developerWorks®

Functional thinking: Either trees and pattern matching

Using Either and generics to create Scala-style pattern matching in Java

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

10 July 2012

Scala's ability to perform dispatch based on pattern matching is a feature much envied by Java™ developers. This installment shows how a combination of standard data structures and generics provides a pattern-matching-like syntax in pure Java.

View more content in this series

About this series

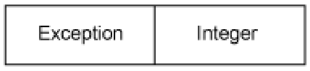
This series aims to reorient your perspective toward a functional mindset, helping you look at common problems in new ways and find ways to improve your day-to-day coding. It explores functional programming concepts, frameworks that allow functional programming within the Java language, functional programming languages that run on the JVM, and some future-leaning directions of language design. The series is geared toward developers who know Java and how its abstractions work but have little or no experience using a functional language.

In the last installment, I introduced a common abstraction in the functional programming world: `Either`. In this installment, I continue my exploration of `Either`, using it to build tree-shaped structures — which in turn enables me to mimic Scala's pattern matching for building traversal methods.

Using generics in Java, an `Either` becomes a single data structure that accepts either of two types, which it stores in `left` and `right` parts.

In the last installment's Roman numeral parser example, `Either` holds either an `Exception` (left) or an `Integer` (right), as illustrated in Figure 1:

Figure 1. The `Either` abstraction holding either of two types



In that example, this assignment populates the `Either`:

```
Either<Exception, Integer> result = RomanNumeralParser.parseNumber("XLII");
```

Scala pattern matching

One of the nice features of Scala is the ability to use *pattern matching* for dispatch (see Resources). It's easier to show than to describe, so consider the function in Listing 1, which converts numerical scores to letter grades:

Listing 1. Using Scala pattern matching to assign letter grades based on score

```
val VALID_GRADES = Set("A", "B", "C", "D", "F")


def letterGrade(value: Any) : String = value match {
  case x:Int if (90 to 100).contains(x) => "A"
  case x:Int if (80 to 90).contains(x) => "B"
  case x:Int if (70 to 80).contains(x) => "C"
  case x:Int if (60 to 70).contains(x) => "D"
  case x:Int if (0 to 60).contains(x) => "F"
  case x:String if VALID_GRADES(x.toUpperCase) => x.toUpperCase
}

printf("Amy scores %d and receives %s\n", 91, letterGrade(91))
printf("Bob scores %d and receives %s\n", 72, letterGrade(72))
printf("Sam never showed for class, scored %d, and received %s\n",
    44, letterGrade(44))
printf("Roy transfered and already had %s, which translated as %s\n",
    "B", letterGrade("B"))
```

In Listing 1, the entire body of the function consists of `match` applied to `value`. For each of the options, a *pattern guard* enables me to filter the matches based on criteria in addition to the argument's type. The advantage of this syntax is a clean partitioning of the options instead of a cumbersome series of `if` statements.

Pattern matching works in conjunction with Scala's *case classes*, which are classes with specialized properties — including the ability to perform pattern matching — to eliminate decision sequences. Consider matching color combinations, as shown in Listing 2:

Listing 2. Matching case classes in Scala

```
class Color(val red:Int, val green:Int, val blue:Int)

case class Red(r:Int) extends Color(r, 0, 0)
case class Green(g:Int) extends Color(0, g, 0)
case class Blue(b:Int) extends Color(0, 0, b)

def printColor(c:Color) = c match {
  case Red(v) => println("Red: " + v)
  case Green(v) => println("Green: " + v)
  case Blue(v) => println("Blue: " + v)
  case col:Color => {
    print("R: " + col.red + ", ")
    print("G: " + col.green + ", ")
    println("B: " + col.blue)
  }

  case null => println("invalid color")
}
```

In Listing 2, I create a base `Color` class, then create specialized single color versions as case classes. When determining which color was passed to the function, I use `match` to pattern match against all the possible options, including the last case, which handles `null`.

Java doesn't have pattern matching, so it can't replicate Scala's ability to create cleanly readable dispatch code. But marrying generics and well-known data structures brings it close, which brings me back to `Either`.

