

# The busy Java developer's guide to Scala: Class action

## Understand Scala's class syntax and semantics

Skill Level: Introductory

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

19 Feb 2008

It makes sense for Java™ developers to use objects as a first point of reference for understanding Scala. In this second installment of *The busy Java developer's guide to Scala series*, Ted Neward follows a basic premise of language measurement: that the power of a language can be measured in direct relation to its ability to integrate new facilities -- in this case, support for complex numbers. Along the way you'll see some interesting tidbits related to class definitions and usage in Scala.

In last month's [article](#), you saw just a touch of Scala's syntax, the bare minimum necessary to run a Scala program and observe some of its simpler features. The Hello World and Timer examples from that article let you see Scala's `Application` class, its syntax for method definitions and anonymous functions, just a glimpse of an `Array[ ]`, and a bit on type-inferencing. Scala has a great deal more to offer, so this article investigates the intricacies of Scala coding.

### About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this new developerWorks [series](#), 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?

Scala's functional programming features are compelling, but they're not the only reason Java developers should be interested in the language. In fact, Scala blends functional concepts and object orientation. In order to let the *Java-cum-Scala* programmer feel more at home, it makes sense to look at Scala's object features and see how they map over to Java linguistically. Bear in mind that there isn't a direct mapping for some of these features, or in some cases, the "mapping" is more of an analog than a direct parallel. But where the distinction is important, I'll point it out.

## Scala has class(es), too

Rather than embark on a lengthy and abstract discussion of the class features that Scala supports, let's look at a definition for a class that might be used to bring rational number support to the Scala platform (largely swiped from "Scala By Example" -- see [Resources](#)):

### Listing 1. rational.scala

```
class Rational(n:Int, d:Int)
{
  private def gcd(x:Int, y:Int): Int =
  {
    if (x==0) y
    else if (x<0) gcd(-x, y)
    else if (y<0) -gcd(x, -y)
    else gcd(y%x, x)
  }
  private val g = gcd(n,d)

  val numer:Int = n/g
  val denom:Int = d/g

  def +(that:Rational) =
    new Rational(numer*that.denom + that.numer*denom, denom * that.denom)
  def -(that:Rational) =
    new Rational(numer * that.denom - that.numer * denom, denom * that.denom)
  def *(that:Rational) =
    new Rational(numer * that.numer, denom * that.denom)
  def /(that:Rational) =
    new Rational(numer * that.denom, denom * that.numer)

  override def toString() =
    "Rational: [" + numer + " / " + denom + "]"
}
```

While the overall structure of Listing 1 is lexically similar to what you've seen in Java code over the last decade, some new elements clearly are at work here. Before picking this definition apart, take a look at the code to exercise the new `Rational` class:

### Listing 2. RunRational

```
class Rational(n:Int, d:Int)
{
  // ... as before
}

object RunRational extends Application
{
  val r1 = new Rational(1, 3)
  val r2 = new Rational(2, 5)
  val r3 = r1 - r2
  val r4 = r1 + r2
  Console.println("r1 = " + r1)
  Console.println("r2 = " + r2)
  Console.println("r3 = r1 - r2 = " + r3)
  Console.println("r4 = r1 + r2 = " + r4)
}
```

What you see in Listing 2 isn't terribly exciting: I create a couple of rational numbers, create two more `Rational`s as the addition and subtraction of the first two, and echo everything to the console. (Note that `Console.println()` comes from the Scala core library, living in `scala.*`, and is implicitly imported into every Scala program, just as `java.lang` is in Java programming.)

## How many ways shall I construct thee?

Now look again at the first line in the `Rational` class definition:

### Listing 3. Scala's default constructor

```
class Rational(n:Int, d:Int)
{
  // ...
}
```

Although you might think you're looking at some kind of generics-like syntax in Listing 3, it's actually the default and preferred constructor for the `Rational` class: `n` and `d` are simply the parameters to that constructor.

Scala's preference for a single constructor makes a certain kind of sense -- most classes end up having a single constructor or a collection of constructors that all "chain" through a single constructor as a convenience. If you wanted to, you could define more constructors on a `Rational` like so:

### Listing 4. A chain of constructors

```
class Rational(n:Int, d:Int)
{
  def this(d:Int) = { this(0, d) }
```

Note that Scala's constructor chain does the usual Java-constructor-chaining thing by calling into the preferred constructor (the `Int, Int` version).

## Details, (implementation) details...

When working with rational numbers, it helps to perform a bit of numerical legerdemain: namely that of finding a common denominator to make certain operations easier. If you want to add 1-over-2 (also known as "one-half") to 2-over-4 (also known as "two-fourths"), the `Rational` class should be smart enough to realize that 2-over-4 is the same as 1-over-2, and convert it accordingly before adding the two together.

