Functional thinking: Immutability

Make Java code more functional by changing less

Neal Ford
Software Architect / Meme Wrangler
ThoughtWorks Inc.

26 July 2011

Immutability is one of the building blocks of functional programming. This *Functional thinking* installment discusses the many aspects of immutability in the Java™ language and shows how to create immutable Java classes in both traditional and newer styles. It also shows two ways to create immutable classes in Groovy, removing much of the pain of the Java implementation. Finally, you'll learn when this abstraction is appropriate.

View more content in this series

Object-oriented programming makes code understandable by encapsulating moving parts. Functional programming makes code understandable by minimizing *moving parts.*
— Michael Feathers, author of *Working with Legacy Code*, via Twitter

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 this installment, I discuss one of the building blocks of functional programming: *immutability*. An immutable object's state cannot change after construction. In other words, the constructor is the only way you can mutate the object's state. If you want to change an immutable object, you don't — you create a new object with the changed value and point your reference to it. (`String` is a classic example of an immutable class built into the core of the Java language.) Immutability is key to functional programming because it matches the goals of minimizing the parts that change, making it easier to reason about those parts.

Implementing immutable classes in Java

Modern object-oriented languages like Java, Ruby, Perl, Groovy, and C# have built convenient mechanisms to make it easy to modify state in controlled ways. However, state is so fundamental to computation that you can never predict where it will leak out. For example, writing high-performance, correct multithreaded code is difficult in object-oriented languages because of the myriad mutability mechanisms. Because Java is optimized for manipulating state, you have to work around some of those mechanisms to gain the benefits of immutability. But once you know to avoid a few pitfalls, building immutable classes in Java gets easier.

Defining immutable classes

To make a Java class immutable, you must:

- Make all fields `final`.
  When you define fields as `final` in Java, you must either initialize them at declaration time or in the constructor. Don't panic if your IDE complains that you don't initialize them at the declaration site. It'll realize that you've come back to your senses when you write the appropriate code in the constructor.
- Make the class `final` so that it cannot be overridden.
  If the class can be overridden, its methods' behaviors can be overridden as well, so your safest bet is to disallow subclassing. Notice that this is the strategy used by Java's `String` class.
- Do not provide a no-argument constructor.
  If you have an immutable object, you must set whatever state it will contain in the constructor. If you have no state to set, why do you have an object? Static methods on a stateless class would work just as well. Thus, you should never have a no-argument constructor for an immutable class. If you're using a framework that requires this for some reason, see if you can satisfy it by providing a private no-argument constructor (which is visible via reflection). Notice that the lack of a no-argument constructor violates the JavaBeans standard, which insists on a default constructor. But JavaBeans cannot be immutable anyway, because of the way the `set`*`XXX`* methods work.
- Provide at least one constructor.
  If you haven't provided a no-argument one, this is your last chance to add some state to the object!
- Do not provide any mutating methods other than the constructor.
  Not only must you avoid typical JavaBeans-inspired `set`*`XXX`* methods, but you must also be careful not to return mutable object references. The fact that the object reference is `final` doesn't mean that you can't change what it points to. Thus, you need to make sure you defensively copy any object references you return from `get`*`XXX`* methods.

"Traditional" immutable class

An immutable class that meets the previous requirements appears in Listing 1:

Listing 1. An immutable `Address` class in Java

```
public final class Address {
    private final String name;
    private final List<String> streets;
```

```
    private final String city;
    private final String state;
    private final String zip;

    public Address(String name, List<String> streets,
                   String city, String state, String zip) {
        this.name = name;
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getName() {
        return name;
    }

    public List<String> getStreets() {
        return Collections.unmodifiableList(streets);
    }

    public String getCity() {
        return city;
    }

    public String getState() {
        return state;
    }

    public String getZip() {
        return zip;
    }
}
```

Note the use of the `Collections.unmodifiableList()` method in Listing 1 to make a defensive copy of the list of streets. You should always use collections to create immutable lists rather than arrays. Although it is possible to copy arrays defensively, it leads to some undesirable side-effects. Consider the code in Listing 2:

