Java.next: Common ground in Groovy, Scala, and Clojure, Part 1

Explore how these next-generation JVM languages handle operator overloading

Neal Ford
Director / Software Architect / Meme Wrangler
ThoughtWorks Inc.

Skill Level: Intermediate

Date: 12 Mar 2013
(Updated 16 Apr 2013)

The Java.next languages (Groovy, Scala, and Clojure) have more commonalities than differences, converging toward common ground in many of their features and conveniences. This installment explores how they each address a longstanding deficiency in the Java™ language — the inability to overload operators. It also discusses the related concepts of associativity and precedence.
16 Apr 2013 - *Added links to "The Java.next languages" and "Common ground in Groovy, Scala, and Clojure, Part 2" in Resources.*

View more content in this series

About this series

The Java legacy will be the platform, not the language. More than 200 languages run on the JVM, each bringing interesting new capabilities beyond the capabilities of the Java language. This series explores three next-generation JVM languages — Groovy, Scala, and Clojure — comparing and contrasting new capabilities and paradigms. The series aims to give Java developers a glimpse into their own near future — and help them make educated choices about the time they devote to new-language learning.

Good ideas in programming languages persist and spread to other languages, permeating them over time. Thus, unsurprisingly, the Java.next languages — Groovy, Scala, and Clojure — share many common features. In this and upcoming *Java.next* installments, I explore how the convergence of function manifests in the syntax of each. I start with a feature — the ability to overload operators — that makes up for a longstanding deficiency in the Java language.

Operator overloading

If you ever dabble with the Java `BigDecimal` class, you probably see code similar to Listing 1:

Listing 1. Lackluster `BigDecimal` support in Java code

```
BigDecimal op1 = new BigDecimal(1e12);
BigDecimal op2 = new BigDecimal(2.2e9);
// (op1 + (op2 * 2)) / (op1/(op1 + (op2 * 1.5e2))
BigDecimal lhs = op1.add(op2.multiply(BigDecimal.valueOf(2)));
BigDecimal rhs = op1.divide(
        op1.add(op2.multiply(BigDecimal.valueOf(1.5e2))),
            RoundingMode.HALF_UP);
BigDecimal result = lhs.divide(rhs);
System.out.println(String.format("%,.2f", result));
```

In Listing 1, I try to fulfill the formula that I listed as a comment. In Java programming, I cannot overload mathematical operators, forcing me to fall back on method invocations. Static imports can help, but a clear need exists for appropriate operator overloading for selected contexts. The original Java engineers purposely omitted operator overloading from the language, feeling that it added too much complexity. But experience shows that the complexity forced on developers by the lack of this feature outweighs the potential abuse opportunities.

In slightly different ways, all three of the Java.next languages implement operator overloading.

Scala's operators

Scala allows operator overloading by discarding the distinction between operators and methods. Operators are merely methods with special names. To override the multiplication operator, for example, you override the `*` method. [`*` is a valid method name, which is one reason that Scala uses the underscore (`_`) character for imports rather than the Java asterisk (`*`) character.]

I use *complex numbers* to illustrate overloading. Complex numbers are a mathematical notation that includes both a real part and imaginary part, typically written as, for example, `3 + 4i` (see Resources). Complex numbers are common in many scientific fields, including engineering, physics, electromagnetism, and chaos theory. Listing 2 shows the Scala implementation of complex numbers:

Listing 2. Scala complex numbers

```
final class Complex(val real: Int, val imaginary: Int) {
  require (real != 0 || imaginary != 0)

  def +(operand: Complex) =
      new Complex(real + operand.real, imaginary + operand.imaginary)

  def +(operand: Int) =
    new Complex(real + operand, imaginary)

  def -(operand: Complex) =
```

```
    new Complex(real - operand.real, imaginary - operand.imaginary)

  def -(operand: Int) =
    new Complex(real - operand, imaginary)

  def *(operand: Complex) =
      new Complex(real * operand.real - imaginary * operand.imaginary,
          real * operand.imaginary + imaginary * operand.real)

  override def toString() =
      real + (if (imaginary < 0) "" else "+") + imaginary + "i"

  override def equals(that: Any) = that match {
    case other : Complex => (real == other.real) && (imaginary == other.imaginary)
    case _ => false
  }

  override def hashCode(): Int =
    41 * ((41 + real) + imaginary)
}
```

`equals()` and the `match` keyword

Another interesting feature in Listing 2 is the use of pattern matching within the `equals()` method. Although type casting is possible in Scala, matching against type is more common. The `that` parameter is declared as `Any` — Scala's top of the inheritance hierarchy. The method's body consists of the `match` call, which checks the value of the fields when the passed type matches and defaults to `false` otherwise.

