The Java™ Language entered the Computer Science realm around 1995, making a huge splash. Java and the Virtual Machine that it runs on began to slowly revolutionize programming. At the time, C++ was coming into the spotlight. Developers were moving from purely C-style programs and learning how to effectively use object orientation. Despite all the benefits that C++ offered, it still suffered a few pain points, such as difficulties in distributing libraries and complexity in its object orientation. Java solved both these points through a limited form of object orientation and its use of a Virtual Machine. The Java Virtual Machine (JVM) allowed code to be written and compiled on a given platform, then distributed to other platforms with very little effort. Although cross platform problems did not disappear, the cost of cross platform programming was greatly reduced.

Over time, the JVM got better and better at efficient execution while the Java Community grew and grew. The HotSpot™ optimizer was invented, which allowed code to be profiled before optimization occurred. While slower in startup, the eventual runtime performance became ideal for applications such as servers. Although not initially designed for it, the JVM began to prevail in enterprise server application environments. With this came attempts to make Enterprise Server development easy. Things like Enterprise Java Beans™ and more recently the Spring Application Framework™ emerged to help developers utilize the power of the JVM. The community around Java exploded and millions of easily consumed libraries were created. "If you can think of it, there's probably a Java library for it" became a workplace slogan. The Java language continued to evolve slowly, trying to maintain its community.

Meanwhile, a select few developers began stretching their wings and ran into the limitations of Java's design. Java was meant to simplify things, and some members in the community needed increased complexity in a controlled fashion. The second wave of JVM languages occurred and it's continuing to grow. Languages such as Groovy, JRuby, Clojure, and Scala herald a new era for the Java programmer. No longer are we limited to just one language, but we have a multitude of options, each with various advantages and disadvantages. One of the more popular languages is Scala.

Scala was born from the mind of Martin Odersky, a man that 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. This blending of concepts was hoped to create something that real developers would find useful and that could be studied for new programming idioms. It was a large success; so much that the industry has started adopting Scala as a viable and competitive language. Understanding Scala requires understanding a mixture of concepts. Scala attempts to blend three dichotomies of thought into one language:

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

Let's take a look at how these features blend starting with functional and object-oriented programming.

Functional programming meets object orientation

Functional programming and object orientation are two different approaches to software development. Functional programming is not a new concept and is not foreign to a modern developer's toolbox. We'll show this through examples from the Java Ecosystem, specifically the Spring Application framework and the Google Collections library. Each of these libraries has functional concepts blended with object orientation; however, each can be more elegant when translated into Scala. Before proceeding much further, it's important to understand the terms object oriented programming and functional programming (see table 1).

Object-oriented programming is a top-down approach to code design. Object-oriented programming 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 do not have a notion of state or behavior independently of the objects they operate on.

Functional programming approaches software as the combination and application of various functions. Functional programming 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. The immutability encouraged by functional programming aids concurrent programming. Functional programming works very well for composing behaviors but can be awkward when used with things that are expected to change, such as I/O.

Table 1 Attributes commonly ascribed to object-orientated and functional programming

Object-oriented programming	Functional programming
Composition of objects (nouns)	Composition of functions (verbs)
Encapsulated stateful interaction	Stateless interaction
Iterative algorithms	Recursive algorithms
Imperative flow	Lazy evaluation

N/A pattern Matching

Functional programming and object orientation offer unique views of software. It's these differences that make them very useful to each other. Object orientation can deal with the *nouns* and functional programming can deal with the *verbs*. In fact, many Java developers have started moving toward this strategy in recent years. The Enterprise Java Beans (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 (without being 100 percent immutable/pure).

This push of functional-style libraries has come along much farther than the EJB specifications. The Spring Application Framework promotes a very functional style with its Template classes, and the Google Collections library is very functional in design. Let's take a look at these common Java libraries and see how Scala's blend of functional and object orientation can enhance these 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 a bit further and embeds them into the language. Let's do a simple translation of the methods on the popular Spring JdbcTemplate class and see what it starts to look like in Scala (listing 1).