Listing 2. `Customer` class that uses arrays instead of collections

```
public class Customer {
    public final String name;
    private final Address[] address;

    public Customer(String name, Address[] address) {
        this.name = name;
        this.address = address;
    }

    public Address[] getAddress() {
        return address.clone();
    }
}
```

The problem with the code in Listing 2 manifests when you try to do anything with the cloned array that comes back from the call to the `getAddress()` method, as shown in Listing 3:

Listing 3. Test that shows correct but unintuitive outcome

```
public static List<String> streets(String... streets) {
    return asList(streets);
}
```

```
public static Address address(List<String> streets,
                              String city, String state, String zip) {
    return new Address(streets, city, state, zip);
}

@Test public void immutability_of_array_references_issue() {
    Address [] addresses = new Address[] {
        address(streets("201 E Washington Ave", "Ste 600"), "Chicago", "IL", "60601")};
    Customer c = new Customer("ACME", addresses);
    assertEquals(c.getAddress()[0].city, addresses[0].city);
    Address newAddress = new Address(
        streets("HackerzRulz Ln"), "Hackerville", "LA", "00000");
    // doesn't work, but fails invisibly
    c.getAddress()[0] = newAddress;

    // illustration that the above unable to change to Customer's address
    assertNotSame(c.getAddress()[0].city, newAddress.city);
    assertSame(c.getAddress()[0].city, addresses[0].city);
    assertEquals(c.getAddress()[0].city, addresses[0].city);
}
```

When you return a cloned array, you protect the underlying array — but you are handing back an array that looks like an ordinary array, meaning that you can change the array's contents. (Even if the variable holding the array is `final`, that applies only to the array reference itself, not to the array's contents.) Using `Collections.unmodifiableList()` (and the family of methods in `Collections` for other types), you receive an object reference that has no mutating methods available.

Cleaner immutable classes

You frequently hear that you should also make your immutable fields private. I disagree with that sentiment, based on hearing someone who has a different but clear vision clarify ingrained assumptions. In a Michael Fogus interview with Clojure creator Rich Hickey (see Resources), Hickey talks about the lack of data-hiding encapsulation of many of the core parts of Clojure. This aspect of Clojure has always bothered me because I'm so steeped in state-based thinking. But then I realized that you don't need to worry about exposing fields if they are immutable. Many of the safeguards we use for encapsulation are really just there to prevent mutation. Once we tease apart those two concepts, a cleaner Java implementation emerges.

Consider the version of the `Address` class in Listing 4:

Listing 4. `Address` class with public, immutable fields

```
public final class Address {
    private final List<String> streets;
    public final String city;
    public final String state;
    public final String zip;

    public Address(List<String> streets, String city, String state, String zip) {
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public final List<String> getStreets() {
        return Collections.unmodifiableList(streets);
    }
}
```

Declaring public `get`*`XXX`*`()` methods for immutable fields only benefits if you want to hide the underlying representation, But that's of dubious benefit in the era of refactoring IDEs that can find such changes trivially. By making the fields both public and immutable, you can directly access them in code without worrying about accidentally changing them.

If you never need to mutate the collection internally, you can cast the embedded list to `unmodifiableList` in the constructor, which allows you to make the `streets` field public and eliminate the need for the `getStreets()` method. As I'll show in the next example, Groovy allows you to create a guard access method like `getStreets()` and yet allow it to appear as a field.

Using immutable public fields seems unnatural at first if you listen to angry monkeys, but their differentness is a benefit: you are unaccustomed to dealing with immutable types in Java, and this looks like a new type, as illustrated in Listing 5:

Listing 5. Unit test of the `Address` class

```
@Test (expected = UnsupportedOperationException.class)
public void address_access_to_fields_but_enforces_immutability() {
    Address a = new Address(
        streets("201 E Randolph St", "Ste 25"), "Chicago", "IL", "60601");
    assertEquals("Chicago", a.city);
    assertEquals("IL", a.state);
    assertEquals("60601", a.zip);
    assertEquals("201 E Randolph St", a.getStreets().get(0));
    assertEquals("Ste 25", a.getStreets().get(1));
    // compiler disallows
    //a.city = "New York";
    a.getStreets().clear();
}
```

Angry monkeys

I first heard this story from Dave Thomas; it subsequently appeared in my book *The Productive Programmer* (see Resources). I don't know if it's true (despite researching it quite a bit), but who cares? It illustrates a point beautifully.

