

The busy Java developer's guide to Scala: Don't get thrown for a loop!

Inside Scala's control structures

Skill Level: Introductory

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

26 Mar 2008

Scala was written specifically for the Java™ platform, so its syntax is designed to make Java coders feel at ease. At the same time, Scala brings to the JVM the inherent power of functional languages -- and those functional design concepts take some getting used to. In this installment of [The busy Java developer's guide to Scala series](#), Ted Neward starts introducing you to the subtle differences between the two languages, starting with control constructs such as `if`, `while`, and `for`. As you'll learn, Scala gives these constructs a power and complexity you won't find in their Java equivalents.

So far in this [series](#) I have focused on Scala's fidelity to the Java ecosystem, showing you how Scala incorporates much of Java's core object functionality. If Scala were just another way to write objects, though, it wouldn't be anywhere near as interesting -- or as powerful -- as it is. Scala's union of functional and object concepts, combined with its emphasis on programmer efficiency, make learning the language a more complex and subtle exercise than the Java-cum-Scala programmer might recognize right away.

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 -- and therein lies much of Scala's charm! After all, if Java could do it, why bother learning Scala?

For instance, take Scala's approach to control constructs like `if`, `while`, and `for`. While these may look like good old Java constructs to you, Scala imbues them with some wildly different characteristics. Rather than let you risk the frustration of discovering the differences *after* you've made a bunch of mistakes (and written buggy code as a result), this month's article is a primer on what to expect when using control structures in Scala.

Person.scala revisited

In the [last article in this series](#), you saw that Scala can define POJOs by defining methods that mimic the traditional "getters and setters" required by POJO-based environments. After that article was published I received an email from Bill Venners, co-author of the forthcoming canonical Scala reference, *Programming in Scala* (see [Resources](#)). Bill pointed out an even simpler way to do this, using the `scala.reflect.BeanProperty` annotation, like so:

Listing 1. A revised Person.scala

```
class Person(fn:String, ln:String, a:Int)
{
  @scala.reflect.BeanProperty
  var firstName = fn

  @scala.reflect.BeanProperty
  var lastName = ln

  @scala.reflect.BeanProperty
  var age = a

  override def toString =
    "[Person firstName:" + firstName + " lastName:" + lastName +
     " age:" + age + "]"
}
```

The approach in Listing 1 (a revision of Listing 13 from my [previous article](#)) generates the `get/set` method pairs for the `var` specified. The only drawback is that the methods do not actually exist in Scala code and cannot, thus, be called by other Scala code. This normally isn't a big deal because Scala will use the generated methods around the fields it generates for itself; but it could be surprising if you didn't know about it ahead of time.

What most strikes me, looking at the code in Listing 1, is how Scala doesn't just demonstrate the power of functional and object concepts in combination. It also demonstrates some of the benefits of thinking about an object language thirteen years after Java's initial release.

Control is an illusion

Much of the strange magic you're about to see is due to Scala's functional nature, so a little background on the development and evolution of functional languages is probably in order.

In functional languages, it is not usual to build higher and higher-level constructs directly into the language. Instead, the language is defined by a core set of primitive constructs. When combined with the ability to pass functions as objects, the result is *higher-order functions*, which can be used to define functionality that *looks* like it's coming out of the core language, but in fact is just a library. Like any library, this functionality can be replaced, augmented, or extended.

This *compositional* nature, of building a language up from a core set of primitives has a long and rich history, starting as far back as the 1960s and 1970s with Smalltalk, Lisp, and Scheme. Languages like Lisp and Scheme in particular grew a zealous following based on their ability to define higher-level abstractions on top of lower-level ones. Programmers took high-level abstractions and used them to build even more high-level ones. When you hear this process discussed today, it's usually in reference to domain-specific languages, or DSLs (see [Resources](#)). Really, though, it's just about building abstractions on top of abstractions.

In the Java language, your only option is to do this using API calls; in Scala, you can do it by appearing to extend the language itself. Trying to extend the Java language risks creating corner cases that threaten the stability of the whole thing. Trying to extend Scala just means creating a new library.