Listing 1 A query method on Spring's JdbcTemplate class

```
public interface JdbcTemplate {
  List query(PreparedStatementCreator psc,  #A
             RowMapper rowMapper)
  ...
}
```

#A Query for List of objects

Now for a simple translation into Scala, We'll convert the interface into a trait having the same method (listing 2).

Listing 2 A simple Scala translation of the query method

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

The simple translation makes a lot of sense; however, 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 (listing 3).

Listing 3 PreparedStatementCreator interface

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

#A The sole method defined

The PreparedStatementCreator interface contains only one method. This method takes a JDBC Connection and returns a PreparedStatement. The RowMapper interface looks similar (listing 4).

Listing 4 RowMapper interface

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

#A The sole method defined

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, `PreparedStatementCollector` 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 in listing 5.

Listing 5 Initial Scala version of Spring's JdbcTemplate class

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

#A 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 where pieces of the behavior were deferred to 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 we're using `AnyRef` for the second argument's return value. AnyRef is equivalent in Scala to `java.lang.Object`. Since 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 (listing 6).

Listing 6 Typed version of Spring's JdbcTemplate class

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

#A Typed return list

With a few simple transformations, we've created a more functional interface.

Although not as pure as most functional languages, we're well on our way toward a functional looking API. Let's continue looking for examples in the Java Ecosystem of functional design. In particular, let's look at the Google Collections API.

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 take a look at the `Predicate` interface and its uses (listing 7).

Listing 7 Google Collections `Predicate` interface

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

#A matches Function1.apply

The `Predicate` interface is rather simple. Besides equality, it contains an `apply` method, which returns true or false against its argument. This is used in `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 listing 8.

Listing 8 Iterables filter and find methods

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

#A Filters using predicate
#B Find using predicate

There is also a `Predicates` class that contains static methods for the combining predicates (ands/ors) as well as the standard predicates for uses such as `not null`. This simple interface creates some powerful functionality through the potential combinations that can be achieved with very terse code. Also, since 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, thus making the iterable return 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. The fact is that this has been abstracted away, so the library could improve over time with no code changes on our part.

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

Listing 9 Iterables filter and find methods in Scala

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

#A No need for ?

You'll immediately notice that, in Scala, we aren't using any explicit `? super T` type annotations. This is because the Function interface in Scala is appropriately annotated with covariance and contravariance. *Covariance* (+T or `? extends T`) is when a type can be coerced down the inheritance hierarchy. *Contravariance* (-T or `? super T`) is when a type can be coerced up the inheritance hierarchy. *Invariance* is when a type cannot be coerced at all. In this case, a Predicate's argument can be coerced up the inheritance hierarchy as needed. This means, for example, that a predicate against mammals could apply to a collections of cats, assuming a cat is a subclass of mammal. In Scala, you specify co/contra/in-variance at class definition time.

What about combining predicates in Scala? We can accomplish a few of these rather 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 (listing 10). 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`.

Listing 10 Predicates in Scala

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

#A Explicit anonymous function
#B Placeholder } function syntax

We've now started to delve into the realm of functional programming. We are defining first-class functions and combining them to perform new behaviors. You'll notice the `or` method takes two predicates, f1 and `f2`. It then creates a new anonymous function that takes an argument `t` and `or`s the results of `f1` and `f2`. This is the essence of functional programming. 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. Let's look into how Scala's type system allows expressive code.

Static typing and expressiveness

There is a common misconception among developers that static typing leads to verbose code. This myth exists because many languages had derived from C, where types must be explicitly specified in many different places. Since the software and the compiling techniques have improved, 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 which help made it very expressive, these being:

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

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

1.1.3 Changing sides

Scala places type annotations on the right-hand side of variables. In some statically typed languages, like Java or C++, it is 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 because it has a rich set of variable styles, such as volatile, const, pointers, and references (listing 11).