This is the purpose of the nested private `gcd()` function and `g` value inside of the `Rational` class. When the constructor is invoked in Scala, the entire body of the class is evaluated, which means `g` will be initialized with the greatest common denominator of `n` and `d`, and then used in turn to set `n` and `d` appropriately.

Looking back at [Listing 1](#), it's also fairly easy to see that I created an overridden `toString` method to return the values of `Rational`, which will be very useful when I start exercising it from the `RunRational` driver code.

Notice the syntax around `toString`, however: the `override` keyword in the front of the definition is required so that Scala can check to make sure that a corresponding definition exists in the base class. This can help prevent subtle bugs created by accidental keyboard slips. (It was this same motivation that led to the creation of the `@Override` annotation in Java 5.) Notice, as well, that the return type is not specified -- it's obvious from the definition of the method body -- and that the returned value isn't explicitly denoted using the `return` keyword, which Java would require. Instead, the last value in the function is considered the return value implicitly. (You can always use `return` keyword if you prefer Java syntax, however.)

## Some core values

Next up are the definitions of `numer` and `denom`, respectively. The syntax involved, offhand, would lead the Java programmer to believe that `numer` and `denom` are public `Int` fields that are initialized to the value of *n-over-g* and *d-over-g*, respectively; but this assumption is incorrect.

Formally, Scala calls `numer` and `denom` *methods without parameters*, which are used to create a quick-and-easy syntax for defining accessors. The `Rational` class still has three private fields, `n`, `d`, and `g`, but they are hidden from the world by default private access in the case of `n` and `d`, and by explicit private access in the case of `g`.

The Java programmer in you is probably asking at this point, "Where are the corresponding "setters" for `n` and `d`?" No such setters exist. Part of the power of Scala is that it encourages developers to create immutable objects by default. Granted, syntax is available to create methods for modifying the internals of `Rational`, but doing so would ruin the implicit thread-safe nature of this class. As a

result, at least for this example, I'm going to leave `Rational` as it is.

Naturally, that raises the question of how one manipulates a `Rational`. Like `java.lang.Strings`, you can't take an existing `Rational` and modify its values, so the only alternative is to create new `Rationals` out of the values of an existing one, or create it from scratch. This brings into focus the next set of four methods: the curiously named `+`, `-`, `*`, and `/` methods.

And no, contrary to what it might look like, this isn't operator-overloading.

## Operator, ring me a number

Remember that in Scala *everything is an object*. In the last [article](#), you saw how that principle applies to the idea that functions themselves are objects, which allows Scala programmers to assign functions to variables, pass functions as object parameters, and so on. An equally important principle is that *everything is a function*; that is to say, in this particular case, there is no distinction between a function named `add` and a function named `+`. In Scala, all operators are functions on a class. They just happen to have, well, funky names.

In the `Rational` class, then, four operations have been defined for rational numbers. These are the canonical, mathematical operations `add`, `subtract`, `multiply`, and `divide`. Each of these is named by its mathematical symbol: `+`, `-`, `*`, and `/`.

Notice, however, that each of these operators works by constructing a new `Rational` object each time. Again, this is very similar to how `java.lang.String` works, and it is the default implementation because it yields thread-safe code. (If no shared state -- and internal state of an object shared across threads is implicitly shared state -- is modified by a thread, then there is no concern over concurrent access to that state.)

### What's new with you?

The *everything is a function* rule has two powerful effects:

The first, as you've already seen, is that functions can be manipulated and stored as objects themselves. This leads to powerful re-use scenarios like the one explored in the [first article](#) in this series.

The second effect is that there is no special distinction between the operators that the Scala-language designers might think to provide and the operators that Scala programmers think *should be* provided. For example, let's assume for a moment that it makes sense to provide an "inversion" operator, which will flip the numerator and denominator and return a new `Rational` (so that `Rational(2,5)` will return `Rational(5,2)`). If you decide that the `~` symbol best represents this concept,

then you can define a new method using that as a name, and it will behave just as any other operator would in Java code, as shown in Listing 5:

### Listing 5. Let's flip

```
val r6 = ~r1
Console.println(r6) // should print [3 / 1], since r1 = [1 / 3]
```

Defining this unary "operator" in Scala is slightly tricky, but it's purely a syntactic nit:

### Listing 6. This is how you flip

```
class Rational(n:Int, d:Int)
{
  // ... as before ...

  def unary_~ : Rational =
    new Rational(denom, numer)
}
```

The tricky part is, of course, the fact that you have to prefix the ~ name with "unary\_" to tell the Scala compiler that it is intended to be a unary operator; therefore, the syntax will be "flipped" from the traditional reference-then-method syntax common in most object languages.

