# The busy Java developer's guide to Scala: Collection types

## Working with tuples, arrays, and lists in Scala

Ted Neward
Principal
Neward & Associates

27 June 2008

Objects have their place in Scala, but so do functional types such as tuples, arrays, and lists. In this installment of Ted Neward's popular series, you'll begin to explore the functional side of Scala, starting with its support for types common to functional languages.

View more content in this series

### 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 the Java™ developer learning Scala, objects provide a natural and easy point of entry. Over the past several articles in this series, I've introduced you to some of the ways that object-oriented programming in Scala really isn't much different from Java programming. I've also shown you how Scala revisits traditional object-oriented concepts, finds them wanting, and reinvents them for the 21st century. Something important has been lurking behind the curtain all this time, however, waiting to emerge: Scala is also a functional language. (Functional, I say, as opposed to all those other *dys*functional languages.)

Scala's functional orientation is worth exploring, and not only because you've run out of objects to play with. Functional programming in Scala will give you some new design constructs and ideas, as well as built-in constructs that make programming certain scenarios (such as concurrency) much, much easier.

You'll take your first real foray into functional programming with Scala this month, with a look at four types common to most functional languages: lists, tuples and sets, and the `Option` type. You'll also learn about Scala's arrays, which are actually pretty foreign to other functional languages.

### C# 2.0 nullable types

Other languages have attempted to solve the "nullability" problem in a variety of ways: C++ essentially ignored it for as long as possible before finally deciding that null and zero were different values. The Java language still hasn't fully resolved the issue, but relies on autoboxing — the automatic conversion of primitive types to their wrapper objects (which themselves weren't introduced until 1.1) — to carry Java programmers through. Some pattern aficionados suggest that each type should have a corresponding "Null Object," an instance of the type (actually a subtype) that has each of its methods overridden to do nothing — which turns out in practice to be a lot of work. After C# 1.0 shipped, the C# designers decided to take an entirely different approach to nullability.

C# 2.0 introduced the concept of *nullable types*, essentially adding syntactic support to suggest that any particular value type (basically, a primitive type) could be made to support *null* by wrapping it in a generic/template class, `Nullable<T>`, which itself was silently introduced via the `?` modifier in the type declaration. Thus, `int?` indicates an integer that might sometimes be null.

On the surface, this seemed a reasonable decision, but things quickly got complicated. Should `int` and `int?` be considered compatible types, and if so, when does an `int` get promoted to an `int?`, or vice versa? What happens when an `int` is added to an `int?`, and can that result ever be null? And so on. Several major twists and turns in the type system later, nullable types shipped as part of 2.0 — where they were almost completely ignored by C# programmers.

In retrospect, the functional approach of the `Option` type, which makes the line between `Option[T]` and `Int` clear, seems more straightforward than the others, particularly when compared against some of the counterintuitive promotion rules surrounding nullable types. (It doesn't hurt that the functional world has been meditating on this concept for close to two decades.) `Option[T]` takes a little getting used to, but in general, it seems to yield clearer code and expectations.

Each of these types offers a new way to think about writing code. When combined with traditional object-oriented features, they can produce startlingly concise results.

## Exercise your Option(s)

When is nothing not really nothing? When it's 0, as opposed to null.

For a concept that most of us understand particularly intimately, it turns out that trying to represent "nothing" is surprisingly difficult in software. Look at all the debates that raged within the C++ community around `NULL` and 0, for example, or the debates that rage inside the SQL community around `NULL` columnar values. `NULL`, or null, if you prefer, is the way most programmers think about "nothing," but this represents some particular problems in Java programming.

