

The busy Java developer's guide to Scala: Functional programming for the object oriented

Find out how Scala leverages the best of both worlds

Skill Level: Introductory

[Ted Neward \(ted@tedneward.com\)](mailto:ted@tedneward.com)
Principal
Neward & Associates

22 Jan 2008

The Java™ platform has historically been the province of object-oriented programming, but even Java language stalwarts are starting to pay attention to the latest old-is-new trend in application development: functional programming. In this new series, Ted Neward introduces Scala, a programming language that combines functional and object-oriented techniques for the JVM. Along the way, Ted makes the case for why you should take the time to learn Scala — concurrency, for one — and shows you how quickly it will pay off.

You never forget your first love.

For me, her name was Tabinda (Bindi) Khan. It was the halcyon years of my youth, seventh grade to be exact, and she was beautiful, smart, and, best of all, she laughed at my clumsy teenage boy jokes. We were "going out" (as it was called back then), on and off, for most of the seventh and eighth grades. But by ninth grade, we'd drifted apart, which is the polite way of saying that she got tired of hearing the same clumsy teenage boy jokes for two years running. I will never forget her (particularly because we ran into each other again at our 10-year high-school reunion); but more importantly, I will never lose those cherished — if somewhat exaggerated — memories.

About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this new developerWorks [series](#), you'll learn what all the recent hype is about and see some of Scala's linguistic capabilities in action. Scala code and Java code will be shown side by side wherever comparison is relevant, but (as you'll discover) many things in Scala have no direct correlation to anything you've found in Java programming, and therein lies much of Scala's charm!

After all, if Java code could do it, why bother learning Scala?

Java programming, and object-orientation, was the first love for many programmers, and we treat it with the same respect and outright adoration I gave to Bindi. Some developers will tell you that Java programming rescued them from the hellfire-and-brimstone pits of memory-management and C++. Others will say that Java programming elevated them out of the depths of procedural despair. There are even developers for whom object-oriented programming in Java code simply is "the way we've always done things." (And hey, if it worked for my daddy, and his daddy before him!)

Time ultimately overcomes all first loves, however, and there comes a time to move on. The feelings have changed and the actors in the story have matured (and hopefully learned a few new jokes). But more importantly, the world around us has changed. Many Java developers are realizing that much as we love Java programming, it's time to catch up to the new opportunities in our development landscape and see what we can make of them.

I will always love you ...

Within the last half-decade, a growing surge of discontent with the Java language has begun to emerge. While some may point to the growth of Ruby on Rails as the principal factor here, I'd argue that RoR (as it is known to the Ruby *cognoscenti*) is the effect, not the cause. Or, perhaps more accurately, that the Java developers taking up Ruby are the effects of a deeper, more insidious cause.

Put simply, Java programming is showing its age.

Or, to be more precise, the Java *language* is showing its age.

Consider this: When the Java language was first born, Clinton (the first) was in office and the Internet was still only being used by real geeks on a regular basis, largely because dialup was the only way to get it at home. Blogs hadn't yet been invented and everyone believed that inheritance was the fundamental approach to reuse. We also believed that objects were the best way to model the world and that Moore's Law would rule exponentially forever.

Actually, it's Moore's Law that has many in the industry particularly concerned. Since 2002/2003, the growing trend in microprocessors has been to create CPUs with multiple "cores": In essence, multiple CPUs within a single chip. This obviates Moore's Law, which says that CPU speeds will double every 18 months. Having multithreaded environments executing on two CPUs simultaneously, rather than doing the standard round-robin cycle on the single CPU, means that code must be rock-solidly thread-safe if it's to survive.

Much research has been going on in the academic community around this particular problem, resulting in a plethora of new languages. The critical flaw is the fact that most of these languages were built on top of their own virtual machine or interpreter,

so they represent (like Ruby does) a transition to a new platform. The concurrency crunch is a real concern, and some of the new languages offer powerful answers, but too many corporations and enterprises remember migrating from C++ to the Java platform just 10 years ago. To move to a new platform is a risk that many companies just aren't going to seriously consider. Many, in fact, are still nursing the scars from the last move to the Java platform.