If only, my friend

We'll start with the traditional `if` construct -- certainly this has to be one of the simplest to address, correct? After all, in theory, the `if` simply checks a condition, and if the condition is true, executes the block of code that follows.

Ah, but such simplicity can be deceiving. Traditionally, the Java language has always made the `else` clause of an `if` optional, assuming that you can simply skip the block of code if the condition holds false. In a functional language, however, this is not the case. In keeping with the mathematical nature of a functional language, everything must evaluate to an expression, including the `if` clause itself. (For Java developers, this is exactly the way the ternary operator -- the `?:` expression -- works.)

In Scala, the non-true block (the `else` portion of the block) must be present and must result in the same kind of value that the `if` block does. This means that whichever way the code executes, a value always results. Consider the following

Java code, for example:

Listing 2. Which config file? (Java version)

```
// This is Java
String filename = "default.properties";
if (options.contains("configFile"))
    filename = (String)options.get("configFile");
```

Because the `if` construct in Scala is itself an expression, you can rewrite the above code to the arguably more correct snippet shown in Listing 3:

Listing 3. Which config file? (Scala version)

```
// This is Scala
val filename =
  if (options.contains("configFile"))
    options.get("configFile")
  else
    "default.properties"
```

val vs var

You might think there's more to know about the difference between `val` and `var`, but in truth, the difference is just that -- one is a read-only *value*, the other a mutable *variable*. In general, functional languages, particularly those considered "pure" functional languages (and thus permitting no side effects, such as mutable state) have only supported the `val` concept; but Scala, in its mandate to be attractive to both functional and imperative/object programmers, offers both.

That said, Scala programmers should, in general, choose the `val` construct first, only opting for `var` when it is obvious that mutability is necessary. The reason is quite simple: in addition to `val` making it easier to reason about a program's thread-safety, it is an underlying theme in Scala that developers do not need mutable state nearly as often as we think we do. Starting from immutable fields and locals (`vals`) is one way to demonstrate that, even to the largest Java skeptic. Starting with `final` in Java would be far less reasonable, due to Java's non-functional nature, though no less desirable. Curious Java developers may want to try it.

The real win, though, is that in Scala, the code can be written to assign the result to a `val`, instead of a `var`. Once set, a `val` cannot be changed, in much the same way that `final` variables operate in the Java language. The most notable side effect of the immutable local variable is that concurrency is now easier. Trying to do the same thing in your Java code would put you on the fringe of what many consider good, readable code, as shown in Listing 4:

Listing 4. Which config file? (Java, ternary style)

```
//This is Java
final String filename =
    options.contains("configFile") ?
        options.get("configFile") : "default.properties";
```

Explaining that one in a code review could be tricky. Correct, perhaps, but many Java programmers would look askance and ask, "What did you do that for?"

While you were out ...

Next, let's take a look at `while` and his little cousin, `do-while`. Fundamentally, each of these does exactly the same thing: it tests a condition, and if true, continues to execute the block of code provided.

Normally, functional languages eschew the `while` loop, because most of what `while` does can be done instead with recursion. Functional languages *really like* recursion. Consider, for example, the `quicksort` implementation shown off in "Scala by Example" (see [Resources](#)), which ships with the Scala implementation:

Listing 5. Quicksort (Java version)

```
//This is Java
void sort(int[] xs) {
    sort(xs, 0, xs.length - 1 );
}
void sort(int[] xs, int l, int r) {
    int pivot = xs[(l+r)/2];
    int a = l; int b = r;
    while (a <= b)
        while (xs[a] < pivot) { a = a + 1; }
        while (xs[b] > pivot) { b = b - 1; }
        if (a <= b) {
            swap(xs, a, b);
            a = a + 1;
            b = b - 1;
        }
    }
    if (l < b) sort(xs, l, b);
    if (b < r) sort(xs, a, r);
}
void swap(int[] arr, int i, int j) {
    int t = arr[i]; arr[i] = arr[j]; arr[j] = t;
}
```

Without going into too many of the details here, you can see that the use of the `while` loop is to iterate through the various elements in the array, finding a pivot point and sorting each of those sub-elements in turn. Not surprisingly, the `while` loop also requires a set of mutable local variables, in this case named `a` and `b`, in which to store the current pivots. Notice that this version even makes use of recursiveness in itself, calling itself twice, once to sort the left-hand side of the list,

and once again to sort the right-hand side of the list.

