The busy Java developer's guide to Scala: Building a calculator, Part 1

Scala's case classes and pattern matching

Skill Level: Introductory

Ted Neward (ted@tedneward.com)
Principal
Neward & Associates

26 Aug 2008

Domain-specific languages have become a hot topic; much of the buzz around functional languages is their applicability to build such languages. In this, the eighth article in *The busy Java™ developer's guide to Scala*, Ted Neward starts building a simple calculator DSL that demonstrates the power of functional languages for building "external" DSLs. Toward that end, he explores a new feature of Scala, *case classes*, and revisits an old functional friend, *pattern matching*.

After last month's article, driven by reader feedback, I got some additional complaints/comments that the examples in the series thus far had been a bit on the trivial side. While it's certainly reasonable to use trivial examples during the early stages of learning a new language, it's also perfectly reasonable for readers to demand something a bit more "real world" to demonstrate the deep scope and the power of the language and its advantages. So, this month we begin a two-part exercise to build a domain-specific language (DSL) — in this case, a small calculator language.

About this series

Ted Neward dives into the Scala programming language and takes you along with him. In this exciting 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 code — and therein lies much of Scala's charm! After all, if Java code could do it, why bother learning Scala?

Domain-specific languages

If you don't have the ability (or time) to crawl out from underneath the rock your project managers keep you under, I'll be quick to the point: *A domain-specific language is nothing more than an attempt to (once again) put the power of an application in exactly the right place where it belongs — in the hands of its users.*

By defining a new textual language that users can understand and use directly, programmers effectively remove themselves from the never-ending cycle of UI requests and enhancements and can let users create scripts and other tools that allow them to create emergent behavior in the applications they build. While this example may be tempting fate (and a few hate e-mails), the canonical example of a wildly successful DSL is the Microsoft® Office Excel "language" used to express the various calculations and contents of the cells in the spreadsheet. Some might even go so far as to suggest that SQL itself is a DSL, this time a language aimed specifically at interacting with the relational database. (Imagine if programmers had to fetch data out of Oracle via traditional API `read()`/`write()` calls — eeesh.)

The DSL being built here is a simple calculator language, designed to take mathematical expressions and evaluate them. In essence, the goal here is to create a small language that allows users to type in relatively simple algebraic expressions that this code will evaluate and produce a result. For simplicity's sake, the language won't support a lot of the features a more full-featured calculator should, but I don't want to keep it entirely in the realm of pedagogy either — the language should be extensible enough that readers can use what's here as the core of a more powerful language without having to rebuild it from the ground up. This means the language should be easily extensible and maintain encapsulation as much as possible without creating significant hurdles to using it.

More on DSLs

The subject of DSLs is a large one; it is certainly much richer and more extensive than the one-paragraph introduction in this article can even begin to explain. Readers looking for more information on the subject of DSLs are encouraged to check out Martin Fowler's "book-in-progress" listed at the end of this article; in particular, pay attention to the discussion between "internal" and "external" DSLs. By virtue of its flexible syntax and functional nature, Scala works as a powerful language for building both internal or external DSLs.

In other words, the goal is (eventually) to allow clients to write code along the lines of:

Listing 1. Calculator DSL: The goal

```
// This is Java using the Calculator
String s = "((5 * 10) + 7)";
double result =
com.tedneward.calcdsl.Calculator.evaluate(s);
System.out.println("We got " + result); // Should
be 57
```

We won't get there all in one article, but we can get part of the way this time and finish it off next time.

From an implementation and design perspective, it would be tempting to build a string-based parser to begin with and start building things along the lines of some kind of "pick each character and evaluate as we go" parser, but while this might work for simpler languages, it doesn't really scale well. Given that easy extensibility is a goal for this language, it behooves us to take a moment and think about the design of the language before diving into its implementation.

For those versed in rudimentary compiler theory, you know that the basic operation of a language processor (which includes both interpreters and compilers) consists of at least two basic phases:

- A parser to take the incoming text and turn it into an Abstract Syntax Tree (AST).
- Either a code generator (in the case of a compiler) to take the AST and generate the desired bytecode from it, or an evaluator (in the case of an interpreter) to take the AST and execute what it finds there.

The reason for this distinction becomes more apparent when we realize that having an AST offers opportunities to do some optimization on the resulting tree; in the case of our calculator, we might go through the expressions and look for places where we can cut out whole swaths of expressions such as those where one of the operands in a multiplication expression is "0" (which yields "0" regardless of what the other operand is).

The first item on our plate is to define this AST for the calculator language. Fortunately, Scala has *case classes*: classes that are data-rich and encapsulation-thin and have some useful features that make them quite suitable for building an AST.

