Functional thinking: Laziness, Part 1

Exploring lazy evaluation in Java

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

20 November 2012

A common feature of many functional programming languages is *lazy evaluation*, whereby expressions are evaluated only when necessary rather than upon declaration. Java™ doesn't support this style of laziness, but several frameworks and related languages do. This article shows how to build laziness into your Java applications, using pure Java and functional frameworks.

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.

Lazy evaluation — deferral of expression evaluation for as long as possible — is a feature of many functional programming languages. Lazy collections deliver their elements as needed rather than precalculating them, offering several benefits. First, you can defer expensive calculations until they're absolutely needed. Second, you can create infinite collections, which keep delivering elements as long as they keep receiving requests. Third, lazy use of functional concepts such as `map` and `filter` enable you to generate more-efficient code (see Resources for a link to a relevant discussion by Brian Goetz). Java doesn't natively support laziness, but several frameworks and successor languages do, which I explore in this installment and the next.

Consider this snippet of pseudo code for printing the length of a list:

```
print length([2+1, 3*2, 1/0, 5-4])
```

If you try to execute this code, the result will vary depending on the type of programming language it's written in: *strict* or *nonstrict* (also known as *lazy*). In a strict programming language, executing (or perhaps even compiling) this code results in a `DivByZero` exception because of the list's third element. In a nonstrict language, the result is `4`, which accurately reports the number of items in the list. After all, the method I'm calling is `length()`, not `lengthAndThrowExceptionWhenDivByZero()`! Haskell is one of the few nonstrict languages in use (see Resources). Alas, Java doesn't support nonstrict evaluation, but you can still take advantage of the concept of laziness in Java.

Lazy iterator in Java

Java's lack of native support for lazy collections doesn't mean you can't simulate one using an `Iterator`. As in several previous installments of this series, I'll use a simple prime-number algorithm to illustrate functional concepts. I'll build on the optimized class presented in the last installment with the enhancements that appear in Listing 1:

Listing 1. Simple algorithm for determining prime numbers

```
import java.util.HashSet;
import java.util.Set;
import static java.lang.Math.sqrt;

public class Prime {

    public static boolean isFactor(int potential, int number) {
        return number % potential == 0;
    }

    public static Set<Integer> getFactors(int number) {
        Set<Integer> factors = new HashSet<Integer>();
        factors.add(1);
        factors.add(number);
        for (int i = 2; i < sqrt(number) + 1; i++)
            if (isFactor(i, number)) {
                factors.add(i);
                factors.add(number / i);
            }
        return factors;
    }

    public static int sumFactors(int number) {
        int sum = 0;
        for (int i : getFactors(number))
            sum += i;
        return sum;
    }

    public static boolean isPrime(int number) {
        return number == 2 || sumFactors(number) == number + 1;
    }

    public static Integer nextPrimeFrom(int lastPrime) {
        lastPrime++;
        while (! isPrime(lastPrime)) lastPrime++;
        return lastPrime;
    }
}
```

A previous installment discusses at length the inner details of how this class determines if an integer is a prime number. In Listing 1, I add the `nextPrimeFrom()` method to generate the next prime number based on the input parameter. That method plays a role in this article's upcoming examples.

Generally, developers think of iterators as using collections as backing stores, but anything that supports the `Iterator` interface qualifies. Thus, I can create an infinite iterator of prime numbers, as shown in Listing 2:

Listing 2. Creating a lazy iterator

```
public class PrimeIterator implements Iterator<Integer> {
    private int lastPrime = 1;

    public boolean hasNext() {
        return true;
    }

    public Integer next() {
        return lastPrime = Prime.nextPrimeFrom(lastPrime);
    }

    public void remove() {
       throw new RuntimeException("Can't change the fundamental nature of the universe!");
    }
}
```

In Listing 2, the `hasNext()` method always returns `true` because, as far as we know, the number of prime numbers is infinite. The `remove()` method doesn't apply here, so I throw an exception in case of accidental invocation. The workhorse method is the `next()` method, which handles two chores with its single line. First, it generates the next prime number based on the last one by calling the `nextPrimeFrom()` method that I added in Listing 1. Second, it exploits Java's ability to assign and return in a single statement, updating the internal `lastPrime` field. I exercise the lazy iterator in Listing 3:

Listing 3. Testing the lazy iterator

```
public class PrimeTest {
    private ArrayList<Integer> PRIMES_BELOW_50 = new ArrayList<Integer>() {{
                add(2);  add(3);  add(5);  add(7);  add(11);  add(13);
                add(17); add(19); add(23); add(29); add(31);  add(37);
                add(41); add(43); add(47);
            }};

    @Test
    public void prime_iterator() {
        Iterator<Integer> it = new PrimeIterator();
        for (int i : PRIMES_BELOW_50) {
            assertTrue(i == it.next());
        }
    }
}
```

In Listing 3, I create a `PrimeIterator` and verify that it reports the first 50 prime numbers. Although not the typical use of an iterator, it does mimic some of the useful behavior of lazy collections.