Suffice it to say that the `quicksort` in Listing 5 is not exactly easy to read, much less to reason about. Now, consider the *direct* (meaning, keeping as close as possible to the above version) equivalent in Scala:

Listing 6. Quicksort (Scala version)

```
//This is Scala
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        j -= 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

In itself, the code in Listing 6 looks pretty close to the Java version. Which is to say, it's long, ugly, difficult to reason about (particularly in the area around concurrency), and clearly holds no advantage over the Java version.

So, I'll improve it ...

Listing 7. Quicksort (better Scala version)

```
//This is Scala
def sort(xs: Array[Int]): Array[Int] =
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
      xs filter (pivot ==),
      sort(xs filter (pivot <))
    )
  }
```

Clearly the Scala code in Listing 7 is simpler. Note the use of recursion to avoid the while loop entirely. And the use of the `filter` function on the `Array` type to apply the "greater-than," "equals," and "less-than" functions to each element therein. And, to boot, the fact that, because the `if` expression is an expression returning a value, the return from `sort()` is implicitly the (single) expression in the definition of

```
sort().
```

In short, I've refactored the `while` loop's mutable state entirely into the parameters passed to the various invocations of `sort()` -- which many Scala aficionados would say is the proper way to write Scala code.

It's probably worth mentioning, again, that Scala itself does not mind if you use `while` instead of recursion -- you won't see a "What are you, stupid?" warning from the compiler, thank heavens. Scala also will not prevent you from writing code with mutable state. Using either `while` or mutable state sacrifices a key aspect of the Scala language, however, which is the encouragement to write code that parallelizes well. Where possible and practical, the "Scala way" suggests that you prefer recursion over imperative blocks.

Write your own language constructs

I'd like to take a quick detour from discussing Scala's control constructs to do something that most Java developers will find absolutely incredulous -- create your own language constructs.

True language wonks will find it interesting that the `while` loop, a primitive construct in Scala, could just as well be a predefined function. This idea is explored in the Scala documentation, along with a definition of this hypothetical "While":

```
// This is Scala
def While (p: => Boolean) (s: => Unit) {
  if (p) { s ; While(p)(s) }
}
```

The arguments above specify an expression that yields a boolean value and a block of code that returns nothing (`Unit`), which is exactly what `while` expects.

Extension along these lines could be written easily and used as needed, simply by importing the correct library. This, as I mentioned before, is the compositional approach to building up a language. Keep this in mind as we look at the `try` construct in the next section.

Try and try again

The `try` construct allows you to write code like this:

Listing 8. If at first you do not succeed....

```
// This is Scala
val url =
  try {
```

```
    new URL(possibleURL)
  }
  catch {
    case ex: MalformedURLException =>
      new URL("www.tedneward.com")
  }
}
```

The code in Listing 8 is far cry from the `if` examples from [Listing 2](#) or [Listing 3](#). In fact, it would be rather tricky to write in traditional Java code, particularly if you wanted the captured value to be stored in an immutable location (as I did with the `final` example in [Listing 4](#)). Another score for Scala's functional nature!

The `case ex:` syntax you see in Listing 8 is part of another Scala construct, the *match expression*, which is used for pattern matching in Scala. We'll explore pattern matching, which is a common feature of functional languages, a little later; for now, just think of it as a concept that is to `switch/case` what C-style structs are to classes.

Now take a second to think about exception-handling. Granted, Scala's support is interesting in that it is an expression, but one of the things developers often want is a standardized way to handle exceptions, not just the ability to catch them. In AspectJ, this was done by creating aspects that wove themselves around parts of your code, defined by pointcuts that had to be written carefully if you wanted different behavior in different parts of your codebase for different kinds of exceptions -- `SQLExceptions` should be handled differently from `IOExceptions`, and so on.

In Scala, it's trivial. Watch.

Listing 9. A custom expression for exceptions