Listing 11 Examples of integer variables in C++

```
int x  #A
const int x  #B
int & x  #C
const int & const x  #D
```

#A Mutable integer variable
#B Immutable integer variable
#C Reference to integer value
#D Immutable reference to immutable integer value

This mingling of the style of variable with the type annotation leads to some rather complex definitions. Scala, like a few other languages, places its type annotations on the right of variables (listing 12). This separates the style of variable from the type annotation, helping to reduce some complexity when reading values.

Listing 12 Examples of variable styles in Scala

```
var x : Int  #A
val x : Int  #B
lazy val x : Int  #C
```

#A Mutable integer variable
#B Immutable integer variable

#C Immutable integer variable with deferred execution

This demonstrates the simplicity achieved merely by migrating type annotations to the right of the variable names; however, things still aren't as simple as they could be. Scala further reduces syntactic clutter by attempting to infer the missing type annotations.

1.1.4 Type inference

Scala helps to further reduce syntactic noise by performing 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 (see listing 13).

Listing 13 Type inference in Scala

```
val x : Int = 5  #A
val y = 5  #B
```

#A User specified type
#B Inferred type

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. If a method is known to take a function argument, the compiler can infer the types used in a function literal (listing 14).

Listing 14 Functional literal type inference

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

#A Explicit type
#B Type inferred

1.1.5 Dropping the 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 walking into 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 is seen in idiomatic usage. Listing 15 shows the function for quicksort in Scala.

Listing 15 Verbose Scala quicksort

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

#A <% means 'view'
#B ++ and :: mean aggregate

This code accepts a list whose type, `T`, is able to be implicitly converted into a variable of type `Ordered[T]` (in other words `T <% Ordered[T]`). We're requiring that the list contain elements for which we have some notion of ordering, specifically a less than function (`<`). We then examine the list. If is empty or Nil, then we return a Nil list. If it encounters a list, we extract the head (`x`) and the 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 rather cluttered and difficult to read. There's quite a bit of syntactic noise preventing the meaning of the code from being clear. Not only that, but there's an awful lot of type information after `qsort`. Let's pull out our surgical knife and start cutting out cruft. First, let's 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.

Removing semicolons isn't quite enough to reduce the clutter, however. We should also use *operator notation*. Operator notation 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 also are special rules for certain characters; for example, `:` at the end of a method name reverses the order of a method call. These rules are demonstrated in listing 16.

Listing 16 Operator notation

```
x.foo(); /*is the same as*/ x foo  #A
x.foo(y); /*is the same as*/ x foo y  #B
x.::(y); /*is the same as*/ y :: x  #C
```

#A Postfix notation
#B Infix notation
#C Inverted infix notation

Scala also provides placeholder notation when defining anonymous functions (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 very simple functions, such as the less-than comparison in our quicksort.

We can pair this notation with an operator notation to achieve the result shown in listing 17 on our quicksort algorithm.

Listing 17 Less verbose Scala quicksort

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

#A Placeholder notation used instead of =>

Scala not only offers syntactic shortcuts for simple cases, it also provides a mechanism to bend the type system via implicits conversions and implicit arguments.

1.1.6 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. As an example, I can use an int literal when declaring a long value. The actual 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 is 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. Let's take a look at the set of methods defined for the Byte type (listing 18).

Listing 18 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 simply 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 is 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 does not 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 very powerful feature and are mistrusted by some. The key to implicits in Scala are knowing how and when to use them.

1.1.7 Using Scala's implicit keyword

Utilizing implicits is key to manipulating Scala's type system. They are primarily used to automatically convert from one type to another as needed, but can also be used for limited forms of compiler time metaprogramming. Implicits must be associated with a lexical scope to be used. This can be done either via companion objects or 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 linearizion—they can be used to change the return type of methods. This allows some very advanced APIs and type-system tricker such as that used in the Scala collections API.

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, after checking the type hierarchy, it will look for an implicit conversion to apply to the parameter. An implicit conversion is a simple 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 cannot 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.

These features combine to Scala a very expressive syntax 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 also interoperates very well with Java, which is a critical design for Scala.

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; however, caution needs to be used with some of Scala's advanced feature set. Scala has some rather 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.

1.1.8 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, let's take a Java class that has a constructor, a method and a static helper method (listing 19).