Using `LazyList`

Jakarta Commons includes a `LazyList` class (see Resources), which uses a combination of the Decorator design pattern and a factory. To use Commons `LazyList`, you must wrap an existing list to make it lazy, and create a factory for new values. Consider the usage of `LazyList` in Listing 4:

Listing 4. Testing a Commons `LazyList`

```
public class PrimeTest {
    private ArrayList<Integer> PRIMES_BELOW_50 = new ArrayList<Integer>() {{
                add(2);  add(3);  add(5);  add(7);  add(11);  add(13);
                add(17); add(19); add(23); add(29); add(31);  add(37);
                add(41); add(43); add(47);
            }};

    @Test
    public void prime_factory() {
        List<Integer> primes = new ArrayList<Integer>();
        List<Integer> lazyPrimes = LazyList.decorate(primes, new PrimeFactory());
        for (int i = 0; i < PRIMES_BELOW_50.size(); i++)
            assertEquals(PRIMES_BELOW_50.get(i), lazyPrimes.get(i));
    }
}
```

In Listing 4, I create a new empty `ArrayList` and wrap it in the Commons `LazyList.decorate()` method, along with a `PrimeFactory` for generating new values. The Commons `LazyList` will use whatever values already reside in the list, but when the `get()` method is called for an index that doesn't yet have a value, `LazyList` uses the factory (in this case, `PrimeFactory()`) to generate and populate the values. `PrimeFactory` appears in Listing 5:

Listing 5. `PrimeFactory` used by `LazyList`

```
public class PrimeFactory implements Factory {
    private int index = 0;

    @Override
    public Object create() {
        return Prime.indexedPrime(index++);
    }
}
```

All lazy lists need a way to generate subsequent values. In Listing 2, I use the combination of the `next()` method and `Prime`'s `nextPrimeFrom()` method. For Commons `LazyList`s in Listing 4, I use the `PrimeFactory` instance.

One quirk of the Commons `LazyList` implementation is the dearth of information passed to the factory method when a new value is requested. As designed, it doesn't even pass the index of the requested element, forcing the maintenance of the current state upon the `PrimeFactory` class. This creates an undesirable dependence on the backing list (because it must initialize as empty to sync up the index numbers with `PrimeFactory`'s internal state). Commons `LazyList` is a rudimentary implementation at best; much better open source alternatives exist, such as Totally Lazy.

Totally Lazy

Totally Lazy is a framework that adds first-class laziness to Java (see Resources). In a previous installment, I introduced Totally Lazy but didn't do it idiomatic justice. One of the framework's goals is to create highly readable Java code by using combinations of static imports. The simple prime-number finder in Listing 6 is written to exploit this Totally Lazy feature fully:

Listing 6. Totally Lazy, fully utilizing static imports

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

import static com.googlecode.totallylazy.Predicates.is;
import static com.googlecode.totallylazy.numbers.Numbers.equalTo;
import static com.googlecode.totallylazy.numbers.Numbers.increment;
import static com.googlecode.totallylazy.numbers.Numbers.range;
import static com.googlecode.totallylazy.numbers.Numbers.remainder;
import static com.googlecode.totallylazy.numbers.Numbers.sum;
import static com.googlecode.totallylazy.numbers.Numbers.zero;
import static com.googlecode.totallylazy.predicates.WherePredicate.where;

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

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

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

    public static boolean isPrime(Number n){
        return equalTo(increment(n), sumFactors(n));
    }
}
```

In Listing 6, after the static imports are completed, the code is atypical of Java yet quite readable. Totally Lazy was partly inspired by the Hamcrest testing extension fluent interface for JUnit (see Resources) and uses some of Hamcrest's classes. The `isFactor()` method becomes a call to the `where()` method, using Totally Lazy's `remainder()` method in conjunction with the Hamcrest `is()` method. Similarly, the `factors()` method becomes a `filter()` call on a `range()` object, and I use the now-familiar `reduce()` method to determine the sum. Finally, the `isPrime()` method uses Hamcrest's `equalTo()` method to determine if the sum of factors equals the incremented number.

Astute readers will note that the implementation in Listing 6 does implement the optimization I wrote about in the preceding installment, using a more efficient algorithm to determine factors. The optimized version appears in Listing 7:

Listing 7. Totally Lazy implementation of the optimized prime-number finder

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

    public static Sequence<Number> getFactors(final Number n){
        Sequence<Number> lowerRange = range(1, squareRoot(n)).filter(isFactor(n));
        return lowerRange.join(lowerRange.map(divide().apply(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 isPrime(Number n){
        return equalTo(increment(n), sumFactors(n));
    }
}
```

