Functional thinking: Rethinking dispatch

How next-generation JVM languages add nuance to method dispatch

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

21 August 2012

Next-generation languages for the Java™ platform have more-flexible method-dispatch mechanisms than the Java language. In this *Functional thinking* installment, Neal Ford explores dispatch mechanisms in functional languages like Scala and Clojure, showing new ways to think about executing code.

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 explored the use of Java generics to mimic *pattern matching* in Scala, which lets you write concise, readable conditional statements. Scala pattern matching is one example of an alternative *dispatch mechanism*, which I'm using as a broad term to describe ways languages dynamically choose behavior. This installment extends the discussion to show how dispatch mechanisms in various functional JVM languages enable more conciseness and flexibility than their Java counterparts.

Improving dispatch with Groovy

In Java, conditional execution ends up using the `if` statement except in very limited cases where the `switch` statement applies. Because long series of `if` statements become difficult to read, Java developers rely on the Gang of Four (GoF) Factory (or Abstract Factory) pattern (see Resources). If you use a language that includes a more flexible decision expression, you can simplify a lot of your code even more.

Groovy has a powerful `switch` statement that mimics the syntax — but not the behavior — of Java's `switch` statement, as shown in Listing 1:

Listing 1. Groovy's vastly improved `switch` statement

```
class LetterGrade {
  def gradeFromScore(score) {
    switch (score) {
      case 90..100 : return "A"
      case 80..<90 : return "B"
      case 70..<80 : return "C"
      case 60..<70 : return "D"
      case 0..<60  : return "F"
      case ~"[ABCDFabcdf]" : return score.toUpperCase()
      default: throw new IllegalArgumentException("Invalid score: ${score}")
    }
  }
}
```

The Groovy `switch` statement accepts a wide variety of dynamic types. In Listing 1, the `score` parameter should be either a number between 0 and 100 or a letter grade. As in Java, you must terminate each `case` with either a `return` or `break`, following the same fall-through semantics. But in Groovy, unlike Java, I can specify ranges (`90..100`, noninclusive ranges (`80..<90`), regular expressions (`~"[ABCDFabcdf]"`), and a default condition.

Groovy's dynamic typing enables me to send different types of parameters and react appropriately, as shown in the unit tests in Listing 2:

Listing 2. Testing Groovy letter grades

```
@Test
public void test_letter_grades() {
  def lg = new LetterGrade()
  assertEquals("A", lg.gradeFromScore(92))
  assertEquals("B", lg.gradeFromScore(85))
  assertEquals("D", lg.gradeFromScore(65))
  assertEquals("F", lg.gradeFromScore("f"))
}
```

A more powerful `switch` gives you a useful middle ground between serial `if`s and the Factory design pattern. Groovy's `switch` lets you match ranges and other complex types, which is similar in intent to pattern matching in Scala.

Scala pattern matching

Scala pattern matching lets you specify matching cases with corresponding behavior. Consider the previous installment's letter-grade example, shown in Listing 3:

Listing 3. Letter grades in Scala

```
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
}
```

In Scala, I allow dynamic input by declaring the parameter type as `Any`. The operator in action is `match`, which tries to match the first true condition and return the results. As shown in Listing 3, each case can include a guard condition that specifies conditions.

Listing 4 shows the results of executing some letter-grade choices:

Listing 4. Testing letters in Scala

```
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"))
```

Pattern matching in Scala is often used in conjunction with Scala's *case classes*, which are intended to represent algebraic and other structured data types.

Clojure's "bendable" language

Another next-generation functional language for the Java platform is Clojure (see Resources). Clojure — an implementation of Lisp on the JVM — has a strikingly different syntax from most contemporary languages. Although developers easily adapt to the syntax, it impresses mainstream Java developers as odd. One of the best features of the Lisp family of languages is *homoiconicity*, meaning that language is implemented using its own data structures, allowing extension to a degree unavailable to other languages.

Java and languages like it include *keywords* — the syntactic scaffolding of the language. Developers can't create new keywords in the language (although some Java-like languages allow extension via metaprogramming), and keywords have semantics unavailable to developers. For example, the Java `if` statement "understands" things like short-circuit Boolean evaluation. Although you can create methods and classes in Java, you can't create fundamental building blocks, so you need to translate problems *into* the syntax of the programming language. (In fact, many developers think their job is to perform this translation.) In Lisp variants like Clojure, the developer can *modify the language toward the problem*, blurring the distinction between what the language designer and developers using the language can create. I'll explore the full implications of homoiconicity in a future installment; the important characteristic to understand here is the philosophy behind Clojure (and other Lisps).