Consider a simple operation designed to look up a programmer's salary out of some in-memory or on-disk database: the API is designed to allow a caller to pass in a `String` containing the programmer's name, and it returns ... what? From a modeling perspective, it should return an `Int`, conveying the amount the programmer makes on a yearly basis; but there's the thorny problem of what to return in the event that the programmer isn't in the database. (Maybe she hasn't been hired, maybe she's been fired, maybe there's a typo in the name ...) If the return type is `Int`, then we can't return *null*, the usual "flag" indicating that the user wasn't found in the database. (You might think an exception should be thrown, but a missing value from a database isn't really an exceptional condition most of the time, so an exception here would be inappropriate.)

In Java code, we end up marking the method as returning `java.lang.Integer`, which forces callers to know that the method could return *null*. Naturally, we can rely on programmers to thoroughly document this scenario, and we can also rely on programmers to *read* the carefully-prepared documentation. Right ... just as we can rely on managers to thoroughly hear our objections to the impossible deadlines they levy on programmers and carefully take those objections back to their management and customers.

Scala offers a common functional alternative to this impasse. In some ways, the `Option` type, or `Option[T]`, defies description. It is a generic class with exactly two subclasses, `Some[T]` and `None`, which is used to help convey the possibility of "no value" without requiring the language type system to go through serious gyrations to support the concept. Actually using the `Option[T]` type, as we'll do in the next section, should make things more clear.

When working with `Option[T]`, the key is to realize that it is essentially a strongly-typed collection of size "one" that uses a different value, *None*, to represent the possibility of a "nothing" value. Thus, instead of a method returning null to indicate that no data was found, the method is declared to return `Option[T]`, where `T` is the original type returned. Then, in the scenarios where no data is found, simply return *None*, like so:

## Listing 1. Are you ready for some football?

```
@Test def simpleOptionTest =
{
  val footballTeamsAFCEast =
    Map("New England" -> "Patriots",
        "New York" -> "Jets",
        "Buffalo" -> "Bills",
        "Miami" -> "Dolphins",
        "Los Angeles" -> null)

  assertEquals(footballTeamsAFCEast.get("Miami"), Some("Dolphins"))
  assertEquals(footballTeamsAFCEast.get("Miami").get(), "Dolphins")
  assertEquals(footballTeamsAFCEast.get("Los Angeles"), Some(null))
  assertEquals(footballTeamsAFCEast.get("Sacramento"), None)
}
```

Note that the return value of a `get` on a Scala `Map` is not the actual value corresponding to the passed key. Instead, it's an `Option[T]` instance, either `Some()` around the value in question, or else `None`, which makes it clear when the key wasn't found in the map. This becomes particularly important if it is acceptable for a given key to exist in the map but have a corresponding value of null, as is the case for the *Los Angeles* key in Listing 1.

Most of the time, when working with `Option[T]`, programmers will use *pattern matching*, a highly functional concept that allows one to effectively "switch" on both types and/or values, not to mention bind values into variables at the point of definition, switch between `Some()` and `None`, and extract the value out of `Some` without having to call the deprecated `get()` method. Listing 2 shows Scala's pattern matching in action:

### Listing 2. A (pattern) match made in heaven

```
  @Test def optionWithPM =
  {
    val footballTeamsAFCEast =
      Map("New England" -> "Patriots",
          "New York" -> "Jets",
          "Buffalo" -> "Bills",
          "Miami" -> "Dolphins")

    def show(value : Option[String]) =
    {
      value match
      {
        case Some(x) => x
        case None => "No team found"
      }
    }

    assertEquals(show(footballTeamsAFCEast.get("Miami")), "Dolphins")
  }
```

## Tuples and sets

In C++, we called them structs. In Java programming, we called them data transfer objects or parameter objects. In Scala, we call them *tuples*. In essence, they are classes that simply collect a couple of other data types together into a single instance, with little to no attempt at encapsulation or abstraction — in fact, it is frequently more useful to *not* have any abstraction in place.