Two primary changes appear in Listing 7. First, I improve the `getFactors()` algorithm to harvest the factors below the square root, then generate the symmetrical ones above the square root. In Totally Lazy, even operations like `divide()` can be expressed in its fluent-interface style. The second change involves memoization, which automatically caches function invocations with the same parameters; I've changed the `sumFactors()` method to use the `factors()` method, which is the memoized `getFactors()` method. Totally Lazy implements memoization as part of the framework, so no further code is necessary to implement this optimization; however, the framework author spells it `memorise()` instead of the more traditional (as in Groovy) `memoize()`.

True to its name, Totally Lazy tries to use laziness as much as possible throughout the framework. In fact, the Totally Lazy framework itself includes a `primes()` generator that implements an infinite sequence of prime numbers using the framework's building blocks. Consider the excerpts from the `Numbers` class that are shown in Listing 8:

Listing 8. Totally Lazy excerpts implementing infinite prime numbers

```
public static Function1<Number, Number> nextPrime = new Function1<Number, Number>() {
       @Override
       public Number call(Number number) throws Exception {
              return nextPrime(number);
      }
};

public static Computation<Number> primes = computation(2, computation(3, nextPrime));

public static Sequence<Number> primes() {
       return primes;
}

public static LogicalPredicate<Number> prime = new LogicalPredicate<Number>() {
    public final boolean matches(final Number candidate) {
        return isPrime(candidate);
    }

};

public static Number nextPrime(Number number) {
    return iterate(add(2), number).filter(prime).second();
}
```

The `nextPrime()` method creates a new `Function1`, which is Totally Lazy's implementation of a pseudo higher-order function, this one designed to accept a single `Number` parameter and produce a `Number` result. In this case, it returns the result from the `nextPrime()` method. The `primes` variable is created to hold the state of the prime numbers, performing a computation with `2` (the first prime number) as the seed value, and using a new computation for the next prime number. This is a typical pattern in lazy implementations: hold the next element plus a method for generating subsequent values. The `prime()` method is merely a wrapper around the `prime` computation performed earlier.

To determine the `nextPrime()` in Listing 8, Totally Lazy creates a new `LogicalPredicate` to encapsulate the determination of primeness, then creates the `nextPrime()` method, which uses the fluent interfaces within Totally Lazy to determine the next prime number.

Totally Lazy does an excellent job of using the lowly static import in Java to facilitate quite readable code. Many developers believe Java is a poor host for internal domain-specific languages, but Totally Lazy debunks that attitude. And it uses laziness aggressively, deferring every possible operation.

Conclusion

In this installment, I explored laziness, first by creating a simulated lazy collection in Java using an iterator, then by using the rudimentary `LazyList` class from Jakarta Commons Collections. Finally, I implemented the sample code with Totally Lazy, using lazy collections both internally for the determination of prime numbers and in the lazy infinite collection of prime numbers. Totally Lazy also illustrates the expressiveness of the fluent-interface style, using static imports to improve code readability.

In the next installment, I'll continue the exploration of laziness, moving to Groovy, Scala, and Clojure.

Resources

Learn

- Haskell: Haskell is an open source advanced functional programming language, the product of many years of research.
- `LazyList`: This is the API page for the Jakarta Commons `LazyList` implementation.
- "State of the Lambda: Libraries Edition": Brian Goetz discusses the benefits of laziness in code generation.
- Totally Lazy: The Totally Lazy framework adds tons of functional extensions to Java, using an intuitive DSL-like interface.
- Hamcrest: Hamcrest provides a library of matcher objects (also known as constraints or predicates) that enable "match" rules to be defined declaratively, for use in other frameworks. (Hamcrest is being ported to GitHub.)
- "*Evolutionary architecture and emergent design:* Fluent interfaces" (Neal Ford, developerWorks, July 2010): See how fluent interfaces remove unnecessary noise from code syntax, making it more readable.
- *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma et al., Addison-Wesley, 1994): You can read about the Decorator pattern in the Gang of Four's classic work on design patterns.
- Scala: Scala is a modern, functional language on the JVM.
- Clojure: Clojure is a modern, functional Lisp that runs on 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

- 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.