In Clojure, developers use the language to create readable (Lisp) code. For example, Listing 5 shows the letter-grade example in Clojure:

Listing 5. Letter grades in Clojure

```
(defn letter-grade [score]
  (cond
    (in score 90 100) "A"
    (in score 80 90)  "B"
    (in score 70 80)  "C"
    (in score 60 70)  "D"
    (in score 0 60)   "F"
    (re-find #"[ABCDFabcdf]" score) (.toUpperCase score)))

(defn in [score low high]
  (and (number? score) (<= low score high)))
```

In Listing 5, I wrote the `letter-grade` method to read nicely, then implemented the `in` method to make it work. In this code, the `cond` function enables me to evaluate a sequence of tests, handled by my `in` method. As in the previous examples, I handle both numeric and existing letter-grade strings. Ultimately, the return value should be an uppercase character, so if the input is in lowercase, I call the `toUpperCase` method on the returned string. In Clojure, methods are first-class citizens rather than classes, making method invocations "inside-out": the call to `score.toUpperCase()` in Java is equivalent to Clojure's `(.toUpperCase score)`.

I test Clojure's letter-grade implementation in Listing 6:

Listing 6. Testing Clojure letter grades

```
(ns nealford-test
  (:use clojure.test)
  (:use lettergrades))


(deftest numeric-letter-grades
  (dorun (map #(is (= "A" (letter-grade %))) (range 90 100)))
  (dorun (map #(is (= "B" (letter-grade %))) (range 80 89)))
  (dorun (map #(is (= "C" (letter-grade %))) (range 70 79)))
  (dorun (map #(is (= "D" (letter-grade %))) (range 60 69)))
  (dorun (map #(is (= "F" (letter-grade %))) (range 0 59))))

(deftest string-letter-grades
  (dorun (map #(is (= (.toUpperCase %)
           (letter-grade %))) ["A" "B" "C" "D" "F" "a" "b" "c" "d" "f"])))

(run-all-tests)
```

In this case, the test code is more complex than the implementation! However, it shows how concise Clojure code can be.

In the `numeric-letter-grades` test, I want to check every value in the appropriate ranges. If you are unfamiliar with Lisp, the easiest way to decode it is to read inside out. First, the code `#(is (= "A" (letter-grade %)))` creates a new anonymous function that takes a single parameter (if you have an anonymous function that takes a single parameter, you can represent it as `%` within the body) and returns `true` if the correct letter grade returns. The `map` function maps this anonymous function across the collection in the second parameter, which is the list of numbers between in the

appropriate range. In other words, `map` calls this function on each item in the collection, returning a collection of modified values (which I ignore). The `dorun` function allows side effects to occur, which the testing framework relies on. Calling `map` across each `range` in Listing 6 returns a list of all `true` values. The `is` method from the `clojure.test` namespace verifies the value as a side effect. Calling the mapping function within `dorun` allows the side effect to occur correctly and return test results.

Clojure multimethods

A long series of `if` statements is hard to read and debug, yet Java doesn't have any particularly good alternatives at the language level. This problem is typically solved by using either the Factory or Abstract Factory design patterns from the GoF. The Factory pattern works in Java because of class-based polymorphism, allowing me to define a general method signature in a parent class or interface, then choose the implementation that executes dynamically.

Factories and polymorphism

Groovy has a less verbose and easier-to-read syntax than Java, so I'll use it instead of Java for the next couple of code examples — but polymorphism works the same in both languages. Consider this combination of interface and classes to define a `Product` factory, shown in Listing 7:

Listing 7. Creating a product factory in Groovy

```
interface Product {
  public int evaluate(int op1, int op2)
}

class Multiply implements Product {
  @Override
  int evaluate(int op1, int op2) {
    op1 * op2
  }
}

class Incrementation implements Product {
  @Override
  int evaluate(int op1, int op2) {
    def sum = 0
    op2.times {
      sum += op1
    }
    sum
  }
}

class ProductFactory {
  static Product getProduct(int maxNumber) {
    if (maxNumber > 10000)
      return new Multiply()
    else
      return new Incrementation()
  }
}
```

In Listing 7, I create an interface to define the semantics of how to get the product of two numbers, and implement two different versions of the algorithm. In the `ProductFactory`, I determine the rules as to which implementation returns from the factory.