Creating a tuple type in Scala is ridiculously easy, which is part of the point: there is no value in creating a name that is descriptive of the elements inside the type if those elements are exposed to the public in the first place. Consider Listing 3:

### Listing 3. tuples.scala

```
// JUnit test suite
//
class TupleTest
{
  import org.junit._, Assert._
  import java.util.Date

  @Test def simpleTuples() =
  {
    val tedsStartingDateWithScala = Date.parse("3/7/2006")

    val tuple = ("Ted", "Scala", tedsStartingDateWithScala)

    assertEquals(tuple._1, "Ted")
    assertEquals(tuple._2, "Scala")
    assertEquals(tuple._3, tedsStartingDateWithScala)
  }
}
```

Creating a tuple is a simple matter of putting the values inside a set of parentheses, almost as if they were inside a method call invocation. Extracting the values requires nothing more than invoking the "`_n` method, where *n* is the positional argument of the tuple element of interest: `_1` for the first, `_2` for the second, and so on. A traditional Java `java.util.Map`, then, is essentially a collection of two-part tuples.

Tuples make it trivial to carry multiple values around as a single entity, which means that tuples can offer something that would otherwise be extremely heavyweight to do in Java programming: multiple return values. For example, a method may offer to count the number of characters in a `String` and return the most popular character in that `String`, but if a programmer wants to know *both* the most popular character *and* the number of times it appeared, then the design gets tricky here: either an explicit class containing both the character and its count will need to be created, or else the values will have to be held in an object as fields and those field values returned when asked. Either way, it's going to be a fairly long set of code, compared to the Scala version; by simply returning the tuple containing the character and its associated count, not to mention the easy "_1", “_2”, … access to each of the values in the tuple, Scala makes it easy to return multiple return values.

By simply returning the tuple containing the character and its associated count, not to mention the easy `_1` access to each of the values in the tuple, Scala makes it easy to return multiple return values.

As you'll see in the next section, Scala programmers frequently store `Option`s and tuples in collections (such as `Array[T]` or lists), which gives us tremendous flexibility and power in a relatively simple construct.

## Array of sunshine on a cloudy day

Let's first revisit an old friend — the array — now under new management as `Array[T]`. Like arrays in Java code, Scala's `Array[T]` is an ordered sequence of elements, indexed by a numeric value that represents the position in the array and must not exceed the total size of the array, as shown in Listing 4:

### Listing 4. array.scala

```
object ArrayExample1
{
  def main(args : Array[String]) : Unit =
  {
    for (i <- 0 to args.length-1)
    {
      System.out.println(args(i))
    }
  }
}
```

Although they're equivalent to arrays in Java code (which is what they compile down to, after all), arrays in Scala are definitely defined differently. For starters, arrays in Scala are effectively generic classes, with no "built-in" status accrued to them (at least, no more than any other class that ships with the Scala library). For example, in Scala, arrays are formally defined as instances of `Array[T]`, a class with a number of interesting additional methods defined on it, including the ubiquitous "length" method that, not surprisingly, returns the length of the array. Thus, in Scala, it's possible to use `Array` in the traditional sense, such as iterating using an `Int` from 0 to `args.length - 1`, and obtaining the *i'th* element of the array. (You use parentheses instead of square brackets to specify which element to return, which in fact is another one of those methods with funny names.)

But that's no fun ... and it's no fun(ctional). (Sorry. Bit of programming humor there. Right. Moving on.)

**Tip**

See the Scaladocs for the complete `Array[T]` hierarchy. It's actually quite nice and reminiscent of the `java.util Collections` classes in many ways.

## Extending Array

It turns out that `Array` has a large number of methods on it that it inherits from a surprisingly rich parent hierarchy: `Array` extends `Array0`, which extends `ArrayLike[A]`, which extends `Mutable[A]`, which extends `RandomAccessSeq[A]`, which extends `Seq[A]`, and so on, and so on, and so on. Naturally, all this parentage means that `Array` has a good many operations against it, making it easier to work with arrays in Scala when compared to Java programming.