Either trees

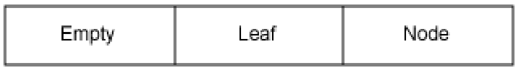
It is possible to model a tree data structure with three abstractions, as shown in Table 1:

Table 1. Building a tree with three abstractions

Empty	**Cell has no value**
Leaf	Cell has a value of a particular data type
Node	Points to other *Leaf*s or *Node*s

I implemented a simple version of the `Either` class in the last installment, but for convenience I'll use the one from the Functional Java framework (see Resources) in this example. Conceptually, the `Either` abstraction expands into as many slots as needed. For example, consider the declaration `Either<Empty, Either<Leaf, Node>>`, which creates a three-part data structure like the one shown in Figure 2:

Figure 2. `Either<Empty, Either<Leaf, Node>>`'s data structure



Armed with an `Either` implementation of the three tree abstractions, I define a tree as shown in Listing 3:

Listing 3. Tree based on `Either`

```
import fj.data.Either;
import static fj.data.Either.left;
import static fj.data.Either.right;

public abstract class Tree {
    private Tree() {}

    public abstract Either<Empty, Either<Leaf, Node>> toEither();

    public static final class Empty extends Tree {
        public Either<Empty, Either<Leaf, Node>> toEither() {
            return left(this);
        }

        public Empty() {}
    }

    public static final class Leaf extends Tree {
        public final int n;

        public Either<Empty, Either<Leaf, Node>> toEither() {
            return right(Either.<Leaf, Node>left(this));
        }
        public Leaf(int n) { this.n = n; }
    }

    public static final class Node extends Tree {
        public final Tree left;
        public final Tree right;

        public Either<Empty, Either<Leaf, Node>> toEither() {
            return right(Either.<Leaf, Node>right(this));
        }

        public Node(Tree left, Tree right) {
            this.left = left;
            this.right = right;
        }
    }
}
```

The abstract `Tree` class in Listing 3 defines within it three `final` concrete classes: `Empty`, `Leaf`, and `Node`. Internally, the `Tree` class uses the three-slotted `Either` shown in Figure 2, enforcing the convention that the leftmost slot always holds `Empty`, the middle slot holds a `Leaf`, and the rightmost slot holds a `Node`. It does this by requiring each class to implement the `toEither()` method, returning the appropriate "slot" for that type. Each "cell" in the data structure is a *union* in the traditional computer-science sense, designed to hold only one of the three possible types at any given time.

Given this tree structure and the fact that I know that its internal structure is based on `<Either, <Left, Node>>`, I can mimic pattern matching for visiting each element in the tree.

Pattern matching for tree traversal

Scala's pattern matching encourages you to think about discrete cases. The `left()` and `right()` methods of Functional Java's `Either` implementation both implement the `Iterable` interface; this enables me to write pattern-matching-inspired code, as shown in Listing 4, to determine the tree's depth:

Listing 4. Checking a tree's depth using pattern-matching-like syntax

```
static public int depth(Tree t) {
    for (Empty e : t.toEither().left())
        return 0;
    for (Either<Leaf, Node> ln: t.toEither().right()) {
        for (Leaf leaf : ln.left())
            return 1;
        for (Node node : ln.right())
            return 1 + max(depth(node.left), depth(node.right));
    }
    throw new RuntimeException("Inexhaustible pattern match on tree");
}
```

The `depth()` method in Listing 4 is a recursive depth-finding function. Because my tree is based on a specific data structure (`<Either, <Left, Node>>`), I can treat each "slot" as a specific case. If the cell is empty, this branch has no depth. If the cell is a leaf, I count it as a tree level. If the cell is a node, I know that I should recursively search both left and right sides, adding a `1` for another level of recursion.

I can also use the same pattern-matching syntax to perform a recursive search of the tree, as shown in Listing 5:

Listing 5. Determining presence in a tree

```
static public boolean inTree(Tree t, int value) {
    for (Empty e : t.toEither().left())
        return false;
    for (Either<Leaf, Node> ln: t.toEither().right()) {
        for (Leaf leaf : ln.left())
            return value == leaf.n;
        for (Node node : ln.right())
            return inTree(node.left, value) | inTree(node.right, value);
    }
    return false;
}
```

As before, I specify the return value for each possible "slot" in the data structure. If I encounter an empty cell, I return `false`; my search has failed. For a leaf, I check the passed value, returning `true` if they match. Otherwise, when encountering a node, I recurse through the tree, using the `|` (non-short circuited `or` operator) to combine the returned boolean values.