Back in the 1960s, behavioral scientists conducted an experiment in which they put five monkeys in a room with a stepladder and a bunch of bananas hanging from the ceiling.

> The monkeys quickly figured out that they could climb the ladder and eat bananas. Then, every time a monkey got near the stepladder, the scientists doused the entire room in ice-cold water. Soon, none of the monkeys would go near the ladder. Then the scientists replaced one of the doused monkeys with a fresh monkey who had not yet been subject to the experiment. When he made a beeline for the ladder, all the other monkeys beat him up. He didn't know *why* they were beating him up, but he quickly learned: don't go near the ladder. Gradually, the scientists replaced the original monkeys with fresh monkeys until they had a group of monkeys who had never been doused with cold water yet would attack any monkey that approached the ladder.
>
> The point? In software projects, lots of the practices exist because "that's the way we've always done it."

Accessing the public, immutable fields avoids the visual overhead of a series of `get`*`XXX`*`()` calls. Notice also that the compiler won't allow you to assign to one of the primitives, and if you try to call a mutating method on the `street` collection, you'll get an `UnsupportedOperationException` (as caught at the top of the test). Using this style of code is a strong visual indicator that it's an immutable class.

Downsides

One possible disadvantage of the cleaner syntax is the effort entailed in learning this new idiom. But I think it's worth it: it encourages you to think about immutability when creating classes because of an obvious stylistic difference, and it cuts down on unnecessary boilerplate code. But there are some downsides to this coding style in Java (which, to be fair, was never designed to accommodate immutability directly):

- As Glenn Vanderburg pointed out to me, the biggest downside is that the style violates what Bertrand Meyer (creator of the Eiffel programming language) called the Uniform Access Principle: "All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation." In other words, accessing a field shouldn't expose whether it's a field or a method that returns a value. The `Address` class's `getStreets()` method isn't uniform with the other fields. This problem can't really be solved in Java; it is solved in some of the other JVM languages in the way they enable immutability.
- Some frameworks that rely heavily on reflection won't work with this idiom because they require a default constructor.
- Because you create new objects rather than mutate old ones, systems with lots of updates can cause inefficiencies with garbage collection. Languages like Clojure have built-in facilities to make this more efficient with immutable references, which is the default in those languages.

Immutability in Groovy

Building the public-immutable-field version of the `Address` class in Groovy yields a nice clean implementation, shown in Listing 6:

Listing 6. Immutable `Address` class in Groovy

```
class Address {
    def public final List<String> streets;
    def public final city;
    def public final state;
    def public final zip;

    def Address(streets, city, state, zip) {
        this.streets = streets;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    def getStreets() {
        Collections.unmodifiableList(streets);
    }
}
```

As usual, Groovy requires less boilerplate code than Java — and there are other benefits as well. Because Groovy allows you to create properties by using familiar `get`/`set` syntax, you can create a truly protected property for the object reference. Consider the unit tests shown in Listing 7:

Listing 7. Unit tests showing uniform access in Groovy

```
class AddressTest {
    @Test (expected = ReadOnlyPropertyException.class)
    void address_primitives_immutability() {
        Address a = new Address(
            ["201 E Randolph St", "25th Floor"], "Chicago", "IL", "60601")
        assertEquals "Chicago", a.city
        a.city = "New York"
    }

    @Test (expected=UnsupportedOperationException.class)
    void address_list_references() {
        Address a = new Address(
            ["201 E Randolph St", "25th Floor"], "Chicago", "IL", "60601")
        assertEquals "201 E Randolph St", a.streets[0]
        assertEquals "25th Floor", a.streets[1]
        a.streets[0] = "404 W Randoph St"
    }
}
```

Notice that in both cases, the test terminates when an exception is thrown because of a violation of the immutability contract. In Listing 7, however, the `streets` property looks just like the primitives, but it is actually protected via its `getStreets()` method.