```
// This is Scala
object Application
{
  def generateException()
  {
    System.out.println("Generating exception...");
    throw new Exception("Generated exception");
  }

  def main(args : Array[String])
  {
    tryWithLogging // This is not part of the language
    {
      generateException
    }
    System.out.println("Exiting main()");
  }

  def tryWithLogging (s: => _) {
    try {
      s
    }
    catch {
      case ex: Exception =>
        // where would you like to log this?
    }
  }
}
```

```
    // I choose the console window, for now
    ex.printStackTrace()
  }
}
```

Like the previously discussed `While` construct, the `tryWithLogging` code is simply a function call coming out of a library (or, in this case, out of the same class). Different variations of that theme can be used where appropriate without having to write complex pointcut code.

The beauty of this approach is that it leverages Scala's ability to capture cross-cutting logic in a first-class construct -- something previously only the aspect-oriented crowd could claim. The first-class construct in Listing 9 captures exceptions (both checked and unchecked) and handles them in a specific manner. Spin-offs from the idea above are limited only by the imagination. You just need to remember that Scala, like many functional languages, allows you to take blocks of code -- *aka* functions -- as parameters and apply them as desired.

A "for" generation language

All of this leads us to the real powerhouse of the Scala control construct suite: the `for` construct. What looks like a simple throwback to Java's enhanced `for` loop turns out to be far more powerful than anything the average Java programmer can begin to imagine.

Let's start out with a look at how Scala handles the simple sequential iteration through a collection, which you know so well from your Java programming:

Listing 10. For one and for all

```
// This is Scala
object Application
{
  def main(args : Array[String])
  {
    for (i <- 1 to 10) // the left-arrow means "assignment" in Scala
      System.out.println("Counting " + i)
  }
}
```

This does pretty much what you'd expect, looping 10 times and printing out the values each time. Be careful with this, though: the "1 to 10" expression doesn't mean that Scala has built-in awareness of integers and how to count from 1 to 10. Technically, what's going on here is more subtle: the compiler is using the method `to` defined on the `Int` type (everything in Scala is an object, remember?) to produce a `Range` object, which contains the elements to iterate over. If you rewrote the above code the way the Scala compiler sees it, it would look more like this:

Listing 11. What the compiler sees

```
// This is Scala
object Application
{
  def main(args : Array[String])
  {
    for (i <- 1.to(10)) // the left-arrow means "assignment" in Scala
      System.out.println("Counting " + i)
  }
}
```

So Scala's `for` doesn't, in fact, understand numbers any better than it does any other object type. What it understands is `scala.Iterable`, which defines the basic behaviors for iterating across a collection. Anything that provides an `Iterable` facility (technically a *trait* in Scala, but think of it for now as an interface) can be used as the heart of the `for` expression. `Lists`, `Arrays`, or even your own custom types can be used in a `for`.

Bringing Scala closer to English

You might note how much easier it is to understand Scala's version of the `for` loop in Listing 11. This is thanks to the fact that the `Range` object is implicitly inclusive on both sides, following the English language syntax more closely than Java does. Having a `Range` statement that says "from 1 to 10, do this" means no more accidental off-by-one errors.

For specificity

As it turns out, the `for` loop can do much more than just walk through an iterable list of items. In fact, a `for` loop can be used to filter the items along the way, producing a new list at each stage:

Listing 12. Count the (not odd) ways I love thee

```
// This is Scala
object Application
{
  def main(args : Array[String])
  {
    for (i <- 1 to 10; i % 2 == 0)
      System.out.println("Counting " + i)
  }
}
```

Notice the second clause to the `for` expression in Listing 12? It's a filter, and only those elements that pass the filter (that is, evaluate to *true*) will in fact "carry forward" to the body of the loop. In this case, only the even numbers from 1 to 10 will be printed.

The various stages of the `for` expression need not be "just" filters, either. You could even slide something entirely uneventful (from the perspective of the loop itself) into the pipeline. For instance, here's something that just displays what the current value of `i` is before evaluating it in the next stage:

Listing 13. How do I love thee? Verbosely, baby