Enter Scala.

A SCALable LAnguage

Why Scala?

Learning a new programming language is always a daunting task, particularly for one with an entirely new mindset on approaching problems, such as the functional approach that Scala embraces, and even more so when it is hybridized with another approach, such as how Scala melds object-orientation with functional concepts. Learning Scala requires time, and with that commitment looming over your already busy schedule, you may not be able to identify the return on your investment at first blush. Let me assure you, Scala provides a number of intriguing features, many of which will be covered in future pieces in this series. Following is a partial list to help you see the benefits of tackling this language. With Scala, you'll be able to:

- **Create *internal DSLs*** (think Ruby), thanks to Scala's flexibility regarding identifiers.
- **Create highly scalable, concurrent data processors**, thanks to Scala's immutable-state-by-default stance.
- **Reduce equivalent Java code** by one-half or two-thirds, thanks to a variety of Scala's syntactic features such as closures and implicit definitions.
- **Take advantage of parallel hardware architectures** (such as multi-core CPUs), thanks to Scala's encouragement of functional designs.
- **Understand larger code bases**, thanks to Scala's simplification of certain type rules, in essence demanding that "everything is an object."

Certainly Scala represents a powerful, new way to look at programming; the fact that it compiles to and runs on the JVM just makes using it for "real work" all that much easier.

Scala is a functional-object hybrid language with several powerful factors working in its favor:

- First, Scala compiles to Java bytecode, meaning it runs on the JVM. In addition to enabling you to continue leveraging the rich Java open-source ecosystem, Scala can be integrated into an existing IT environment with zero migration effort.

- Second, Scala is based on the functional principles of Haskell and ML, yet still borrows heavily from the familiar object-oriented concepts Java programmers love. As a result, it can blend the best of both worlds into a whole that offers significant benefit without sacrificing the familiarity we've come to depend on.
- Finally, Scala was developed by Martin Odersky, probably best known in the Java community for the Pizza and GJ languages, the latter of which became the working prototype for Java 5's generics. As such, it comes with a feeling of "seriousness"; this language was not created on a whim, and it will not be abandoned in the same fashion.

As Scala's name suggests, it is also a highly *scalable* language. I'll explain more about that once we're a little deeper into this series.

Download and install Scala

You can [download](#) the Scala bundle from the Scala Web site. The current release as of this writing is 2.6.1-final. It is available in a Java installer version, RPM and Debian packages, gzip/bz2/zip bundles that can simply be unpackaged in the target directory, and a source tarball that can be built from scratch. (Version 2.5.0-1 is available for Debian users from the Debian Web site with a simple "apt-get install." The 2.6 version has some subtle differences, though, so downloading and installing directly from the Scala Web site is recommended.)

Install Scala into the target directory of your choice — I'm in a Windows® environment as I write this, so mine will be the directory C:/Prg/scala-2.6.1-final. Define an environment variable, `SCALA_HOME`, to be this directory, and put `SCALA_HOME\bin` on your `PATH` for easy invocation from the command-line. To test your installation, just fire up "`scalac -version`" from a command prompt. It should respond with the Scala version, 2.6.1-final.

Functional concepts

Before we begin, I'll lay out a few functional concepts that are required to understand why Scala looks and acts the way it does. If you've spent some time with a functional language — Haskell, ML, or the more recent entry into the functional world, F# — you can [skip to the next section](#).

Functional languages get their name from the concept that programs should behave like mathematical functions; in other words, given a set of inputs, a function should always return the same output. Not only does this mean that every function must return a value, but that functions must inherently carry no intrinsic state from one call to the next. This intrinsic notion of statelessness, carried over into the functional/object world to mean immutable objects by default, is a large part of why functional languages are being hailed as the great saviors of a madly concurrent world.