Listing 19 Simple Java Object

```
class SimpleJavaClass {
  private String name;
  public SimpleJavaClass(String name) {  #A
    this.name = name;
  }
  public String getName() {   #B
```

```
    return name;
  }
  public static SimpleJavaClass create(String name) {   #C
    return new SimpleJavaClass(name);
  }
}
```
#A Constructor
#B Class method
#C Static class helper

Now, let’s use this in Scala (listing 20).

Listing 20 Simple Java Object used in Scala

```
val x = SimpleJavaClass.create("Test") #A
x.getName()  #B
val y = new SimpleJavaClass("Test")  #C
```
#A Calling Java static methods
#B Calling Java methods
#C 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 some form of thin Scala wrapper that provides some of the more advanced features a Java API could not provide. These features are very apparent when trying to make use of Scala libraries inside Java.

1.1.9 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 do not 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 really instances of a singleton class. This class name is compiled as the name of the object with a $ appended to the end. There is a MODULE$ static field on this class, which is designed to be the sole instance. All methods/fields can be accessed via this MODULE$ instance. Scala also provides forwarding static methods when it can. These static methods exist on the companion class (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 (see listings 21 and 22).

Listing 21 Simple Scala object

```
object ScalaUtils {
  def log(msg : String) : Unit = Console.println(msg)  #A

  val MAX_LOG_SIZE = 1056  #B
}
```
#A Simple Scala method
#B Simple Scala field

Listing 22 Using simple Scala object in Java

```
ScalaUtils.log("Hello!");  #A
ScalaUtils$.MODULE$.log("Hello!");  #B
System.out.println(ScalaUtils$.MODULE$.MAX_LOG_SIZE());  #C
System.out.println(ScalaUtils.MAX_LOG_SIZE());  #D
```
#A Acts like static call
#B Use the singleton instance
#C Variables become *methods*
#D 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 itself). Therefore, Scala creates the notion of Function traits. These are a set of 23 traits that represent functions of arities 0 through 23. 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 do not map into Java, the passing of first-class functions from Java into Scala is also inhibited but not impossible (see listing 22).

Listing 22 Using simple Scala object in Java

```
ScalaUtils.log("Hello!");  #A
ScalaUtils$.MODULE$.log("Hello!");  #B
System.out.println(ScalaUtils$.MODULE$.MAX_LOG_SIZE());  #C
System.out.println(ScalaUtils.MAX_LOG_SIZE()); #D
```

#A Acts like static call
#B Use the singleton instance
#C Variables become *methods*
#D Static forwarder

We've created an abstract class in Scala that Java can implement more easily then a function trait. Although this eases the implementation in Java, it doesn't make things 100 percent 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.

Listing 24 Implementing a first-class function in Java

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

#A Coerce the types
#B First-class function
#C Function logic

So, it is possible to use first-class functions, and with it a more functional approach when combining Scala and Java. However, other alternatives exist to make this work. As you can see, Scala can integrate quite 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.

1.1.10 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. Not only is it well tested, there has 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.

HOTSPOT-ING

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 optimizations 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:

- 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++. 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 vs. 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. Escape analysis is performed by HotSpot 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; however, it can be applied to other situations. Dynamic deoptimization is the key feature of HotSpot. It is the ability to determine if an optimization did *not*, in fact, improve performance and undo that optimization, allowing others to be applied. These features combine into a pretty compelling picture of why new/old languages (for example, Ruby) desire to run on the JVM.

Summary

You've learned a little bit about the philosophy of Scala. Scala was designed with the idea of blending together various concepts from other languages. Scala blends together functional and object-oriented programming, although clearly 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 very tight integration with Java and runs on top of the Java Virtual Machine, which is perhaps the single most important aspect that makes Scala relevant. It can be utilized in our day-to-day jobs with little cost.

As Scala blends together various concepts, users of Scala will find themselves striking a delicate balance between 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 is the intersection of competing ideas where Scala thrives and also where the greatest care must be taken.

Last updated: October 11, 2011