Scala simplifies much of the verbosity in the Java language by collapsing needless scaffolding. For example, in Listing 2, the constructor parameters and fields in the class appear with the class definition. The body of the class acts as the constructor in this case, so the call to the `require()` method validates the presence of values as the first instantiation action. Because Scala automatically provides fields, the remainder of the class contains method definitions. For the `+`, `-`, and `*` operators, I declare eponymous methods that accept `Complex` numbers as parameters. Multiplication of complex numbers is less straightforward than addition and subtraction. The overloaded `*` method in Listing 2 implements the formula:

```
(x + yi)(u + vi) = (xu - yv) + (xv + yu)i
```

The `toString()` method in Listing 2 exemplifies another bit of common ground among the Java.next languages: use of *expressions* rather than *statements*. In the `toString()` method, I must supply the plus (`+`) sign if the imaginary part is positive, but the imaginary part's implicit minus sign suffices otherwise. In Scala, `if` is an expression rather than a statement, eliminating the need for the Java ternary operator (`?:`).

In practice, the added `+`, `-`, and `*` methods are indistinguishable from standard operators, as shown in the unit tests in Listing 3:

Listing 3. Exercising Scala complex numbers

```
class ComplexTest extends FunSuite {
```

```
  test("addition") {
    val c1 = new Complex(1, 3)
    val c2 = new Complex(4, 5)
    assert(c1 + c2 === new Complex(1+4, 3+5))
  }

  test("subtraction") {
    val c1 = new Complex(1, 3)
    val c2 = new Complex(4, 5)
    assert(c1 - c2 === new Complex(1-4, 3-5))
  }

  test("multiplication") {
    val c1 = new Complex(1, 3)
    val c2 = new Complex(4, 5)
    assert(c1 * c2 === new Complex(
        c1.real * c2.real - c1.imaginary * c2.imaginary,
        c1.real * c2.imaginary + c1.imaginary * c2.real))
  }
}
```

The tests in Listing 3 fail to reveal an interesting inconsistency. I expose and solve that problem momentarily, when I discuss *associativity*. First, though, a word about overloading in Groovy and Clojure.

Groovy's mappings

Groovy overloads any Java operators by providing mapping methods that you can override. (For example, to override the `+` operator, you override the `plus()` method on the `Integer` class.) I cover Groovy's operator-overloading details, with the same complex-number example, in "Functional design patterns, Part 3," an installment of my *Functional thinking* series about extensibility in functional languages.

In Groovy, you cannot create new operators (although you can certainly create new methods). Some frameworks (such as the Spock testing framework; see Resources) overload esoteric but existing operators such as `>>>`. Both Scala and Clojure treat operators and methods more uniformly, although in distinctive ways.

Groovy also introduced several handy new operators, such as `?.` — the *safe navigation* operator, which ensures that none of the callers is null — and the *Elvis* operator (`?:`), a shortening of the Java ternary operator that is useful for easily supplying default values. Groovy has no extension methods for its new operators, preventing developers from overloading them. And it is not clear why developers would want to overload them: The typical rationale for operator overloading lies with using prior experience with an operator to make your code more readable. You are unlikely to develop experience with these operators outside of Groovy. Operator overloading becomes dangerous if you use the operators for convenience but harm readability.

Clojure's operators

As in Scala, operators in Clojure are merely methods with symbolic names. Thus, you can trivially create a `+` method, for example, for your custom type. However,

to override operators properly in Clojure, you must understand *protocols* and a technique for generating a set of methods from a common core. I save that discussion for a later installment.

Associativity

Operator *associativity* refers to whether the operator is a method on the left or right side of the equation. Scala uses white space differently from most other languages, in that essentially any Scala method can act as an operator. For example, the expression `x + y` is really the method invocation `x.+(y)`, as in the Scala REPL (interpreter) session in Listing 4:

Listing 4. White space translation in Scala

```
scala> val sum1 = x.+(y)
sum1: Int = 22

scala> val sum2 = (12).+(10)
sum2: Int = 22
```

You can see in Listing 4 that white space translation also works for constants. You can treat all the methods in Scala as operators if you like. For example, the `String` class has an `indexOf()` method that returns the index position within the string of the character that is passed as the argument. In Scala, you can call it traditionally through `s.indexOf('a')` or as an operator — as in `s indexOf 'a'`. (This particular method is interesting because it has an overloaded version that accepts an extra parameter to specify the index position where the search begins. You can still call it using operator notation, but you must put the parameters in parentheses, as in `s indexOf('a', 3)`).

Groovy follows the Java associativity conventions, so the rules for a specific operator are defined by the language. Clojure is free of any concerns about associativity; its Lisp syntax does not rely on associativity because all statements are unambiguous.

Because one of Scala's goals is to allow developers to use anything as an operator, it cannot rely on arbitrary associativity rules. How can the language allow ad hoc operators yet still establish rules? Scala solves this problem in an innovative way that enables maximum developer freedom — by using a naming convention for operators. By default, operators in Scala left-associate: The expression resolves to a method call on the left operand, meaning that the expression `x + y` resolves to `x.+(y)`, for example. However, if the method name ends with `:`, the operator right-associates. For example, the invocation `i +: j` translates to `j.+:(i)`.