Note that this combines with the "everything is an object" rule to create some powerful -- but easy-to-explain -- code opportunities:

### Listing 7. Add it up

```
1 + 2 + 3 // same as 1.+(2.+(3))
r1 + r2 + r3 // same as r1.+(r2.+(r3))
```

Naturally, the Scala compiler "does the right thing" for the straight integer addition examples, but syntactically it's all the same. This means that you can develop types that are no different from the "built-in" types that come as part of the Scala language.

The Scala compiler will even try to infer some meaning out of the "operators" that have some predetermined meaning, such as the += operator. Note how the following code just does what it should, despite the fact that the Rational class doesn't have an explicit definition for +=:

### Listing 8. Scala infers

```
var r5 = new Rational(3,4)
r5 += r1
Console.println(r5)
```

When printed, `r5` has the value `[13 / 12]`, which is exactly what it should be.

## Scala under the hood

Remember that Scala compiles to Java bytecode, meaning that it runs on the JVM. If you need proof, look no further than the fact that the compiler is producing `.class` files that begin with `0xCAFEBAFE`, just like `javac` does. Also note what happens if you fire up the Java bytecode disassembler that comes with the JDK (`javap`) and point it at the generated `Rational` class, as shown in Listing 9:

### Listing 9. Classes compiled from `rational.scala`

```
C:\Projects\scala-classes\code>javap -private -classpath classes Rational
Compiled from "rational.scala"
public class Rational extends java.lang.Object implements scala.ScalaObject{
    private int denom;
    private int numer;
    private int g;
    public Rational(int, int);
    public Rational unary_~();
    public java.lang.String toString();
    public Rational $div(Rational);
    public Rational $times(Rational);
    public Rational $minus(Rational);
    public Rational $plus(Rational);
    public int denom();
    public int numer();
    private int g();
    private int gcd(int, int);
    public Rational(int);
    public int $tag();
}
```

```
C:\Projects\scala-classes\code>
```

The "operators" defined in the Scala class transmogrify into method calls in the best tradition of Java programming, though they do seem to be based on funny names. Two constructors are defined on the class: one taking an `int` and one taking a pair of `ints`. And, if you happen to be at all concerned that the use of the upper-case `Int` type is somehow a `java.lang.Integer` in disguise, note that the Scala compiler is smart enough to transform them into regular Java primitive `ints` in the class definition.

### Testing, testing, 1-2-3...

It is a well-known meme that good programmers write code, and great programmers write tests; thus far, I have been lax in exercising this rule for my Scala code, so let's see what happens when you put this `Rational` class inside of a traditional JUnit test suite, as shown in Listing 10:

## Listing 10. RationalTest.java

```
import org.junit.*;
import static org.junit.Assert.*;

public class RationalTest
{
    @Test public void test2ArgRationalConstructor()
    {
        Rational r = new Rational(2, 5);

        assertTrue(r.numer() == 2);
        assertTrue(r.denom() == 5);
    }

    @Test public void test1ArgRationalConstructor()
    {
        Rational r = new Rational(5);

        assertTrue(r.numer() == 0);
        assertTrue(r.denom() == 1);
        // 1 because of gcd() invocation during construction;
        // 0-over-5 is the same as 0-over-1
    }

    @Test public void testAddRationals()
    {
        Rational r1 = new Rational(2, 5);
        Rational r2 = new Rational(1, 3);

        Rational r3 = (Rational) reflectInvoke(r1, "$plus", r2); //r1.$plus(r2);

        assertTrue(r3.numer() == 11);
        assertTrue(r3.denom() == 15);
    }

    // ... some details omitted
}
```

### SUnit

A Scala-based unit test suite already exists, going by the name of SUnit. If you used SUnit for the test shown in Listing 10, you wouldn't need the Reflection-based workaround. The Scala-based unit test code would be compiled against the Scala class, so the compiler would be able to make the symbols line up. Some developers actually find it more attractive to write unit tests in Scala, exercising POJOs, instead of the other way around.

SUnit is part of the standard Scala distribution, in the `scala.testing` package. (For more information on SUnit, see [Resources](#).)

Aside from confirming that the `Rational` class behaves, well, rationally, the above test suite also proves that it is possible to call Scala code from Java code (albeit with a little bit of an impedance mismatch when it comes to the operators). The cool thing about this, of course, is that it lets you try out Scala slowly, by migrating Java classes over to Scala classes without ever having to change the tests that back them.

The only weirdness you might notice in the test code has to do with operator invocation, in this case, the `+` method on the `Rational` class. Looking back at the `javap` output, Scala has obviously translated the `+` function into the JVM method `$plus`, but the Java Language Specification does not allow the `$` character in identifiers (which is why it's used in nested and anonymous nested class names).