For example, iterating across an array, as I did in Listing 4, can be done in an (arguably) much simpler and (not arguably) much more functional manner using the `foreach` method, which it inherits from the `Iterable` trait:

## Listing 5. ArrayExample2

```
object
{
  def main(args : Array[String]) : Unit =
  {
    args.foreach( (arg) => System.out.println(arg) )
  }
}
```

It may not seem like you've saved much, but the ability to pass a function (anonymous or otherwise) into another class for execution under particular semantics — in this case, while iterating across the array — is a common theme in functional programming. And the use of higher order functions in this way is hardly restricted to iteration; in fact, it's not uncommon to do some kind of *filtration* process on the contents of an array to weed out unworthy candidates, then somehow process the results. For instance, in Scala, it's easy to use the `filter` method for filtration, then take the resulting list and process each element using either `map` and another function (this time of type `(T) => U`, where T and U are again both generic types), or `foreach` again. I've tried the latter approach in Listing 6. (Note that `filter` takes a `(T) : Boolean`method, meaning it takes a single parameter of whatever type the array holds and returns a `Boolean`.)

## Listing 6. Find all the Scala programmers

```
class ArrayTest
{
  import org.junit._, Assert._

  @Test def testFilter =
  {
    val programmers = Array(
        new Person("Ted", "Neward", 37, 50000,
          Array("C++", "Java", "Scala", "Groovy", "C#", "F#", "Ruby")),
        new Person("Amanda", "Laucher", 27, 45000,
          Array("C#", "F#", "Java", "Scala")),
        new Person("Luke", "Hoban", 32, 45000,
          Array("C#", "Visual Basic", "F#")),
  new Person("Scott", "Davis", 40, 50000,
    Array("Java", "Groovy"))
      )

    // Find all the Scala programmers ...
    val scalaProgs =
      programmers.filter((p) => p.skills.contains("Scala") )

    // Should only be 2
    assertEquals(2, scalaProgs.length)

    // ... now perform an operation on each programmer in the resulting
    // array of Scala programmers (give them a raise, of course!)
    //
    scalaProgs.foreach((p) => p.salary += 5000)

    // Should each be increased by 5000 ...
    assertEquals(programmers(0).salary, 50000 + 5000)
    assertEquals(programmers(1).salary, 45000 + 5000)

    // ... except for our programmers who don't know Scala
    assertEquals(programmers(2).salary, 45000)
 assertEquals(programmers(3).salary, 50000)
  }
}
```

The `map` function would be used where a new `Array` was to be created, leaving the original array contents untouched, which is actually the way most functional programmers would prefer to operate:

## Listing 7. Filter and map

```
  @Test def testFilterAndMap =
  {
    val programmers = Array(
        new Person("Ted", "Neward", 37, 50000,
          Array("C++", "Java", "Scala", "C#", "F#", "Ruby")),
        new Person("Amanda", "Laucher", 27, 45000,
          Array("C#", "F#", "Java", "Scala")),
        new Person("Luke", "Hoban", 32, 45000,
          Array("C#", "Visual Basic", "F#"))
  new Person("Scott", "Davis", 40, 50000,
    Array("Java", "Groovy"))
      )

    // Find all the Scala programmers ...
    val scalaProgs =
      programmers.filter((p) => p.skills.contains("Scala") )

    // Should only be 2
    assertEquals(2, scalaProgs.length)

    // ... now perform an operation on each programmer in the resulting
    // array of Scala programmers (give them a raise, of course!)
    //
    def raiseTheScalaProgrammer(p : Person) =
    {
      new Person(p.firstName, p.lastName, p.age,
        p.salary + 5000, p.skills)
    }
    val raisedScalaProgs =
      scalaProgs.map(raiseTheScalaProgrammer)
    assertEquals(2, raisedScalaProgs.length)
    assertEquals(50000 + 5000, raisedScalaProgs(0).salary)
    assertEquals(45000 + 5000, raisedScalaProgs(1).salary)
  }
```

Notice that in Listing 7 the salary member of `Person` could be marked as "`val`, making it immutable, rather than the "`var` version I had to use above in order to modify the various programmers' salaries.

