Functional thinking: Why functional programming is on the rise

Why you should care about functional programming, even if you don't plan to change languages any time soon

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

29 January 2013

Java™ developers should learn functional paradigms now, even if they have no immediate plans to move to a functional language such as Scala or Clojure. Over time, all mainstream languages will become more functional; Neal Ford explores the reasons why in this installment.

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 each installment in this series so far, I've illustrated reasons why understanding functional programming is important. But some reasons span installments and only become fully clear within the larger context of combined ideas. In this installment, I explore all the reasons why functional programming is ascendant, incorporating the individual lessons of previous installments.

Over the short history of computer science, the mainstream of technology has sometimes spawned branches, either practical or academic. The 4GLs (fourth-generation languages) of the 1990s exemplify a practical offshoot, and functional programming is an example from academia. Every once in a while a branch will join the mainstream, which is what's happening to functional programming now. Functional languages are sprouting not just on the JVM — where the two most interesting new languages are Scala and Clojure — but also on the .NET platform, where F# is a first-class citizen. Why this embrace of functional programming by all platforms? The answer is

that over time, as runtimes have become capable of handling more busywork, developers have been able to cede more control of mundane tasks to them.

Ceding control

In the early 1980s, when I was in university, we used a development environment called Pecan Pascal. Its unique feature was that the same Pascal code could run on either the Apple II or IBM PC. The Pecan engineers achieved this feat by using something mysterious called "byte code." Developers compiled their Pascal code to this "byte code," which ran on a "virtual machine" written natively for each platform. It was a hideous experience! The resulting code was achingly slow even for simple class assignments. The hardware at the time just wasn't up to the challenge.

A decade after Pecan Pascal, Sun released Java using the same architecture, straining but succeeding in mid-1990s hardware environments. It also added other developer-friendly features such as automatic garbage collection. Having worked in languages like C++, I never want to code in a non-garbage-collected language again. I'd rather spend time at a higher level of abstraction thinking about ways to solve complex business problems, not complicated plumbing problems like memory management.

Java eased our interaction with memory management; functional programming languages enable us to replace other core building blocks with higher-order abstractions, and to focus more on results than on steps.

Results over steps

One of the hallmarks of functional programming is the presence of powerful abstractions that hide many of the details of mundane operations such as iteration. An example I've used throughout this series is number classification — discovering whether a number is *perfect*, *abundant*, or *deficient* (see the first installment for a complete definition). A Java implementation to solve this problem appears in Listing 1:

Listing 1. Java number classifier with cached sum

```
import static java.lang.Math.sqrt;

public class ImpNumberClassifier {
    private Set<Integer> _factors;
    private int _number;
    private int _sum;

    public ImpNumberClassifier(int number) {
        _number = number;
        _factors = new HashSet<Integer>();
        _factors.add(1);
        _factors.add(_number);
        _sum = 0;
    }

    private boolean isFactor(int factor) {
        return _number % factor == 0;
    }

    private void calculateFactors() {
        for (int i = 1; i <= sqrt(_number) + 1; i++)
```

```
            if (isFactor(i))
                addFactor(i);
    }

    private void addFactor(int factor) {
        _factors.add(factor);
        _factors.add(_number / factor);
    }

    private void sumFactors() {
        calculateFactors();
        for (int i : _factors)
            _sum += i;
    }

    private int getSum() {
        if (_sum == 0)
            sumFactors();
        return _sum;
    }

    public boolean isPerfect() {
        return getSum() - _number == _number;
    }

    public boolean isAbundant() {
        return getSum() - _number > _number;
    }

    public boolean isDeficient() {
        return getSum() - _number < _number;
    }
}
```

The code in Listing 1 is typical Java code that uses iteration to determine and sum factors. With functional programming languages, developers care less about details like iteration (used by `calculateFactors()`) and transformations such as summing a list (used by `sumFactors()`), preferring to cede those details to higher-order functions and coarse-grained abstractions.

