Functional thinking: Functional features in Groovy, Part 3

Memoization and caching

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

31 January 2012

Modern dynamic languages have incorporated many functional features to take mundane tasks off developers' hands. This article explores the benefits of caching at the function level with Groovy, contrasting it with an imperative approach. It illustrates two types of caching — intramethod and external — and discusses the advantages and disadvantages of the imperative and functional versions.

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.

The more low-level details a programming language can handle for you, the fewer places that leaves you to introduce bugs and complexity. (A classic example is garbage collection and memory management on the JVM.) As I've emphasized in previous installments, one of the characteristics of functional languages — and of dynamic languages with functional features — is that they let you selectively cede control over mundane details to the language and runtime. In "Functional features in Groovy, Part 1," for example, I showed how recursion eliminates the need for the developer to maintain state. In this installment, I do the same for caching with Groovy.

Caching is a common requirement (and source of hard-to-find bugs). In object-oriented languages, caching generally happens at the data object level, and developers must manage it themselves. Many functional programming languages build caching in at the *function* level via a mechanism called *memoization*. In this article, I investigate two caching use cases for functions: intraclass

versus external calls. I also illustrate two alternatives for implementing caching: hand-crafted state and memoization.

Method-level caching

If you have read any of the installments of this series, you're familiar with the problem of number classification (explained in the first installment). The starting point for this article is a Groovy version (derived in the last installment), shown in Listing 1:

Listing 1. Classifier in Groovy

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

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

  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 internal algorithm of the `factorsOf()` method in Listing 1 can be optimized by using tricks I covered in "Thinking functionally, Part 2," but I'm more interested in function-level caching for this example.

Because the `Classifier` class classifies numbers, a common use for it would entail running the same number through multiple methods to classify it. For example, consider the code in Listing 2:

Listing 2. Common usage for number classification

```
//...
if (Classifier.isPerfect(n)) print '!'
else if (Classifier.isAbundant(n)) print '+'
else if (Classifier.isDeficient(n)) print '-'
//...
```

In the current implementation, I must recalculate the sum of the factors for every classification method that I call. This is an example of *intraclass* caching: during normal use, the `sumOfFactors()` method is typically called multiple times per number. In this common use case, this is an inefficient approach.

Caching sum

One of the ways to make the code more efficient is to leverage hard work already done. Because generating the sum of the factors is expensive, I want to do it only once for each number. Toward that end, I create a cache to store calculations, as shown in Listing 3:

Listing 3. Caching sum

```
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
    }
  }
  //... remainder of code unchanged
}
```

In Listing 3, I create a hash named `sumCache` in the constructor. In the `sumOfFactors()` method, I check to see if the sum for the parameter is already cached and return it. Otherwise, I do the expensive calculation and put the sum in the cache before returning it.

The code is more complicated, but the results speak for themselves. I run all the examples through a series of unit tests that follow the pattern shown in Listing 4:

Listing 4. Tests

```
class ClassifierTest {
  def final TEST_NUMBER_MAX = 1000
  def classifier = new ClassifierCachedSum()

  @Test
  void classifier_many_checks_without_caching() {
    print "Nonoptimized:              "
    def start = System.currentTimeMillis()
    (1..TEST_NUMBER_MAX).each {n ->
      if (Classifier.isPerfect(n)) print '!'
      else if (Classifier.isAbundant(n)) print '+'
      else if (Classifier.isDeficient(n)) print '-'
    }
    println "\n\t ${System.currentTimeMillis() - start} ms"
    print "Nonoptimized (2nd):        "
    start = System.currentTimeMillis()
    (1..TEST_NUMBER_MAX).each {n ->
      if (Classifier.isPerfect(n)) print '!'
      else if (Classifier.isAbundant(n)) print '+'
      else if (Classifier.isDeficient(n)) print '-'
    }
    println "\n\t ${System.currentTimeMillis() - start} ms"
  }
```

When I run the tests in Listing 4, the results indicate that the caching helps, as shown in Listing 5:

Listing 5. Profiling results of caching sum

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
```

Listing 5 illustrates that the nonoptimized version (from Listing 1) runs in 577ms the first time, compared to the cached version, which takes 600ms for its first run. For these two cases, the difference is insignificant. However, the second run of the nonoptimized version scores 280ms. The difference between the first and second can be attributed to environmental factors such as garbage collection. The second run of the cached version shows a dramatic speed increase, scoring a mere 50ms. When the second run happens, all the values are cached; now I'm measuring how fast I can read from a hash. The difference between the nonoptimized first run and cached first run is negligible, but it's dramatic for the second run. This is an example of *external caching*: the overall results are used by whoever is calling the code, so the second run is very fast.

Caching sums makes a huge difference but includes trade-offs. `ClassifierCachedSum` can no longer contain pure static methods. The internal cache represents state, so I must make all the methods that interact with the cache nonstatic, which has a ripple effect. I could rig some Singleton solution (see Resources), but that adds complexity too. Because I control the cache variable, I must ensure correctness (by using tests, for example). Although caching improves performance, it isn't free: it adds accidental complexity and a maintenance burden to my code (see Resources).

Caching everything

If caching the sums speeds up the code so much, why not cache every intermediate result that's likely to recur? That's the goal in Listing 6:

Listing 6. Caching everything

```
class ClassifierCached {
  private sumCache, factorCache

  ClassifierCached() {
    sumCache = [:]
    factorCache = [:]
  }

  def sumOfFactors(number) {
    sumCache.containsKey(number) ?
      sumCache[number] :
      sumCache.putAt(number, factorsOf(number).inject(0, {i, j -> i + j}))
  }

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

  def factorsOf(number) {
    factorCache.containsKey(number) ?
      factorCache[number] :
      factorCache.putAt(number, (1..number).findAll { i -> isFactor(number, i) })
```

```
  }

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

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

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

}
```

In `ClassifierCached` in Listing 6, I add caches both for the sum of factors and for the factors of a number. Rather than use the verbose syntax shown in Listing 3, I use the tertiary operator, which is surprisingly expressive in this case. For example, in the `sumOfFactors()` method, I use the condition part of the tertiary operator to check the cache for the number. In Groovy, the last line of a method is the method's return value, so the cached value is returned if I get a cache hit; otherwise, I calculate the number, put it in the cache, and return the value. (Groovy's `putAt()` method returns the value added to the hash.) Listing 7 shows the results:

Listing 7. Caching everything: Test results

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
Cached:
  411 ms
Cached (2nd run):
  38 ms
```

The fully cached version (which is an entirely new class and instance variable in these test runs) scores 411ms for the first run and a blazing 38ms in the second run, once the cache has been primed. Although these results are good, this approach doesn't scale particularly well. In Listing 8, which shows results for testing 5,000 numbers, the cached version takes much longer to prime but still has very fast read times for subsequent runs:

Listing 8. Classifying 5,000 numbers test results

```
Test for range 1-5000
Nonoptimized:
  6199 ms
Nonoptimized (2nd):
  5064 ms
Cached sum:
  5962 ms
Cached sum (2nd run):
  93 ms
Cached:
  6494 ms
Cached (2nd run):
  41 ms
```

The outcome for 10,000 numbers is more dire, as shown in Listing 9:

Listing 9. (Attempting to) classify 10,000 numbers

```
Test for range 1-10000
Nonoptimized:
  43937 ms
Nonoptimized (2nd):
  39479 ms
Cached sum:
  44109 ms
Cached sum (2nd run):
  289 ms
Cached:
java.lang.OutOfMemoryError: Java heap space
```

As Listing 9 shows, the developer responsible for the caching code must worry about both its correctness and its execution conditions. This is a perfect example of *moving parts*: state in code that a developer must maintain and dissect implications for.

Memoization

Functional programming strives to minimize moving parts by building reusable mechanisms into the runtime. *Memoization* is a feature built into a programming language that enables automatic caching of recurring function-return values. In other words, it automatically supplies the code I've written in Listing 3 and Listing 6. Many functional languages support memoization, and Groovy recently added support as well (see Resources).

To memoize a function in Groovy, you define it as a code block, then execute the `memoize()` method to return a function whose results will be cached.