Case classes

Before getting too deep into the AST definition, let me provide a quick overview of what case classes are. A case class is a convenience mechanism for the Scala programmer to create a class with some assumed defaults in place. For example, when writing the following:

Listing 2. Casing the person

```
case class Person(first:String, last:String, age:Int)
{

}
```

the Scala compiler does more than just generate the expected constructor that we've come to expect of it — the Scala compiler also generates a common-sense `equals()`, `toString()`, and `hashCode()` implementation. In fact, this kind of case class is so common (that is, one with no additional members) that the curly-braces after the case class declaration are optional:

Listing 3. World's shortest class listing

```
case class Person(first:String, last:String, age:Int)
```

This is easily verifiable via our old friend, `javap`:

Listing 4. Holy code generator, Batman!

```
C:\Projects\Exploration\Scala>javap Person
Compiled from "case.scala"
public class Person extends java.lang.Object implements scala.ScalaObject,scala.
Product,java.io.Serializable{
    public Person(java.lang.String, java.lang.String, int);
    public java.lang.Object productElement(int);
    public int productArity();
    public java.lang.String productPrefix();
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public int hashCode();
    public int $tag();
    public int age();
    public java.lang.String last();
    public java.lang.String first();
}
```

As you can see, lots of things happened with a case class that don't normally happen for a traditional class. This is because case classes are designed to be used in conjunction with Scala's *pattern matching* (which we briefly examined a few articles ago in "Collection types").

Using case classes is a bit different from traditional classes because they're generally not constructed via the traditional "new" syntax; in fact, they're generally created via a factory method whose name is the same as the class:

Listing 5. No new?!?

```
object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
  }
}
```

In and of itself, case classes may not seem more interesting, or different, than a traditional class, but an important distinction takes place when using them. The generated code for the case class becomes more interested in bitwise equality than in referential equality, so the following code has some interesting surprises in store for the Java programmer:

Listing 6. This is not your father's class

```
object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
    val ted2 = Person("Ted", "Neward", 37)
    val amanda = Person("Amanda", "Laucher", 27)

    System.out.println("ted == amanda: " +
      (if (ted == amanda) "Yes" else "No"))
    System.out.println("ted == ted: " +
      (if (ted == ted) "Yes" else "No"))
    System.out.println("ted == ted2: " +
      (if (ted == ted2) "Yes" else "No"))
  }
}

/*
C:\Projects\Exploration\Scala>scala App
ted == amanda: No
ted == ted: Yes
ted == ted2: Yes
 */
```

Where the real value of case classes come into play is in pattern matching, which readers of this series will recall (from the second article in this series on the various control constructs in Scala) as being something like Java's "switch/case" but with vastly deeper capabilities and functionality. Not only can pattern matching examine the value of the matched construct for matching values, but the value can be matched against partial wildcards (think something along the lines of a partial "default"), cases can include guards to test the match, values from the match criteria can be bound into local variables, and even the type itself from the match criteria can be matched against.

With case classes, pattern matching begins to take on whole new levels of power, as shown in Listing 7:

Listing 7. This is not your father's switch, either

```
case class Person(first:String, last:String, age:Int);

object App
{
  def main(args : Array[String]) : Unit =
  {
    val ted = Person("Ted", "Neward", 37)
    val amanda = Person("Amanda", "Laucher", 27)

    System.out.println(process(ted))
    System.out.println(process(amanda))
  }
  def process(p : Person) =
  {
    "Processing " + p + " reveals that" +
    (p match
    {
      case Person(_, _, a) if a > 30 =>
        " they're certainly old."
      case Person(_, "Neward", _) =>
        " they come from good genes...."
      case Person(first, last, ageInYears) if ageInYears > 17 =>
        first + " " + last + " is " + ageInYears + " years old."
      case _ =>
        " I have no idea what to do with this person"
    })
  }
}

/*
C:\Projects\Exploration\Scala>scala App
Processing Person(Ted,Neward,37) reveals that they're certainly old.
Processing Person(Amanda,Laucher,27) reveals that Amanda Laucher is 27 years old
.
 */
```

There's a bunch of things happening all at the same time in Listing 7. Let's walk it through slowly, then get back to the calculator and see how to apply them.

First, the entire `match` expression is wrapped in parentheses: This isn't a necessary part of the pattern-matching syntax, but is happening because I'm concatenating the result of the pattern-match expression (remember that everything in a functional language is an expression) against the prefix string in front of it.

Second, the first `case` expression has two wildcards in it (the underscore character is the wildcard), meaning this match will take any value for those two fields in the matched `Person`, but it introduces a local variable a to which the value in `p.age` will be bound. This case only succeeds if the accompanying guard expression (the `if` expression that follows it) succeeds, which will only happen for the first `Person` but not the second. The second `case` expression uses a wildcard for the `firstName` part of `Person` but matches against the constant string `Neward` in the `lastName` part and wildcards against the `age` part.