Coarse-grained abstractions

The presence of abstractions to handle tasks such as iteration make for less code to maintain, and therefore fewer places where errors can occur. Listing 2 shows a terser version of the number classifier, written in Groovy using its functionally styled methods:

Listing 2. Groovy number classifier

```
import static java.lang.Math.sqrt

class Classifier {
  def static isFactor(number, potential) {
    number % potential == 0;
  }

  def static factorsOf(number) {
    (1..number).findAll { isFactor(number, it) }
  }

  def static sumOfFactors(number) {
    factorsOf(number).inject(0, {i, j -> i + j})
  }
```

```
  def static isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }

  def static isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def static isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }
}
```

The code in Listing 2 does all that Listing 1 does (minus the cached sum, which will reappear in an example below) with noticeably less code. For example, the iteration to determine factors in `factorsOf()` disappears with the use of the `findAll()` method, which accepts a code block (a higher-order function) with my filter criteria. Groovy allows for even terser code blocks by letting single-parameter blocks use `it` as the implicit parameter name. Similarly, the `sumOfFactors()` method uses `inject()`, which — using 0 as the seed value — applies the code block to each element, reducing each pair into a single value. The `{i, j -> i + j}` code block returns the sum of the two parameters; applying this block as I "fold" the list a pair at a time yields the sum.

Java developers are accustomed to reuse at the *framework* level; the necessary artifices for reuse in object-oriented languages require so much effort that they are typically reserved for larger problems. Functional languages offer reuse at a more granular level, atop fundamental data structures such as lists and maps, by accommodating customization via higher-order functions.

Few data structures, lots of operations

In object-oriented imperative programming languages, the units of reuse are classes and the messages they communicate with, captured in a class diagram. The seminal work in that space, *Design Patterns: Elements of Reusable Object-Oriented Software* (see Resources), includes at least one class diagram with each pattern. In the OOP world, developers are encouraged to create unique data structures, with specific operations attached in the form of methods. Functional programming languages don't try to achieve reuse in the same way. They prefer a few key data structures (such as list, set, and map) with highly optimized operations on those data structures. You pass data structures plus higher-order functions to "plug into" this machinery, customizing it for a particular use. For example, in Listing 2, the `findAll()` method accepts a code block as the "plug-in" higher-order function that determines the filter criteria, and the machinery applies the filter criteria in an efficient way, returning the filtered list.

Encapsulation at the function level enables reuse at a more granular, fundamental level than building custom class structures. One advantage of this approach is already appearing in Clojure. Recent clever innovations in the libraries have rewritten the `map` function to be automatically parallelizable, meaning that all map operations benefit from a performance boost without developer intervention.

For example, consider the case of parsing XML. A huge number of frameworks exist for this task in Java, each with custom data structures and method semantics (for example, SAX vs. DOM). Clojure parses XML into a standard `Map` structure, rather than forcing you to use a custom data

structure. Because Clojure includes lots of tools for working with maps, performing XPath-style queries is simple using the built-in list-comprehension function, `for`, as shown in Listing 3:

Listing 3. Parsing XML in Clojure

```
(use 'clojure.xml)

(def WEATHER-URI "http://weather.yahooapis.com/forecastrss?w=%d&u=f")

(defn get-location [city-code]
  (for [x (xml-seq (parse (format WEATHER-URI city-code)))
        :when (= :yweather:location (:tag x))]
    (str (:city (:attrs x)) "," (:region (:attrs x)))))

(defn get-temp [city-code]
  (for [x (xml-seq (parse (format WEATHER-URI city-code)))
        :when (= :yweather:condition (:tag x))]
    (:temp (:attrs x))))

(println "weather for " (get-location 12770744) "is " (get-temp 12770744))
```

In Listing 3, I access Yahoo's weather service to fetch a given city's weather forecast. Because Clojure is a Lisp variant, it's easiest to read inside out. The actual call to the service endpoint occurs at `(parse (format WEATHER-URI city-code))`, which uses `String`'s `format()` function to embed the `city-code` into the string. The list-comprehension function, `for`, places the parsed XML, cast using `xml-seq` into a queryable map named `x`. The `:when` predicate determines the match criteria; in this case, I'm searching for a tag (translated into a Clojure keyword) `:yweather:condition`.