Unlike many of the dynamic languages that have recently begun to carve out a

space of their own on the Java platform, Scala is statically typed, just as Java code is. Unlike the Java platform, however, Scala makes heavy use of *type inferencing*, meaning the compiler analyzes the code deeply to determine what type a particular value is, without programmer intervention. Type inferencing requires less redundant type code. For example, consider the Java code required to declare local variables and assign to them, shown in Listing 1:

Listing 1. The genius of javac (sigh)

```
class BrainDead {
    public static void main(String[] args) {
        String message = "Why does javac need to be told message is a String?" +
            "What else could it be if I'm assigning a String to it?";
    }
}
```

Scala requires no such hand-holding, as I'll show you later on.

Numerous other functional features (such as pattern matching) have made their way into the Scala language, but to list them all would be getting ahead of the story again. Scala also adds in a number of features currently missing in Java programming, such as operator overloading (which, it turns out, isn't at all like how most Java developers imagine it), generics with "upper and lower type bounds," views, and more. These features, among others, make Scala extremely powerful for handling certain tasks, such as processing or generating XML.

But enough of the abstract overview: programmers like to see code, so let's take a look at what Scala can do.

Getting to know you

Our first Scala program will be the standard demonstration program, Hello World, as is required by the Gods of Computer Science:

Listing 2. Hello.Scala

```
object HelloWorld {
    def main(args: Array[String]): Unit = {
        System.out.println("Hello, Scala!")
    }
}
```

Compile this with `scalac Hello.scala` and then run the resulting code either by using the Scala launcher (`scala HelloWorld`) or by using the traditional Java launcher, taking care to include the Scala core library on the JVM's classpath (`java -classpath %SCALA_HOME%\lib\scala-library.jar;. HelloWorld`). Either way, the traditional greeting should appear.

Some elements in Listing 2 are surely familiar to you, but some very new elements are at work here too. For instance, starting with the familiar call to

`System.out.println` demonstrates Scala's fidelity to the underlying Java platform. Scala goes to great lengths to make the full power of the Java platform available to Scala programs. (In fact, it will even allow a Scala type to inherit from a Java class, and vice versa, but more about that later.)

On the other hand, if you're observant, you'll have noticed the lack of a semicolon at the end of the `System.out.println` call; this is no typo. Unlike the Java platform, Scala does not require the semicolon to terminate a statement if the termination is obvious by the line ending. Semicolons are still supported, however, and are sometimes necessary if, for example, more than one statement appears on the same physical line. For the most part, the budding Scala programmer can simply leave off the semicolons, and the Scala compiler will gently remind him or her (usually with a glaring error message) when one is necessary.

Also, although this is a minor nit, Scala does not require the file containing a class definition to mirror the name of the class. Some will find this a refreshing change from Java programming; those who do not can continue to use the Java class-to-file naming convention without problem.

Now let's look at where Scala truly starts to diverge from traditional Java/object-oriented code.

Function and form, together at last

For starters, the Java aficionado will notice that instead of `class`, `HelloWorld` is defined using the keyword `object`. This is Scala's nod to the pervasiveness of the Singleton pattern — the `object` keyword tells the Scala compiler that this will be a singleton object, and as a result Scala will ensure that only one instance of `HelloWorld` ever exists. For this same reason, notice that `main` is not defined as a static method, as it would be in Java programming. In fact, Scala eschews the use of `static` altogether. If an application needs to have both instances of a type as well as some kind of "global" instance, a Scala application will allow both a `class` definition and an `object` definition of the same name.

Next, notice the definition of `main`, which, as with Java code, is the accepted entry point for Scala programs. Its definition, although it looks different from Java's, is identical: `main` takes an array of `Strings` as an argument and returns nothing. In Scala, however, this definition looks a tad different from Java's version. The definition of the `args` parameter is defined as `args: Array[String]`.

In Scala, arrays are represented as instances of the genericized `Array` class, which thus also reveals that Scala uses square brackets ("`[]`") instead of angle brackets ("`<>`") to indicate parameterized types. In addition, this pattern of `"name: type"` appears throughout the language, for consistency.