Scala's `Array` holds more methods than I could possibly list or demonstrate here. As general advice, when working with arrays, aggressively look for ways to take advantage of the methods provided on `Array` rather than using the traditional `for ...` pattern to walk through the array and find or do what needs to be found or done. The easiest way to do this, usually, is to write a function (nested, if necessary, as in the `testFilterAndMap` example in Listing 7) that performs the desired action, then pass that to one of the `map`, `filter`, `foreach`, or other methods on `Array`, depending on the desired result.

## Functional fun with lists

A core feature of functional programming for years, lists have the same degree of "built-in"-ness that arrays have enjoyed for years in the object space. Lists are fundamental to building functional software, and as such you (as a budding Scala programmer) must be able to understand them and how they work. Even if they never factor into a new design, Scala code uses lists extensively throughout its libraries. So learning lists is ... well ... *imperative*.

(Sorry. More functional programming humor there. Won't happen again.)

In Scala, the list is like the array in that its core definition is a standard class from the Scala library, `List[T]`. And, like `Array[T]`, `List[T]` inherits from a number of base classes and traits, starting with `Seq[T]` as an immediate base.

Fundamentally, lists are collections of elements that can be extracted via either the head or the tail of the list. The list comes to us courtesy of Lisp, which trivia buffs will remember as being the name for a language centered primarily on "LISt Processing," where obtaining the head of the list came via the `car` operation, and the tail via the `cdr` operation. (The reasons for the names are historical; bonus points to the first person to send me the reason.)

Working with lists is actually easier than working with arrays in many ways, both because functional languages historically have had very good support for working with lists (which Scala inherits), and because lists compose and decompose nicely. For example, frequently a function will want to pick apart the contents of a list. To do so, it will pick off the first element of the list — the head — to do the processing on that element, and recursively pass the rest of the list to itself again. This greatly reduces the possibility that there will be some kind of shared state inside the processing code and also makes it more likely that (if the processing is somehow non-trivial) the code can be spun out to multiple threads, given each step needs only work with one element.

Putting lists together and taking them apart is pretty straightforward, as you can see in Listing 8:

## Listing 8. And we're... Listing

```
class ListTest
{
  import org.junit._, Assert._

  @Test def simpleList =
  {
    val myFirstList = List("Ted", "Amanda", "Luke")

    assertEquals(myFirstList.isEmpty, false)
    assertEquals(myFirstList.head, "Ted")
    assertEquals(myFirstList.tail, List("Amanda", "Luke")
    assertEquals(myFirstList.last, "Luke")
  }
}
```

Notice that constructing a list is much the same as constructing an array; both work similarly to constructing regular objects, except without the "new" being required. (This is the functionality of a "case class," which we'll explore in a future article.) Pay particularly close attention to the result of the `tail` method call — the result is not the last element in the list (which is given by `last`), but the remainder of the list without the first element.

Of course, part of the power of lists comes in recursive processing of the list elements, which simply means pulling the head from the list until the list is empty, then accumulating the results:

## Listing 9. Recursive processing

```
  @Test def recurseList =
  {
    val myVIPList = List("Ted", "Amanda", "Luke", "Don", "Martin")

    def count(VIPs : List[String]) : Int =
    {
      if (VIPs.isEmpty)
        0
      else
        count(VIPs.tail) + 1
    }

    assertEquals(count(myVIPList), myVIPList.length)
  }
```

Note that if I leave out the return type of `count`, the Scala compiler or interpreter is going to get grumpy with me — because this is a tail-recursive call, an optimization designed to reduce the number of stack frames created in heavily recursive operations, it needs to have its return type specified. Granted, it's certainly easier to just use the "length" member of `List` to get the number of items in the list, but the point is to illustrate, "in the large," how list processing can be so powerful. The entire method in Listing 9 is completely thread-safe. I'm certain of it because the entirety of the intermediate state used in its processing is held on the stack in parameters, and therefore, by definition, cannot be accessed by multiple threads. The beautiful thing about the functional approach is that it's actually quite difficult to program functionally and still create shared state.