To understand the syntax used to pull values from the data structure, it's useful to see what's in it. When parsed, the pertinent call to the weather service returns the data structure shown in this excerpt:

```
({:tag :yweather:condition, :attrs {:text Fair, :code 34, :temp 62, :date Tue,
   04 Dec 2012 9:51 am EST}, :content nil})
```

Because Clojure is optimized to work with maps, keywords become functions on the maps that contain them. The call in Listing 3 to `(:tag x)` is shorthand for "retrieve the value corresponding to the `:tag` key from the map stored in `x`." Thus, `:yweather:condition` yields the maps values associated with that key, which includes the `attrs` map that I fetch `:temp` from using the same syntax.

One of the initially daunting details in Clojure is the seemingly endless ways to interact with maps and other core data structures. However, it's a reflection of the fact that most things in Clojure try to resolve to these core, optimized data structures. Rather than trap parsed XML in a unique framework, it tries instead to convert it to an existing structure that tools already exist for.

An advantage of the reliance on fundamental data structures appears in the XML libraries in Clojure. For traversing tree-shaped structures (such as XML documents), a useful data structure called a *zipper* was created in 1997 (see Resources). A zipper allows you to navigate trees structurally by providing coordinate directions. For example, starting from the root of the tree, you can issue commands such as `(-> z/down z/down z/left)` to navigate to the second-level left

element. Functions already exist in Clojure to covert parsed XML into a zipper, enabling consistent navigation across all tree-shaped structures.

New, different tools

Functional programming offers new types of tools that solve tricky problems in elegant ways. For example, Java developers aren't accustomed to *lazy* data structures, which delay generating their values as long as they can. Whereas the futuristic functional languages offer support for such advanced features, some frameworks retrofit this functionality into Java. For example, the version of number classifier that appears in Listing 4 uses the Totally Lazy framework (see Resources):

Listing 4. Java number classifier using laziness and functional data structures, via Totally Lazy

```
import com.googlecode.totallylazy.Predicate;
import com.googlecode.totallylazy.Sequence;

import static com.googlecode.totallylazy.Predicates.is;
import static com.googlecode.totallylazy.numbers.Numbers.*;
import static com.googlecode.totallylazy.predicates.WherePredicate.where;


public class Classifier {
  public static Predicate<Number> isFactor(Number n) {
      return where(remainder(n), is(zero));
  }

  public static Sequence<Number> getFactors(final Number n){
      return range(1, n).filter(isFactor(n));
  }

  public static Sequence<Number> factors(final Number n) {
      return getFactors(n).memorise();
  }

  public static Number sumFactors(Number n){
      return factors(n).reduce(sum);
  }

  public static boolean isPerfect(Number n){
      return equalTo(n, subtract(sumFactors(n), n));
  }

  public static boolean isAbundant(Number n) {
    return greaterThan(subtract(sumFactors(n), n), n);
  }

  public static boolean isDeficient(Number n) {
    return lessThan(subtract(sumFactors(n), n), n);
  }

}
```

Totally Lazy adds both lazy collections and fluent-interface methods, making heavy use of static imports to make the code readable. If you envy some feature from a next-generation language, some research might yield specific extensions that solve a particular problem.

Bending the language toward the problem

Most developers labor under the misconception that their job is to take a complex business problem and translate it into a language such as Java. They do that because Java isn't particularly flexible as a language, forcing you to mold your ideas into the rigid structure already there. But as developers use malleable languages, they see the opportunity to bend the language more toward their problem rather than the problem toward their language. Languages like Ruby — with it's friendlier-than-mainstream support for domain-specific languages (DSLs) — demonstrated that potential. Modern functional languages go even further. Scala was designed to accommodate hosting internal DSLs, and all Lisps (including Clojure) have unparalleled flexibility in enabling the developer to mold the language to the problem. For example, Listing 5 uses the XML primitives in Scala to implement Listing 3's weather example:

Listing 5. Scala's syntactic sugar for XML

```
import scala.xml._
import java.net._
import scala.io.Source

val theUrl = "http://weather.yahooapis.com/forecastrss?w=12770744&u=f"

val xmlString = Source.fromURL(new URL(theUrl)).mkString
val xml = XML.loadString(xmlString)

val city = xml \\ "location" \\ "@city"
val state = xml \\ "location" \\ "@region"
val temperature = xml \\ "condition" \\ "@temp"

println(city + ", " + state + " " + temperature)
```

Scala was designed for malleability, allowing extensions such as operator overloading and implicit types. In Listing 5, Scala is extended to allow XPath-like queries using the `\\` operator.