As with other traditional functional languages, Scala requires that functions (in this case a method) must always return a value. So, it returns the "non-value" value called `Unit`. For all practical purposes, Java developers can think of `Unit` as the same as `void`, at least for the time being.

The syntax for a method definition looks somewhat interesting, as it uses the "=" operator, almost as if it is assigning the method body that follows to the identifier `main`. In fact, this is precisely what's taking place: In a functional language, functions are first-class concepts, like variables and constants, and so are syntactically treated as such.

Did you say closures?

One implication of functions as first-class concepts is that they must somehow be recognizable as stand-alone constructs, a.k.a. *closures*, something the Java community has been hotly debating recently. In Scala, this is easily done. Before demonstrating the power of closures, consider the simple Scala program in [Listing 3](#). Here, the function `oncePerSecond()` repeats its logic (in this case, printing to `System.out`) once per second.

Listing 3. Timer1.scala

```
object Timer
{
  def oncePerSecond(): Unit =
  {
    while (true)
    {
      System.out.println("Time flies when you're having fun(ctionally)...")
      Thread.sleep(1000)
    }
  }

  def main(args: Array[String]): Unit =
  {
    oncePerSecond()
  }
}
```

Unfortunately, this particular code isn't all that functional ... or even useful. For example, if I wanted to change the message displayed, I'd have to modify the body of the `oncePerSecond` method. The traditional Java programmer would do this by defining a `String` parameter to `oncePerSecond` to contain the message to display. But even that is sharply limited: Any other periodic tasks (such as pinging a remote server) will need their own version of `oncePerSecond`, a clear violation of the Don't Repeat Yourself rule. As [Listing 4](#) illustrates, closures offer a flexible and powerful alternative:

Listing 4. Timer2.scala

```
object Timer
{
  def oncePerSecond(callback: () => Unit): Unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(1000)
    }
  }
}
```

```
def timeFlies(): Unit =
{ Console.println("Time flies when you're having fun(ctionally)..."); }

def main(args: Array[String]): Unit =
{
  oncePerSecond(timeFlies)
}
}
```

Now things are starting to get interesting. In Listing 4, the function `oncePerSecond` takes a parameter, but its type is strange. Formally, the parameter called `callback` takes a function as a parameter. This is true so long as the function passed in takes no parameters (indicated by the `()`), and returns (indicated by the `=>`) nothing (indicated by the functional value `Unit`). Notice that then, in the body of the loop, I use `callback` to invoke the passed parameter function object.

Fortunately, I have such a function elsewhere in the program, called `timeFlies`. So I simply pass it to the `oncePerSecond` function from within `main`. (You will also notice that `timeFlies` makes use of a Scala-introduced class, `Console`, that serves the same basic purpose as `System.out` or the new `java.io.Console` class. This is purely an aesthetic issue; either `System.out` or `Console` would work here.)

Anonymous function, what's your function?

Now, this `timeFlies` function seems like something of a waste — after all, it really serves no other purpose beyond being passed in to the `oncePerSecond` function. So, I won't formally define it at all, shown in Listing 5:

Listing 5. Timer3.scala

```
object Timer
{
  def oncePerSecond(callback: () => Unit): Unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(1000)
    }
  }

  def main(args: Array[String]): Unit =
  {
    oncePerSecond(() =>
      Console.println("Time flies... oh, you get the idea. "))
  }
}
```

In Listing 5, the main function passes an arbitrary block of code as the parameter to `oncePerSecond`, looking for all the world like a *lambda* expression from Lisp or Scheme, which, in fact, is another kind of closure. This *anonymous function* again demonstrates the power of treating functions as first-class citizens, allowing you to genericize code in an entirely new dimension beyond inheritance. (Fans of the Strategy pattern will likely have already started to salivate uncontrollably.)