## List APIs

Lists have some other interesting properties to them, such as alternative ways to construct a list, using the `::` method. (Yes, method. Just another method with a funny name, folks. Nothing to see here, move along.) So, instead of constructing lists using the "`List` constructor syntax, you can "cons" them (as the double-colon method is called) together, like so:

## Listing 10. I thought :: == C++?

```
@Test def recurseConsedList =
{
  val myVIPList = "Ted" :: "Amanda" :: "Luke" :: "Don" :: "Martin" :: Nil

  def count(VIPs : List[String]) : Int =
  {
    if (VIPs.isEmpty)
      0
    else
      count(VIPs.tail) + 1
  }

  assertEquals(count(myVIPList), myVIPList.length)
}
```

Be careful when using the `::` method — it introduces a couple of funny rules. This syntax is so common in functional languages that Scala's creators chose to support it, but for the syntax to work correctly and generally, one quirky rule must kick in: Any "method-with-a-funny-name" that ends in a colon is *right-associative*, meaning the whole expression begins on the far-right-hand side of the expression, with `Nil`, which conveniently happens to be a `List`. So, this means `::` can be determined to be the global `::` method, as opposed to a member method of (in this case) `String`; which in turn means you can build a list out of everything. When using `::`, the right-hand-most element must be a list, or you'll get an error message.

One of the most powerful ways to use lists in Scala is in combination with pattern matching, thanks to the fact that not only can a list match on types as well as values, but it can also do variable binding at the same time. For example, I can simplify the list code from Listing 10 by using pattern matching to realize between a list with at least one element in it, and one that is empty:

## Listing 11. Pattern matching in a list

```
@Test def recurseWithPM =
{
  val myVIPList = "Ted" :: "Amanda" :: "Luke" :: "Don" :: "Martin" :: Nil

  def count(VIPs : List[String]) : Int =
  {
    VIPs match
    {
      case h :: t => count(t) + 1
      case Nil => 0
    }
  }

  assertEquals(count(myVIPList), myVIPList.length)
}
```

In the first `case` expression, the head of the list will be carved off and bound to the variable *h*, whereas the remainder (the tail) will be bound to *t*; in this case, nothing is done with *h* (in fact, it might be better to indicate that the head will never be used by replacing *h* with the wildcard _, to indicate that it's a placeholder for a variable that will never be used). But *t* is recursively passed to `count` again, just as with the earlier example. Remember, too, that every expression in Scala returns a value implicitly; in this case, the result of the pattern matching expression will be the recursive call to `count + 1`, or `0` when we hit the end of the list.

Given that both weigh in at about the same number of lines of code, where is the value of using pattern matching? Truthfully, for something this simple, the value is hard to find. But for something even slightly more complicated, such as extending the sample to catch particular values along the way, you might reconsider.

## Listing 12. Pattern-matcher, pattern-matcher, match me Amanda!

```
@Test def recurseWithPMAndSayHi =
{
  val myVIPList = "Ted" :: "Amanda" :: "Luke" :: "Don" :: "Martin" :: Nil

  var foundAmanda = false
  def count(VIPs : List[String]) : Int =
  {
    VIPs match
    {
      case "Amanda" :: t =>
        System.out.println("Hey, Amanda!"); foundAmanda = true; count(t) + 1
      case h :: t =>
        count(t) + 1
      case Nil =>
        0
    }
  }

  assertEquals(count(myVIPList), myVIPList.length)
  assertTrue(foundAmanda)
}
```