Aligning with language trends

One of the goals in functional programming is minimal mutable state. In Listing 1, two types of shared state manifest. Both `_factors` and `_number` exist to make the code easier to test (the original version of this code was written to illustrate maximum testability), and could be collapsed into larger functions to eliminate them. However, `_sum` exists for a different reason. I anticipate that users of this code will likely need to check more than one classification. (For example, if a check for perfection fails, I'll likely next check for abundance.) The operation to sum the factors is potentially expensive, so I've created a lazily initialized accessor for it. Upon first invocation, it calculates the sum and stores it in the `_sum` member variable to optimize future invocations.

Like garbage collection, caching can be relegated to the language now. The Groovy number classifier in Listing 2 omits the lazy initialization of sum that appears in Listing 1. If I wanted to implement that same functionality, I could change the classifier as shown in Listing 6:

Listing 6. Adding a cache by hand

```
class ClassifierCachedSum {
  private sumCache

  ClassifierCachedSum() {
    sumCache = [:]
  }

  def sumOfFactors(number) {
    if (sumCache.containsKey(number))
      return sumCache[number]
    else {
      def sum = factorsOf(number).inject(0, {i, j -> i + j})
      sumCache.putAt(number, sum)
      return sum
    }
  }
  // ... other code omitted
```

In recent versions of Groovy, the code in Listing 6 is no longer necessary. Consider the improved version of the classifier in Listing 7:

Listing 7. Memoized number classifier

```
class ClassifierMemoized {
  def static dividesBy = { number, potential ->
    number % potential == 0
  }
  def static isFactor = dividesBy.memoize()

  def static factorsOf(number) {
    (1..number).findAll { i -> isFactor.call(number, i) }
  }

  def static sumFactors = { number ->
    factorsOf(number).inject(0, {i, j -> i + j})
  }
  def static sumOfFactors = sumFactors.memoize()

  def static isPerfect(number) {
    sumOfFactors(number) == 2 * number
  }

  def static isAbundant(number) {
    sumOfFactors(number) > 2 * number
  }

  def static isDeficient(number) {
    sumOfFactors(number) < 2 * number
  }
}
```

Any pure function (one that has no side effect) can be *memoized*, as in the `sumOfFactors()` method in Listing 7. Memoizing the function allows the runtime to cache recurring values, eliminating the need for a hand-written cache. In fact, notice the relationship between `getFactors()`, which does the actual work, and the `factors()` method — the memoized version of `getFactors()`. Totally Lazy also adds memoization to Java, yet another advanced functional feature feeding back into the mainstream.

As runtimes gain more power and excess overhead, developers can cede busy work to the language, freeing us to think about more important problems. Memoization in Groovy is one example of many; all modern languages are adding functional constructs as the underlying runtimes allow, including frameworks like Totally Lazy.

Conclusion

The development world is becoming more functional as runtimes gain power and languages gain more-powerful abstractions, allowing developers to spend more time thinking about the implications of results rather than how to generate them. As abstractions such as higher-order functions appear in languages, they become the customization machinery for highly optimized operations. Rather than create frameworks to handle problems such as XML, you can transform it into data structures that you already have tools to work with.

With publication of this 20th installment, *Functional thinking* will go on hiatus, while I pursue a new series that explores three next-generation JVM languages. *Java.next* will give you a glimpse into your near future — and help you make educated choices about the time you must devote to new-language learning.

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.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- *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.
- Zipper: Read Wikipedia's definition of the zipper data structure.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Totally Lazy: Totally Lazy is a functional framework that adds laziness and fluency to Java.
- 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. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

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.