Because the first `Person` already matches against the previous `case` and because the second `Person` doesn't have a last name of `Neward`, this match won't get fired for either `Person`. (However, a `Person("Michael", "Neward", 15)` would because it would fail the guard clause on the first case and thus fall through to the second case.)

The third case demonstrates a common use of pattern matching sometimes called *extraction* in which the values in the matched object p are extracted into local variables (first, last, and `ageInYears`) for use inside the case block. The final case expression is the general-case default which will be fired only if none of the other case expressions are successful.

Armed with that admittedly brief rundown of case classes and pattern matching, let's return to the task of creating the calculator AST.

Calculator AST

To start with, a calculator's AST should probably have a common base type because mathematical expressions are often made up of sub-expressions; the easiest way to see this is via the example "5 + (2 * 10)" where the sub-expression "(2 * 10)" will be the right-hand operand of the "+" operation.

In fact, this expression gives us three of the AST types:

- The base Expression
- A Number type to hold constant values
- A BinaryOperator to hold an operation and two operands

After a moment's thought, also remember that math allows for such unary operators as the negation operator (the minus) to flip a value from positive to negative, so we can introduce the following basic AST:

Listing 8. Calculator AST (src/calc.scala)

```
package com.tedneward.calcdsl
{
  private[calcdsl] abstract class Expr
  private[calcdsl]  case class Number(value : Double) extends Expr
  private[calcdsl]  case class UnaryOp(operator : String, arg : Expr) extends Expr
  private[calcdsl]  case class BinaryOp(operator : String, left : Expr, right : Expr)
   extends Expr
}
```

Note the package declaration to put this all into a package (`com.tedneward.calcdsl`) and the access modifier declarations in front of each of the classes to indicate that they are accessible to other members of this package or sub-packages. You do this because you want to have a series of JUnit tests that exercise this code; the actual clients of the Calculator shouldn't have to see the AST. So write the unit test to be a subpackage of `com.tedneward.calcdsl`:

Listing 9. Calculator tests (testsrc/calctest.scala)

```
package com.tedneward.calcdsl.test
{
  class CalcTest
  {
    import org.junit._, Assert._

    @Test def ASTTest =
    {
      val n1 = Number(5)

      assertEquals(5, n1.value)
    }

    @Test def equalityTest =
    {
      val binop = BinaryOp("+", Number(5), Number(10))

      assertEquals(Number(5), binop.left)
      assertEquals(Number(10), binop.right)
      assertEquals("+", binop.operator)
    }
  }
}
```

So far, so good. We have an AST.

Think about that for a second: With four lines of Scala code, we've built a hierarchy of types for representing an arbitrarily-deep collection of mathematical expressions (simple mathematical expressions certainly, but still useful). This isn't functional so much as it is Scala's gestures toward making object programming easier and more expressive. (The functional bits will come later, no worries.)

Next, we need an evaluation function that will take the AST and evaluate it to a numerical value. This is pretty straightforward to write, given the power of pattern matching:

Listing 10. Calculator (src/calc.scala)

```
package com.tedneward.calcdsl
{
  // ...

  object Calc
  {
    def evaluate(e : Expr) : Double =
    {
      e match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
  }
}
```

Note that `evaluate()` returns a `Double`, which means that each case in the pattern match is going to have to evaluate into a `Double` value. This isn't difficult: Numbers simply return their contained value. But for the rest of the cases, the two kinds of operators, we have to evaluate the operands as well before performing the necessary operation (negation, addition, subtraction, etc.) on those operands. As is common in functional languages, recursion steps in and we simply call `evaluate()` on each operand before performing the operation on the whole.

This idea of performing the evaluation *outside* the various operators themselves is going to strike most dyed-in-the-wool object-oriented programmers as fundamentally wrong — this is clearly a gross violation of the principles of encapsulation and polymorphism. Quite frankly, it's not even worth arguing; it most certainly *is* a violation of encapsulation, at least in the traditional sense.

The larger question here to consider is, precisely what are we trying to encapsulate the code from? Remember that the AST classes aren't visible outside of this package at all and that clients will (eventually) simply pass in a String representation of the expression they want evaluated. It's only the unit tests that are working with the AST case classes directly.

This isn't to suggest that all encapsulation is dead or passe, though. In fact, it's exactly the opposite: It's an attempt to suggest that there are other design approaches that can work well beyond those we're familiar with from the object world. Don't forget that Scala is a fusion of both objects and functions; if there were situations where `Expr` needs additional behavior attached to it and its subclasses (such as pretty-printing `toString` methods, for example), these methods can be added to `Expr` with little work. The combination of the functional and object-oriented worlds is an additive one and neither functional nor object programmers should ignore the other half's design approaches or how the two can combine to produce some interesting effects.