It doesn't take too long before really complex examples, particularly in the areas of regular expressions or XML nodes, begin to weigh heavily in favor of the pattern-matching approach. Pattern matching also isn't limited to lists; there's no reason why it couldn't be extended back to the array examples from earlier. In fact, here's an array example of the `recurseWithPMAndSayHi` test above:

## Listing 13. Pattern-matcher, pattern-matcher, match me an array

```
@Test def recurseWithPMAndSayHi =
{
  val myVIPList = Array("Ted", "Amanda", "Luke", "Don", "Martin")

  var foundAmanda = false

  myVIPList.foreach((s) =>
    s match
    {
      case "Amanda" =>
        System.out.println("Hey, Amanda!")
        foundAmanda = true
      case _ =>
        ; // Do nothing
    }
  )

  assertTrue(foundAmanda)
}
```

If you'd like an exercise, try building a recursive version of Listing 13 that also does the count without a mutable `var` declared in the `recurseWithPMAndSayHi` scope. Hint: multiple pattern-matching blocks may be necessary. (This article's code download includes one solution — but I hope you'll try it yourself before peeking.)

## Conclusion

### What is right-associative?

To better understand what's going on with `::`, remember that operators, like the "cons" operator, are just methods-with-funny-names. Given the normal left-associative syntax, the token on the left is going to be the object on which I invoke the method name (the token on the right). So, normally, the expression `1 + 2` is going to be seen by the compiler as `1.+(2)` .

But in the case of the list, this isn't going to work — every class in the system would need to have the `::` method on every type in the system, and that would be a pretty horrific violation of separation of concerns.

Scala fixes this by saying that any method-with-a-funny-name that ends in a colon (such as `::` or `:::` or even methods I make up myself, such as `foo:`) will be right-associative. So, for instance, `a :: b :: c :: Nil` translates to `Nil.::(c).::(b).::(a)`, which is precisely what I wanted: a `List` to start the whole thing off, so that each call to `::` can take the objet parameter and return a `List`, thus continuing the chain.

It might be nice to specify the right-associative property for other naming conventions, but as of this writing Scala has the rule hard-coded into the language. For now, the colon is the only character that triggers right-associative behavior.

Scala's rich collection of collections (pardon the pun) is a direct result of some of its functional history and feature set; tuples provide an easy way to gather a loosely bound set of values; `Option[T]` provides an easy way to indicate "some" value as opposed to "no" value in a straightforward way; arrays give access to traditional Java-style array semantics with some enhanced features; lists are the principal collection of functional languages, and so on.

Be careful with some of these features, however, particularly tuples: it could be easy to get caught up in using them and forget traditional, basic object modeling in favor of using tuples directly. If a particular tuple — a name, age, salary, and list of programming languages known, for example — appears routinely in the code-base, go ahead and model it as a formal class type and object.

The beautiful thing about Scala is that it's both functional *and* object-oriented, so you can keep a traditional eye on class design, even while you learn to enjoy Scala's functional types.

## Downloads

| Description | Name | Size |
|---|---|---|
| Sample Scala code for this article | j-scala06278.zip | 200KB |

## Resources

### Learn

- *The busy Java developer's guide to Scala* (Ted Neward, developerWorks): Read the complete series.
- "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.
- "Dead like COBOL" (Ted Neward, developerWorks, May 2008): Is it time to leave the Java platform behind for greener pastures? Ted considers the arguments for and against Java's demise.
- Ted Neward on why we need Scala (JavaWorld, June 2008): Hear more of Ted's thoughts about functional programming and Scala's place in the Java ecosystem, in this podcast discussion with Andrew Glover.
- "Scala by Example" (Martin Odersky, December 2007): A short, code-driven introduction to Scala (in PDF).
- *Programming in Scala* (Martin Odersky, Lex Spoon, and Bill Venners; Artima, December 2007): The first book-length introduction to Scala.
- Scaladocs: The API specification for Scala Library.
- The developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.

### Get products and technologies

- Download Scala: 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.