```
// This is Scala
object App
{
  def log(item : _) : Boolean =
  {
    System.out.println("Evaluating " + item)
    true
  }

  def main(args : Array[String]) =
  {
    for (val i <- 1 to 10; log(i); (i % 2) == 0)
      System.out.println("Counting " + i)
  }
}
```

When run, each item in the range of 1 to 10 will be sent to `log`, which will tacitly "approve" each one by explicitly evaluating to `true`. But then the third clause of the `for` will kick in, filtering out only those elements that satisfy the criteria of being even. Thus, again, only the even numbers will be passed to the body of the loop itself.

For brevity

What would be a complex series of statements in Java code can be reduced a fairly simple expression in Scala. For example, here's one way to walk through a directory looking for all the `.scala` files and display the name of each one:

Listing 14. Finding `.scala`

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
    val filesHere = (new java.io.File(".")).listFiles
    for (
      file <- filesHere;
      if file.isFile;
      if file.getName.endsWith(".scala")
    ) System.out.println("Found " + file)
  }
}
```

This kind of filtering-for is common enough (and the semicolons are annoying enough, in this context) to merit the decision to omit the semicolons. Instead, Scala

lets you treat the statements in between the parentheses in the example above as if it were a block of code directly:

Listing 15. Finding .scala (version 2)

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
    val filesHere = (new java.io.File(".")).listFiles
    for {
      file <- filesHere
      if file.isFile
      if file.getName.endsWith(".scala")
    } System.out.println("Found " + file)
  }
}
```

As a Java developer you may find the original parentheses-with-semicolons syntax to be more intuitive, and the curlies-no-semicolons syntax harder to read. Fortunately, both syntactic approaches are identical in terms of the code they produce.

For fun

You can assign to more than one item within the clauses of a `for` expression, as shown in Listing 16.

Listing 16. What's in a name?

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
    // Note the array-initialization syntax; the type (Array[String])
    // is inferred from the initialized elements
    val names = Array("Ted Neward", "Neal Ford", "Scott Davis",
      "Venkat Subramaniam", "David Geary")

    for {
      name <- names
      firstName = name.substring(0, name.indexOf(' '))
    } System.out.println("Found " + firstName)
  }
}
```

This is called "midstream assignment," and it works almost exactly as shown here: the new value, `firstName`, is defined to hold the value of the `substring` call each time through the loop, and you can use it in the body of the loop thereafter.

This also gives rise to the idea of *nested* iteration, all inside the same expression:

Listing 17. Scala grep

```
// This is Scala
object App
{
  def grep(pattern : String, dir : java.io.File) =
  {
    val filesHere = dir.listFiles
    for (
      file <- filesHere;
      if (file.getName.endsWith(".scala") || file.getName.endsWith(".java"));
      line <- scala.io.Source.fromFile(file).getLines;
      if line.trim.matches(pattern)
    ) println(line)
  }

  def main(args : Array[String]) =
  {
    val pattern = ".*object.*"

    grep pattern new java.io.File(".")
  }
}
```

In this example, the `for` inside of `grep` uses two nested iterations -- one that iterates over all of the files found in the specified directory (where each file is bound to `file`, and one that iterates through all of the lines found inside the file currently being iterated (bound to the `line` local).

You can do a lot more with Scala's `for` construct, but the examples so far have hopefully made my point: Scala's `for` is actually a pipeline, processing collections of elements before passing them to the body, one at a time. Parts of this pipeline can either add more elements into the pipeline (generators), redact elements out of the pipeline (filters), or somewhere in between (such as the logging example). Regardless, Scala brings you a long way from the "enhanced `for` loop" introduced in Java 5.