Groovy's `@Immutable` annotation

One of the underlying tenets of this series is that functional languages should handle more low-level details for you. A good illustration is the `@Immutable` annotation added in Groovy version 1.7, which makes all the coding in Listing 6 moot. Listing 8 shows a `Client` class that uses this annotation:

Listing 8. Immutable `Client` class

```
@Immutable
class Client {
    String name, city, state, zip
    String[] streets
}
```

By virtue of using the `@Immutable` annotation, this class has the following characteristics:

- It is final.
- Properties automatically have private backing fields with get methods synthesized.
- Any attempts to update properties result in a `ReadOnlyPropertyException`.
- Groovy creates both ordinal and map-based constructors.
- Collection classes are wrapped in appropriate wrappers, and arrays (and other cloneable objects) are cloned.
- Default `equals`, `hashcode`, and `toString` methods are automatically generated.

This annotation provides a lot of bang for the buck! It also acts as you would expect, as shown in Listing 9:

Listing 9. The `@Immutable` annotation handling expected cases correctly

```
@Test (expected = ReadOnlyPropertyException)
void client_object_references_protected() {
    def c = new Client([streets: ["201 E Randolph St", "Ste 25"]])
    c.streets = new ArrayList();
}

@Test (expected = UnsupportedOperationException)
void client_reference_contents_protected() {
    def c = new Client ([streets: ["201 E Randolph St", "Ste 25"]])
    c.streets[0] = "525 Broadway St"
}

@Test
void equality() {
    def d = new Client(
        [name: "ACME", city:"Chicago", state:"IL",
         zip:"60601",
         streets: ["201 E Randolph St", "Ste 25"]])
    def c = new Client(
            [name: "ACME", city:"Chicago", state:"IL",
             zip:"60601",
             streets: ["201 E Randolph St", "Ste 25"]])
    assertEquals(c, d)
    assertEquals(c.hashCode(), d.hashCode())
    assertFalse(c.is(d))
}
```

Trying to replace the object reference yields an `ReadOnlyPropertyException`. And trying to change what one of the encapsulated object references points to generates a `UnsupportedOperationException`. It also creates appropriate `equals` and `hashcode` methods, as shown in the last test — the object contents are the same, but they do not point to the same reference.

Of course, both Scala and Clojure support and encourage immutability and have clean syntax for it, the implications of which will pop up in future installments.

Benefits of immutability

Embracing immutability is high on the list of ways to think like a functional programmer. Although building immutable objects in Java requires a bit more up-front complexity, the downstream simplification forced by this abstraction easily offsets the effort.

Immutable classes make a host of typically worrisome things in Java go away. One of the benefits of switching to a functional mindset is the realization that tests exist to check that changes occur successfully in code. In other words, testing's real purpose is to validate mutation — and the more mutation you have, the more testing is required to make sure you get it right. If you isolate the places where changes occur by severely restricting mutation, you create a much smaller space for errors to occur and have fewer places to test. Because changes only occur upon construction, immutable classes make it trivial to write unit tests. You do not need a copy constructor, and you need never sweat the gory details of implementing a `clone()` method. Immutable objects make good candidates for use as keys in either `Map`s or `Set`s; keys in dictionary collections in Java cannot change value while being used as a key, so immutable objects make great keys.

Immutable objects are also automatically thread-safe and have no synchronization issues. They can also never exist in unknown or undesirable state because of an exception. Because all initialization occurs at construction time, which is atomic in Java, any exception occurs before you have an object instance. Joshua Bloch calls this *failure atomicity*: success or failure based on mutability is forever resolved once the object is constructed (see Resources).

Finally, one of the best features of immutable classes is how well they fit into the *composition* abstraction. In the next installment, I'll start investigating composition and why it is so important in the functional-thinking world.

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.
- Clojure: Clojure is a modern, functional Lisp that runs on the JVM.
- Rich Hickey Q&A: Michael Fogus interviews Clojure creator Rich Hickey.
- Stuart Halloway on Clojure: Learn more about Clojure from this developerWorks podcast.
- Scala: Scala is a modern, functional language on the JVM.
- *The busy Java developer's guide to Scala*: Dig more deeply into Scala in this developerWorks series by Ted Neward.
- *Effective Java*, 2d ed. (Joshua Bloch, Addison Wesley, 2008): Read more about failure atomicity.
- 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

- 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.