To see tree creation and searching in action, consider the unit test in Listing 6:

Listing 6. Testing tree searchability

```
@Test
public void more_elaborate_searchp_test() {
    Tree t = new Node(new Node(new Node(new Node(
            new Node(new Leaf(4),new Empty()),
            new Leaf(12)), new Leaf(55)),
            new Empty()), new Leaf(4));
    assertTrue(inTree(t, 55));
    assertTrue(inTree(t, 4));
    assertTrue(inTree(t, 12));
    assertFalse(inTree(t, 42));
}
```

In Listing 6, I build a tree, then investigate whether elements are present. The `inTree()` method returns `true` if one of the leaves equals the search value, and the `true` propagates up the recursive call stack because of the `|` ("or") operator, as shown in Listing 5.

The example in Listing 5 determines if an element appears in the tree. A more sophisticated version also checks for the number of occurrences, as shown in Listing 7:

Listing 7. Finding number of occurrences in a tree

```
static public int occurrencesIn(Tree t, int value) {
    for (Empty e: t.toEither().left())
        return 0;
    for (Either<Leaf, Node> ln: t.toEither().right()) {
        for (Leaf leaf : ln.left())
            if (value == leaf.n) return 1;
        for (Node node : ln.right())
            return occurrencesIn(node.left, value) + occurrencesIn(node.right, value);
    }
    return 0;
}
```

In Listing 7, I return `1` for every matching leaf, allowing me to count the number of occurrences of each number in the tree.

Listing 8 illustrates tests for `depth()`, `inTree()`, and `occurrencesIn()` for a complex tree:

Listing 8. Testing depth, presence, and occurrences in complex trees

```
@Test
public void multi_branch_tree_test() {
    Tree t = new Node(new Node(new Node(new Leaf(4),
            new Node(new Leaf(1), new Node(
                    new Node(new Node(new Node(
            new Node(new Node(new Leaf(10), new Leaf(0)),
                    new Leaf(22)), new Node(new Node(
                            new Node(new Leaf(4), new Empty()),
                            new Leaf(101)), new Leaf(555))),
                            new Leaf(201)), new Leaf(1000)),
                    new Leaf(4)))),
            new Leaf(12)), new Leaf(27));
    assertEquals(12, depth(t));
    assertTrue(inTree(t, 555));
    assertEquals(3, occurrencesIn(t, 4));
}
```

Because I've imposed regularity on the tree's internal structure, I can analyze the tree during traversal by thinking individually about each case, reflected in the type of element. Although not as expressive as full-blown Scala pattern matching, the syntax comes surprisingly close to the Scala ideal.

Conclusion

In this installment, I showed how imposing regularity on the internal structure of a tree enables Scala-style pattern matching during tree traversals, taking advantage of some inherent properties of generics, `Iterable`s, Functional Java's `Either` class, and a few other ingredients combined to mimic a powerful Scala feature.

In the next installment, I begin an investigation on various method-dispatch mechanisms present in next-generation Java languages, in contrast to the limited options available in Java.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's book discusses tools and practices that help you improve your coding efficiency.
- Scala: Scala is a modern functional language on the JVM.
- *The busy Java developer's guide to Scala*: Learn more about pattern matching, case classes, and other functional features in Scala.
- "Structural Pattern Matching in Java": This post by Rúnar Óli on the Apocalisp blog (an excellent resource for functional programming) was the inspiration for this installment's topic.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- "Tree visitors in Clojure" (Alex Miller, developerWorks, September 2011): Compare tree-data traversal in Java and in Clojure, a functional Lisp for the JVM.
- "Execution in the Kingdom of Nouns" (Steve Yegge, March 2006): An entertaining rant about some aspects of Java language design.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- 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 a software architect and Meme Wrangler at **Thought**Works, a global IT consultancy. He also designs and develops 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 *The Productive Programmer*. He focuses on designing and building large-scale enterprise applications. He is also an internationally acclaimed speaker at developer conferences worldwide. Check out his Web site.