Memoizing sum

To implement caching for `sumOfFactors()` as I did in Listing 3, I memoize the `sumOfFactors()` method, as shown in Listing 10:

Listing 10. Memoizing sum

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

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

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

 // remainder of code unchanged
}
```

In Listing 10, I create the `sumFactors()` method as a code block (note the `=` and parameter placement). This is a pretty generic method and could just as well be pulled from a library somewhere. To memoize it, I assign the name `sumOfFactors` as the `memoize()` method call on the function reference.

Running the memoized version yields the results shown in Listing 11:

Listing 11. Memoizing sum: Test results

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
Cached:
  411 ms
Cached (2nd run):
  38 ms
Partially Memoized:
  228 ms
Partially Memoized (2nd):
  60 ms
```

The partially memoized first run scores approximately the same as the second nonoptimized run. In both cases, memory and other environmental issues have been resolved, so similar scores make sense. The partially memoized second run, however, shows the same dramatic speed increase as the handwritten cached-sum version — with literally a two-line change to the original code (changing `sumFactors()` into a code block, and making `sumOfFactors()` point to a memoized instance of the code block).

Memoizing everything

Just as I cached everything earlier, why not memoize everything with potentially reusable results? That version of the classifier appears in Listing 12:

Listing 12. Memoizing everything

```
class ClassifierMemoized {
  def static dividesBy = { number, potential ->
    number % potential == 0
  }
  def static isFactor = dividesBy.memoize()
//  def static isFactor = dividesBy.memoizeAtMost(100)

  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
  }
}
```

As with the caching-everything case shown in Listing 6, memoizing everything has pros and cons. Listing 13 shows the results for 1,000 numbers:

Listing 13. Memoizing everything for 1,000 numbers

```
Test for range 1-1000
Nonoptimized:
  577 ms
Nonoptimized (2nd):
  280 ms
Cached sum:
  600 ms
Cached sum (2nd run):
  50 ms
Cached:
  411 ms
Cached (2nd run):
  38 ms
Partially Memoized:
  228 ms
Partially Memoized (2nd):
  60 ms
Memoized:
  956 ms
Memoized(2nd)
  19 ms
```

Memoizing everything slows down the first run but has the fastest subsequent run of any case — but only for small sets of numbers. As with the imperative caching solution tested in Listing 8, large number sets impede performance drastically. In fact, the memoized version runs out of memory

in the 5,000-number case. But for the imperative approach to be robust, safeguards and careful awareness of the execution context are required — another example of imperative moving parts. With memoization, optimization occurs at the function level. Look at the memoization results in Listing 14:

Listing 14. Tuning memoization

```
Test for range 1-10000
Nonoptimized:
  41909 ms
Nonoptimized (2nd):
  22398 ms
Memoized:
  55685 ms
Memoized(2nd)
  98 ms
```

I produce the results shown in Listing 14 by calling the `memoizeAtMost(1000)` method instead of `memoize()`. Like other languages that support memoization, Groovy has several methods to help optimize results, as shown in Table 1:

Table 1. Memoization methods in Groovy

Method	Description
`memoize()`	Creates a caching variant of the closure
`memoizeAtMost()`	Creates a caching variant of the closure with an upper limit on the cache size
`memoizeAtLeast()`	Creates a caching variant of the closure with automatic cache size adjustment and lower limit on the cache size
`memoizeBetween()`	Creates a caching variant of the closure with automatic cache size adjustment and lower and upper limits on the cache size

In the imperative version, the developer owns the code (and responsibility). Functional languages build generic machinery — sometimes with customization knobs (in the form of alternate functions) — that you can apply to standard constructs. Functions are a fundamental language element, so optimizing at that level gives you advanced functionality for free. The memoization versions in this article with small number sets outperform the handwritten caching code.

Conclusion

This installment illustrates a common theme throughout this series: functional programming makes things easier by minimizing moving parts. I juxtaposed two different caching solutions for a common number-classification use case: testing the same number against several categories. Building a cache by hand is straightforward, but it adds statefulness and complexity to the code. Using functional-language features like memoization, I can add caching at the function level, achieving better results (with virtually no change to my code) than the imperative version. Functional programming eliminates moving parts, allowing you to focus your energy on solving real problems.

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.
- Closure memoization: Check out the release notes for the memoization feature added to Groovy 1.8.
- The Singleton Pattern: Singleton is a well-known Gang of Four Design Pattern to handle global state in an object-oriented language.
- "Evolutionary architecture and emergent design: Investigating architecture and design": Read about *essential* versus *accidental* complexity in software.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Groovy: Groovy is a dynamic language on the JVM that offers many functional features.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.

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.