From a design perspective, some other choices are questionable; for example, the use of Strings to hold the operators opens up the possibility of small typos leading to erroneous results. In production code, this could (and perhaps should) be an enumeration instead of Strings, but leaving them as Strings means that we could potentially "open up" the operators to allow calling out to more complex functions (such as abs, sin, cos, tan, and so on) or even as far as to user-defined functions; these are things the enumeration-based approach would have a harder time supporting.

As with all design and implementation decisions, there is no one right way, just consequences. *Caveat emptor.*

There is one interesting implementation trick you can apply here, though. Certain mathematical expressions can be simplified, thus (potentially) optimizing the evaluation of the expression (and not coincidentally, demonstrating the usefulness of an AST):

- Anything adding "0" can be simplified down to the non-zero operand.
- Anything multiplying by "1" can be simplified down to the non-zero operand.
- Anything multiplying by "0" can be simplified down to zero.

And so on. So we introduce a pre-evaluation step, called `simplify()` to perform these exact simplifications:

Listing 11. Calculator (src/calc.scala)

```
    def simplify(e : Expr) : Expr =
    {
      e match {
        // Double negation returns the original value
        case UnaryOp("-", UnaryOp("-", x)) => x
        // Positive returns the original value
        case UnaryOp("+", x) => x
        // Multiplying x by 1 returns the original value
        case BinaryOp("*", x, Number(1)) => x
        // Multiplying 1 by x returns the original value
        case BinaryOp("*", Number(1), x) => x
        // Multiplying x by 0 returns zero
        case BinaryOp("*", x, Number(0)) => Number(0)
        // Multiplying 0 by x returns zero
        case BinaryOp("*", Number(0), x) => Number(0)
        // Dividing x by 1 returns the original value
        case BinaryOp("/", x, Number(1)) => x
        // Adding x to 0 returns the original value
        case BinaryOp("+", x, Number(0)) => x
        // Adding 0 to x returns the original value
        case BinaryOp("+", Number(0), x) => x
        // Anything else cannot (yet) be simplified
        case _ => e
      }
    }
```

Again, notice how using the constant-matching and variable-binding features of pattern matching makes it trivial to write these expressions. The only change to `evaluate()` is to include the call to simplify before evaluating:

Listing 12. Calculator (src/calc.scala)

```
    def evaluate(e : Expr) : Double =
    {
      simplify(e) match {
        case Number(x) => x
        case UnaryOp("-", x) => -(evaluate(x))
        case BinaryOp("+", x1, x2) => (evaluate(x1) + evaluate(x2))
        case BinaryOp("-", x1, x2) => (evaluate(x1) - evaluate(x2))
        case BinaryOp("*", x1, x2) => (evaluate(x1) * evaluate(x2))
        case BinaryOp("/", x1, x2) => (evaluate(x1) / evaluate(x2))
      }
    }
```

Further simplification is possible; notice how this only simplifies at the bottom levels of the tree? If we have a `BinaryOp` that contains a `BinaryOp("*", Number(0), Number(5))` and a `Number(5)`, the inner `BinaryOp` can be simplified to a `Number(0)`, but then the outer one can as well because now one of outer `BinaryOp`'s operands is now zero.

In a fit of authorial pique, I will leave that to readers to define. In fact, let's make it fun. If readers want to send me their implementation, I'll include it (with credit) in the next article's code drop and prose. (There's a couple of unit tests that test this condition and will fail as of right now. Your mission, should you choose to accept it, is to make those tests — and any other test that takes the nesting of `BinaryOps` and `UnaryOps` to any depth — pass.)

Conclusion

I'm clearly not finished yet; there's parsing work yet to be done, but the calculator AST is in pretty good shape. We can add additional operations without requiring major surgery, walking the AST doesn't require huge amounts of code (a la the Visitor pattern from the Gang of Four), and already we have some working code to do the calculations themselves (if clients are willing to build the AST for us for evaluation).

More importantly, you've seen how case classes work well with pattern matching to make it almost trivial to create this AST and evaluate it. This is a frequent design in Scala code (and in most functional languages, in fact) and is one you should be comfortable with if you're going to spend any serious amount of time in this environment.

Resources

Learn

- "*The busy Java developer's guide to Scala*: Collection types" (Ted Neward, developerWorks, June 2008) discusses pattern matching.
- "*The busy Java developer's guide to Scala*: Class action" (Ted Neward, developerWorks, February 2008) discusses various control constructs in Scala.
- "*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.
- "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.
- 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.
- 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