In fact, `oncePerSecond` is still too specific: it puts the unreasonable restriction in place that the callback will be invoked every second. I can genericize this further by taking a second parameter indicating how often to invoke the passed function, shown in Listing 6:

Listing 6. Timer4.scala

```
object Timer
{
  def periodicCall(seconds: Int, callback: () => Unit): Unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(seconds * 1000)
    }
  }

  def main(args: Array[String]): Unit =
  {
    periodicCall(1, () =>
      Console.println("Time flies... oh, you get the idea. "))
  }
}
```

This is a common theme in functional languages: create a high-level abstract function that does one thing, let it take a block of code (an anonymous function) as a parameter, and call that block of code from within the high-level function. Take walking across a collection of objects, for example. Rather than using the traditional Java iterator object inside of a for loop, a functional library will instead define a function — usually called "iter" or "map" — on the collection class that takes a function taking a single parameter (the object being iterated over). So, for instance, the already mentioned `Array` class has a function, `filter`, which is defined in Listing 7:

Listing 7. Partial listing of Array.scala

```
class Array[A]
{
  // ...
  def filter (p : (A) => Boolean) : Array[A] = ... // not shown
}
```

Listing 7 declares that `p` is a function that takes a parameter of the generic type specified by `A` and returns a boolean. The Scala documentation states that `filter` "Returns an array consisting of all elements of this array that satisfy the predicate `p`." This means that if I want to go back to my Hello World program for a moment and find all the command-line arguments that start with the letter "G," writing it is as simple as Listing 8:

Listing 8. Hello, G-men!

```
object HelloWorld
{
  def main(args: Array[String]): Unit = {
    args.filter( (arg:String) => arg.startsWith("G") )
  }
}
```

```
    .foreach( (arg:String) => Console.println("Found " + arg) )  
  }  
}
```

Here, `filter` takes the predicate, an anonymous function that implicitly returns a boolean (the result of the `startsWith()` call) and calls the predicate with every element in the "args" array. If the predicate returns true, it adds it to the results array. After walking through the entire array, it takes the results array and returns it, which is immediately then used as the source for a "foreach" call, which does exactly as it implies: `foreach` takes another function and applies that function to every element in the array (in this case, to simply display each one).

It's not too hard to imagine what the Java equivalent to `HelloG.scala` above would look like, and it's not too hard to recognize that the Scala version is much, much shorter, and a lot clearer, too.

Wrapping up

Programming in Scala is tantalizingly familiar and different at the same time. It's similar in that you get to work with the same core Java objects you've come to know and love over the years, but clearly different in the way you're supposed to think about decomposing a program down into parts. In this first article in the *Busy Java developer's guide to Scala*, I've given you just a glimpse of what Scala will let you do. There's much more yet to come, but for now, happy functionalizing!

Resources

Learn

- [Scala Web site](#): Learn all about Scala.
- "[Java EE meets Web 2.0](#)" (Constantine Plotnikov, Artem Papkov, Jim Smith; developerWorks, November 2007): Identifies principles of the Java EE platform that are incompatible with Web 2.0 and introduces technologies, including Scala, that close the gap.
- "[Java theory and practice: Stick a fork in it](#)" (Brian Goetz, developerWorks, November 2007): The fork-join abstraction provides a natural, Java-based mechanism for decomposing many algorithms to effectively exploit hardware parallelism.
- "[Functional programming in the Java language](#)" (Abhijit Belapurkar, developerWorks, July 2004): Explains the benefits and uses of functional programming from a Java developer's perspective.
- [Programming in Scala](#) (Martin Odersky, Lex Spoon, and Bill Venners; Artima, December 2007): The first book-length introduction to Scala, co-authored by Scala creator Martin Odersky.
- [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Scala](#): Download Scala and start learning it with this series!

Discuss

- [developerWorks blogs](#): Get involved in the [developerWorks community](#).

About the author

Ted Neward

Ted Neward is the principal of Neward & Associates, where he consults, mentors, teaches, and presents on Java, .NET, XML Services, and other platforms. He resides near Seattle, Washington.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.