Associativity explains why the tests in Listing 3 fail to tell the entire story. In the Scala `Complex` definition in Listing 2, I implement versions of `+` and `-` operators that accept both `Complex` and `Int` parameter types. This type flexibility allows complex numbers to interoperate with regular integers (which are complex numbers with a zero imaginary part). Listing 5 illustrates interoperability in the unit tests:

Listing 5. Tests for mixed types

```
test("mixed addition from Complex") {
  val c1 = new Complex(1, 3)
  assert(new Complex(7, 3) == c1 + 6)
}

test("mixed subtraction from Complex") {
  val c1 = new Complex(10, 3)
  assert(new Complex(5, 3) == c1 - 5)
}
```

Both tests in Listing 5 pass with no problems — the `Int` version of the operator method is started. However, if I try the following test, it fails:

```
test("mixed subtraction from Int") {
  val c1 = new Complex(10, 3)
  assert(new Complex(15, 3) == 5 + c1)
}
```

The subtle difference between the two tests concerns associativity. Remember, Scala calls the method of the left operator in this case, meaning that it is trying to start a method that is defined for `Int` that "knows" how to handle a complex number.

To solve this problem, I define an *implicit cast* between `Int` and `Complex`. There are several ways to show this conversion, which I cover in more detail in later installments. In this instance, I create a *companion object* — a place for methods that would be declared `static` in the Java language— called `Complex`:

```
final object Complex {
  implicit def intToComplex(x: Int) = new Complex(x, 0)
}
```

This definition includes a single method, accepting an `Int` and returning it as a `Complex`. By placing this declaration in the same source file as the `Complex` class, I enable the implicit conversion by importing this method within my test case through the `import nealford.javaNext.complexnumbers.Complex.intToComplex` command. When I have the conversion in scope, the test case passes because the test knows how to handle the method invocation that is made through the operator.

Precedence

Operator *precedence* (or *order of operations*) refers to a language's rules for the order in which operations take place in potentially ambiguous situations. Groovy relies on the Java precedence rules for common operators and defines its own rules for its custom operators. Clojure does not have or need precedence rules; because all code is written in fully parenthesized form, the ambiguity inherent in infix notation never appears.

Scala uses the first character of the operator name to determine order of operations, in this precedence hierarchy:

- All other special characters
- `* / %`
- `+ -`
- `:`
- `= !`
- `< >`
- `&`
- `^`
- `|`
- All letters
- All assignment operators

Operators that start with higher-ranked characters have higher precedence. For example, the expression `x *** y ||| z` would resolve to `(x.***(y)).|||(z)`. The lone exception to this rule concerns assignment statements, or any operator that ends with an equals sign (`=`), which automatically have the lowest precedence.

Conclusion

A shared goal of the Java.next languages is to ease cumbersome restrictions that affect the Java language. Operator overloading is a great example of how each language approaches this problem. All three languages allow operator overloading, varying in how they implement that power. The subtle nuances of how to handle problems like associativity and precedence show how connected language parts are to one another. One of Clojure's interesting aspects is that its syntax — because every expression is inherently parenthesized — eliminates ambiguity in precedence and associativity.

In the next installment, I explore just how deep the "Everything is an object" philosophy penetrates in the Java.next languages.

Resources

Learn

- Java.next: The Java.next languages (Neal Ford, developerWorks, January 2013): Explore the similarities and differences of three next-generation JVM languages (Groovy, Scala, and Clojure) in this overview of the Java.next languages and their benefits.
- Java.next: Common ground in Groovy, Scala, and Clojure, Part 2 (Neal Ford, developerWorks, April 2013): Reduce boilerplate and complexity with the syntactic conveniences offered by Java.next languages.
- Complex number: Learn about complex numbers (used for the examples in the article) at Wikipedia.
- Groovy: Groovy is a dynamic language for the JVM. Read the Groovy Operator overloading documentation.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Explore alternative languages for the Java platform: Follow this knowledge path to investigate developerWorks content about various alternative JVM languages.
- *Language designer's notebook*: In this developerWorks series, Java Language Architect Brian Goetz explores some language design issues that present challenges for the evolution of the Java language in Java SE 7, Java SE 8, and beyond.
- *Functional thinking*: Explore functional programming in Neal Ford's column series on developerWorks.
- More articles by this author (Neal Ford, developerWorks, June 2005-current): Learn about Groovy, Scala, Clojure, functional programming, architecture, design, Ruby, Eclipse, and other Java-related technologies.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Spock: Spock is a testing and specification framework for Java and Groovy applications.
- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Neal Ford

Neal Ford is Director, Software Architect, and Meme Wrangler at **Thought**Works, a global IT consultancy. He is also the designer and developer of applications, instructional materials, magazine articles, courseware, and video/DVD presentations, and he is the author or editor of books spanning a variety of technologies, including the most recent *Presentation Patterns*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his website.