I use the factory as an abstract placeholder for a concrete implementation derived via some decision criteria. For example, consider the code in Listing 8:

Listing 8. Dynamically choosing an implementation

```
@Test
public void decisionTest() {
  def p = ProductFactory.getProduct(10010)
  assertTrue p.getClass() == Multiply.class
  assertEquals(2*10010, p.evaluate(2, 10010))
  p = ProductFactory.getProduct(9000)
  assertTrue p.getClass() == Incrementation.class
  assertEquals(3*3000, p.evaluate(3, 3000))
}
```

In Listing 8, I create two versions of my `Product` implementation, verifying that the correct one returns from the factory.

In Java, inheritance and polymorphism are tightly coupled concepts: polymorphism triggers off the class of the object. In other languages, that coupling is loosened.

A la carte polymorphism in Clojure

Many developers dismiss Clojure because it isn't an object-oriented language, believing that object-oriented languages are the pinnacle of power. That's a mistake: Clojure has all the features of an object-oriented language, implemented independently of other features. For example, Clojure supports polymorphism but isn't restricted to evaluating the class to determine dispatch. Clojure supports polymorphic *multimethods* whereby dispatch is triggered by whatever characteristic (or combination) the developer wants.

Here's an example. In Clojure, data typically resides in `structs`, which mimic the data part of classes. Consider the Clojure code in Listing 9:

Listing 9. Defining a color structure in Clojure

```
(defstruct color :red :green :blue)

(defn red [v]
  (struct color v 0 0))

(defn green [v]
  (struct color 0 v 0))

(defn blue [v]
  (struct color 0 0 v))
```

In Listing 9, I define a structure that holds three values, corresponding to color values. I also create three methods that return a structure saturated with a single color.

A multimethod in Clojure is a method definition that accepts a dispatch function, which returns the decision criteria. Subsequent definitions allow you to flesh out different versions of the method.

Listing 10 shows an example of a multimethod definition:

Listing 10. Defining a multimethod

```
(defn basic-colors-in [color]
  (for [[k v] color :when (not= v 0)] k))

(defmulti color-string basic-colors-in)

(defmethod color-string [:red] [color]
  (str "Red: " (:red color)))

(defmethod color-string [:green] [color]
  (str "Green: " (:green color)))

(defmethod color-string [:blue] [color]
  (str "Blue: " (:blue color)))

(defmethod color-string :default [color]
  (str "Red:" (:red color) ", Green: " (:green color) ", Blue: " (:blue color)))
```

In Listing 10, I define a dispatch function called `basic-colors-in`, which returns a vector of all nonzero color values. For the variations on the method, I specify what should happen if the dispatch function returns a single color; in this case, it returns a string with its color. The last case includes the optional `:default` keyword, which handles remaining cases. For this one, I cannot assume that I received a single color, so the return lists the values of all the colors.

Tests to exercise these multimethods appear in Listing 11:

Listing 11. Testing colors in Clojure

```
(ns colors-test
  (:use clojure.test)
  (:use colors))

(deftest pure-colors
  (is (= "Red: 5" (color-string (struct color 5 0 0))))
  (is (= "Green: 12" (color-string (struct color 0 12 0))))
  (is (= "Blue: 40" (color-string (struct color 0 0 40)))))

(deftest varied-colors
  (is (= "Red:5, Green: 40, Blue: 6" (color-string (struct color 5 40 6)))))
```

In Listing 11, when I call the method with a single color, it executes the singular color version of the multimethod. If I call it with a complex color profile, the default method returns all colors.

Decoupling polymorphism from inheritance provides a powerful, contextualized dispatch mechanism. For example, consider the problem of image file formats, each one having a different set of characteristics to define the type. By using a dispatch *function*, Clojure enables you to build powerful dispatch as contextualized as Java polymorphism but with fewer limitations.

Conclusion

In this installment, I provided a whirlwind tour of various dispatch mechanisms that come in next-generation JVM languages. Working in a language with limited dispatch tends to clutter your code with extraneous workarounds like design patterns. Choosing alternatives in new languages where none existed before can be tough, because you have to shift paradigms; it's part of learning to think functionally.

Resources

Learn

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's most recent book discusses tools and practices that help you improve your coding efficiency.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Functional Java: Functional Java is a framework that adds many functional language constructs to Java.
- *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma et al., Addison-Wesley, 1994): The Gang of Four's classic work on design patterns.
- "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

- 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.