developerWorks®

# Java theory and practice: To mutate or not to mutate?

## Immutable objects can greatly simplify your life

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

18 February 2003

Immutable objects have a number of properties that make working with them easier, including relaxed synchronization requirements and the freedom to share and cache object references without concern for data corruption. While immutability may not necessarily make sense for all classes, most programs have at least a few classes that would benefit from being immutable. In this month's *Java theory and practice,* Brian Goetz explains some of the benefits of immutability and some guidelines for constructing immutable classes.

View more content in this series

An immutable object is one whose externally visible state cannot change after it is instantiated. The `String`, `Integer`, and `BigDecimal` classes in the Java class library are examples of immutable objects -- they represent a single value that cannot change over the lifetime of the object.

## Benefits of immutability

Immutable classes, when used properly, can greatly simplify programming. They can only be in one state, so as long as they are properly constructed, they can never get into an inconsistent state. You can freely share and cache references to immutable objects without having to copy or clone them; you can cache their fields or the results of their methods without worrying about the values becoming stale or inconsistent with the rest of the object's state. Immutable classes generally make the best map keys. And they are inherently thread-safe, so you don't have to synchronize access to them across threads.

### Freedom to cache

Because there is no danger of immutable objects changing their value, you can freely cache references to them and be confident that the reference will refer to the same value later. Similarly, because their properties cannot change, you can cache their fields and the results of their methods.

If an object is mutable, you have to exercise some care when storing a reference to it. Consider the code in Listing 1, which queues two tasks for execution by a scheduler. The intent is that the first task would start now and the second task would start in one day.

## Listing 1. A potential problem with a mutable Date object

```
Date d = new Date();
Scheduler.scheduleTask(task1, d);
d.setTime(d.getTime() + ONE_DAY);
scheduler.scheduleTask(task2, d);
```

Because `Date` is mutable, the `scheduleTask` method must be careful to defensively copy the date parameter (perhaps through `clone()`) into its internal data structure. Otherwise, `task1` and `task2` might both execute tomorrow, which is not what was desired. Worse, the internal data structure used by the task scheduler could become corrupt. It is very easy to forget to defensively copy the date parameter when writing a method like `scheduleTask()`. If you do forget, you've created a subtle bug that's not going to show up for a while, and one that will take a long time to track down when it does. An immutable `Date` class would have made this sort of bug impossible.

## Inherent thread safety

Most thread-safety issues arise when multiple threads are trying to modify an object's state concurrently (write-write conflicts) or when one thread is trying to access an object's state while another thread is modifying it (read-write conflicts.) To prevent such conflicts, you must synchronize access to shared objects so that other threads cannot access them while they are in an inconsistent state. This can be difficult to do correctly, requires significant documentation to ensure that the program is extended correctly, and may have negative performance consequences as well. As long as immutable objects are constructed properly (which means not letting the object reference escape from the constructor), they are exempt from the requirement to synchronize access, becuase their state cannot be changed and therefore cannot have write-write or read-write conflicts.

The freedom to share references to immutable objects across threads without synchronization can greatly simplify the process of writing concurrent programs and reduces the number of potential concurrency errors a program could have.

## Safe in the presence of ill-behaved code

Methods that take objects as arguments should not mutate the state of those objects, unless they are explicitly documented to do so or are effectively assuming ownership of that object. When we pass an object to an ordinary method, we generally don't expect that the object will come back changed. However, with mutable objects, this is simply an act of faith. If we pass a `java.awt.Point` to a method such as `Component.setLocation()`, there's nothing stopping `setLocation` from modifying the location of the `Point` we pass in or from storing a reference to that point and changing it later in another method. (Of course, `Component` doesn't do this, because it would be rude, but not all classes are so polite.) Now, the state of our `Point` has changed without our knowledge, with potentially hazardous results -- we still think the point is in one place, when in fact it is in another. However, if `Point` were immutable, then such hostile code would not be able to modify our program state in such a confusing and dangerous way.