In order to invoke those methods, you either have to write the tests in Groovy or JRuby (or some other language that doesn't pose a restriction on the `$` character), or you can write a little bit of `Reflection` code to invoke it. I go with the latter approach, which isn't all that interesting from a Scala perspective, but the result is included in this article's code bundle, should you be curious. (See [Download](#).)

Note that workarounds like these are only necessary for function names that aren't also legitimate Java identifiers.

## A "better" Java

Back when I was first learning C++, Bjarne Stroustrup suggested that one way to learn C++ was to see it as "a better C" (see [Resources](#)). In some ways, Java developers today might come to see Scala as a "better Java," because it provides a more terse and succinct way of writing traditional Java POJOs. Consider the traditional `Person` POJO shown in Listing 11:

### Listing 11. `JavaPerson.java` (original POJO)

```
public class JavaPerson
{
    public JavaPerson(String firstName, String lastName, int age)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName()
    {
        return this.firstName;
    }
    public void setFirstName(String value)
    {
        this.firstName = value;
    }

    public String getLastName()
    {
        return this.lastName;
    }
    public void setLastName(String value)
    {
        this.lastName = value;
    }

    public int getAge()
    {
```

```

        return this.age;
    }
    public void setAge(int value)
    {
        this.age = value;
    }

    public String toString()
    {
        return "[Person: firstName" + firstName + " lastName:" + lastName +
            " age:" + age + " ]";
    }

    private String firstName;
    private String lastName;
    private int age;
}

```

Now consider its equivalent written in Scala:

### Listing 12. person.scala (threadsafe POJO)

```

class Person(firstName:String, lastName:String, age:Int)
{
    def getFirstName = firstName
    def getLastName = lastName
    def getAge = age

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

```

It isn't a complete drop-in replacement, given that the original `Person` had some mutable setters. But considering the original `Person` also had no synchronization code around those mutable setters, the Scala version is safer to use. Also, if the goal is to truly reduce the number of lines of code in `Person`, you could remove the `getFoo` property methods entirely because Scala will generate accessor methods around each of the constructor parameters -- `firstName()` returns a `String`, `lastName()` returns a `String`, and `age()` returns an `int`).

Even if the need for those mutable setter methods is undeniable, the Scala version is still simpler, as you can see in Listing 13:

### Listing 13. person.scala (full POJO)

```

class Person(var firstName:String, var lastName:String, var age:Int)
{
    def getFirstName = firstName
    def getLastName = lastName
    def getAge = age

    def setFirstName(value:String):Unit = firstName = value
    def setLastName(value:String) = lastName = value
    def setAge(value:Int) = age = value
}

```

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

As an aside, notice the introduction of the `var` keyword on the constructor parameters. Without going into too much detail, `var` tells the compiler that the value is mutable. As a result, Scala generates both accessor (`String firstName(void)`) and mutator (`void firstName_$eq(String)`) methods. It then becomes easy to create `setFoo` property mutator methods that use the generated mutator methods under the hood.

## Conclusion

### Share this...



Digg  
this  
story



Post  
to  
del.icio.us



Slashdot  
it!

Scala is an attempt to incorporate functional concepts and terseness without losing the richness of the object paradigm. As you've perhaps begun to see in this series, Scala also corrects some of the egregious (in hindsight) syntactic problems found in the Java language.

This second article in the *Busy Java developer's guide to Scala* series has focused on Scala's object facilities, which let you start using Scala without having to dive too deeply into the functional pool. Based on what you've learned so far, you can already start using Scala to reduce your programming workload. Among other things, you can use Scala to produce the very same POJOs needed for other programming environments, such as Spring or Hibernate.

Hold on to your diving caps and scuba gear, however, because next month's article will mark the beginning of our descent into the deep end of the functional pool.



## Downloads

Description	Name	Size	Download method
Sample Scala code for this article	code.zip	2.6MB	<a href="#">HTTP</a>

[Information about download methods](#)

# Resources

## Learn

- ["The busy Java developer's guide to Scala: Functional programming for the object oriented"](#) (Ted Neward, developerWorks, January 2008): In the first article in this series, get an overview of Scala and understand its functional approach to concurrency, among other things.
- ["Functional programming in the Java language"](#) (Abhijit Belapurkar, developerWorks, July 2004): Learn about 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 (in PDF).
- [Programming in Scala](#) (Martin Odersky, Lex Spoon, and Bill Venners, Artima, December 2007): The first book-length introduction to Scala.
- [Bjarne Stroustrup](#): Designed and implemented C++, which he has described as "a better C" (see Stroustrup's [C++ Programming Language](#) page for this specific reference).
- The [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!
- [SUnit](#): Part of the standard Scala distribution, in the *scala.testing* package.

## 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.