Match me up

The last of Scala's control constructs you'll learn about today is `match`, which provides much of Scala's pattern-matching capabilities. Fundamentally, pattern matching declares blocks of code that are evaluated against a value. The first, closest match results in said code block being executed. So, for example, in Scala you could have:

Listing 18. A simple match

```
// This is Scala
object App
{
  def main(args : Array[String]) =
  {
```

```
for (arg <- args)
  arg match {
    case "Java" => println("Java is nice...")
    case "Scala" => println("Scala is cool...")
    case "Ruby" => println("Ruby is for wimps...")
    case _ => println("What are you, a VB programmer?")
  }
}
```

At first you might conceive of Scala pattern matching as a `String`-enabled "switch," with the underscore character doing its usual service as the wildcard, and therefore the default case in the typical switch block. Such thinking would vastly underestimate the language, however. Pattern matching is another feature found in many, if not most, functional languages, and offers some useful power.

For starters (though this probably shouldn't come as a surprise) consider the fact that the `match` expression itself yields a value, so it can be on the right-hand side of an assignment statement just as `if` and `try` can. This is useful in its own right, but the real power of matching comes when you use matching to match based on types, rather than values of a single type, as above, or more often, a combination of the two.

So, for example, imagine you have a function or method that is declared as returning `Object` -- a good example here might be the result of Java's `java.lang.reflect.Method.invoke()` method. Normally, with the Java language, to evaluate the result you would first determine its type and downcast; in Scala, however, you can use pattern matching to simplify that:

Listing 19. What are you?

```
//This is Scala
object App
{
  def main(args : Array[String]) =
  {
    // The Any type is exactly what it sounds like: a kind of wildcard that
    // accepts any type
    def describe(x: Any) = x match {
      case 5 => "five"
      case true => "truth"
      case "hello" => "hi!"
      case Nil => "the empty list"
      case _ => "something else"
    }

    println describe(5)
    println describe("hello")
  }
}
```

Because of `match`'s ability to easily and concisely describe how code should match against various values and types, pattern matching is commonly used in parsers and interpreters, where the current token in a parse stream is matched against a series

of possible match clauses. The next token is then applied against another series, and so on. (Note that this is partly why many language parsers, compilers, and other code-related tools are written in functional languages like Haskell or ML.)

There's a lot more to say about pattern matching but that would lead us straight to another Scala feature, *case classes*, which I think I'll save for another day.

In conclusion

Scala is deceptively similar to Java in a number of ways, and nowhere is this more obvious than in the `for` construct. The functional nature of the core syntactic elements provide some useful features (such as the assignment capabilities already mentioned), but also the ability to extend the language in new and interesting ways without having to modify the core `javac` compiler itself. This makes the language both more amenable to the definition of internal DSLs (those DSLs that are defined inside of an existing language's syntax) and to programmers looking to build abstractions up from a core set of primitives, *a la* Lisp or Scheme.

There's so much talk about in Scala, but our time this month has elapsed. Remember to grab the latest Scala bits (2.7.0-final at the time of this writing) and play around with the examples provided to get a feel for how the language operates (see [Resources](#)). Until next time, remember, Scala puts the fun(ctional) back into programming!

Resources

Learn

- [The busy Java developer's guide to Scala](#) (Ted Neward, developerWorks): Read the complete series.
- ["Crossing borders: Domain-specific languages in Active Record and Java programming"](#) (Bruce Tate, developerWorks, April 2006): Learn more about DSLs in Ruby.
- ["Functional programming in the Java language"](#) (Abhijit Belapurkar, developerWorks, July 2004): Understand the benefits and uses of functional programming from a Java developer's perspective.
- ["Scala by Example"](#) (Martin Odersky, December 2007): A short, code-driven introduction to Scala, including the Quicksort application used in this article (PDF).
- [Programming in Scala](#) (Martin Odersky, Lex Spoon, and Bill Venners; Artima pre-print published February 2008): The first book-length introduction to Scala, co-authored by Bill Venners.
- The [developerWorks Java technology zone](#): Hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Download Scala](#): Currently in version 2.7.0-final.

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.