## Good keys

Immutable objects make the best `HashMap` or `HashSet` keys. Some mutable objects will change their `hashCode()` value depending on their state (like the `StringHolder` example class in Listing 2). If you use such a mutable object as a `HashSet` key, and then the object changes its state, the `HashSet` implementation will become confused -- the object will still be present if you enumerate the set, but it may not appear to be present if you query the set with `contains()`. Needless to say, this could cause some confusing behavior. The code in Listing 2, which demonstrates this, will print "false," "1," and "moo."

## Listing 2. Mutable StringHolder class, unsuitable for use as a key

```
public class StringHolder {
    private String string;

    public StringHolder(String s) {
        this.string = s;
    }

    public String getString() {
        return string;
    }

    public void setString(String string) {
        this.string = string;
    }

    public boolean equals(Object o) {
        if (this == o)
            return true;
        else if (o == null || !(o instanceof StringHolder))
            return false;
        else {
            final StringHolder other = (StringHolder) o;
            if (string == null)
                return (other.string == null);
            else
                return string.equals(other.string);
        }
    }

    public int hashCode() {
        return (string != null ? string.hashCode() : 0);
    }

    public String toString() {
        return string;
    }

    ...

    StringHolder sh = new StringHolder("blert");
    HashSet h = new HashSet();
    h.add(sh);
    sh.setString("moo");
    System.out.println(h.contains(sh));
    System.out.println(h.size());
    System.out.println(h.iterator().next());
}
```

## When to use immutable classes

Immutable classes are ideal for representing values of abstract data types, such as numbers, enumerated types, or colors. The basic numeric classes in the Java class library, such as `Integer`, `Long`, and `Float`, are immutable, as are other standard numeric types such as `BigInteger` and `BigDecimal`. Classes for representing complex numbers or arbitrary-precision rational numbers would be good candidates for immutability. Depending on your application, even abstract types that contain many discrete values -- such as vectors or matrices -- might be good candidates for implementing as immutable classes.

### The Flyweight pattern

Immutability is what enables the Flyweight pattern, which uses sharing to facilitate using objects to represent large numbers of fine-grained objects efficiently. For example, you might wish to represent each character of a word-processing document or each pixel in an image with an object, but a naive implementation of such a strategy would be prohibitively expensive in memory usage and memory management overhead. The Flyweight pattern employs a factory method to dispense references to immutable fine-grained objects and uses sharing to keep the object count down by only having a single instance of the object corresponding to the letter "a." For more information on the Flyweight pattern, see the classic book, *Design Patterns* (Gamma, et. al; see Resources).

Another good example of immutability in the Java class library is `java.awt.Color`. While colors are generally represented as an ordered set of numeric values in some color representation (such as RGB, HSB, or CMYK), it makes more sense to think of a color as a distinguished value in a color space, rather than an ordered set of individually addressable values, and therefore it makes sense to implement `Color` as an immutable class.

Should we represent objects that are containers for multiple primitive values, such as points, vectors, matrices, or RGB colors, with mutable or immutable objects? The answer is. . . it depends. How are they going to be used? Are they being used primarily to represent multi-dimensional values (like the color of a pixel), or simply as containers for a collection of related properties of some other object (like the height and width of a window)? How often are these properties going to be changed? If they are changed, do the individual component values have meaning within the application on their own?

Events are another good example of candidates for implementation with immutable classes. Events are short-lived, and are often consumed in a different thread than they were created in, so making them immutable has more advantages than disadvantages. Most AWT event classes are not implemented as being strictly immutable, but could be with small modifications. Similarly, in a system that uses some form of messaging to communicate between components, making the message objects immutable is probably sensible.

## Guidelines for writing immutable classes

Writing immutable classes is easy. A class will be immutable if all of the following are true:

- All of its fields are final

- The class is declared final
- The `this` reference is not allowed to escape during construction
- Any fields that contain references to mutable objects, such as arrays, collections, or mutable classes like `Date`:
  - Are private
  - Are never returned or otherwise exposed to callers
  - Are the only reference to the objects that they reference
  - Do not change the state of the referenced objects after construction

The last group of requirements sounds complicated, but it basically means that if you are going to store a reference to an array or other mutable object, you must ensure that your class has exclusive access to that mutable object (because otherwise someone else could change its state) and that you do not modify its state after construction. This complication is necessary to allow immutable objects to store references to arrays, since the Java language has no way to enforce that elements of final arrays are not modified. Note that if array references or other mutable fields are being initialized from arguments passed to a constructor, you must defensively copy the caller-provided arguments or else you can't be sure that you have exclusive access to the array. Otherwise, the caller could modify the state of the array after the call to the constructor. Listing 3 shows the right way -- and wrong way -- to write a constructor for an immutable object that stores a caller-provided array.

## Listing 3. Right and wrong ways to code immutable objects

```
class ImmutableArrayHolder {

  private final int[] theArray;

  // Right way to write a constructor -- copy the array
  public ImmutableArrayHolder(int[] anArray) {
    this.theArray = (int[]) anArray.clone();
  }

  // Wrong way to write a constructor -- copy the reference
  // The caller could change the array after the call to the constructor
  public ImmutableArrayHolder(int[] anArray) {
    this.theArray = anArray;
  }

  // Right way to write an accessor -- don't expose the array reference
  public int getArrayLength() { return theArray.length }
  public int getArray(int n)  { return theArray[n]; }

  // Right way to write an accessor -- use clone()
  public int[] getArray()       { return (int[]) theArray.clone(); }

  // Wrong way to write an accessor -- expose the array reference
  // A caller could get the array reference and then change the contents
  public int[] getArray()       { return theArray }
}
```

With some additional work, it is possible to write immutable classes that use some non-final fields (for example, the standard implementation of `String` uses lazy computation of the `hashCode` value), which may perform better than strictly final classes. If your class represents a value of an abstract type, such as a numeric type or a color, you will want to implement the `hashCode()` and `equals()`

methods too, so that your object will work well as a key in a `HashMap` or `HashSet`. To preserve thread safety, it is important that you not allow the `this` reference to escape from the constructor.

## Infrequently changing data

Some data items remain constant for the lifetime of a program, whereas others change frequently. Constant data are obvious candidates for immutability, and objects with complex and frequently changing states are generally inappropriate candidates for implementation with immutable classes. What about data that changes sometimes, but not often? Is there any way to obtain the convenience and thread-safety benefits of immutability with data that *sometimes* changes?

The `CopyOnWriteArrayList` class, from the `util.concurrent` package, is a good example of how to harness the power of immutability while still permitting occasional modifications. It is ideal for use by classes that support event listeners, such as user interface components. While the list of event listeners can change, it generally changes much less often than events are generated.

`CopyOnWriteArrayList` behaves much like the `ArrayList` class, except that when the list is modified, instead of mutating the underlying array, a new array is created and the old array is discarded. This means that when a caller obtains an iterator, which internally holds a reference to the underlying array, the array referenced by the iterator is effectively immutable and therefore can be traversed without synchronization or risk of concurrent modification. This eliminates the need to either clone the list before traversal or synchronize on the list during traversal, both of which are inconvenient, error-prone, and exact a performance penalty. If traversals are much more frequent than insertions or removals, as is often the case in certain types of situations, `CopyOnWriteArrayList` offers better performance and more convenient access.

## Summary

Immutable objects are much easier to work with than mutable objects. They can only be in one state and so are always consistent, they are inherently thread-safe, and they can be shared freely. There are a whole host of easy-to-commit and hard-to-detect programming errors that are entirely eliminated by using immutable objects, such as failure to synchronize access across threads or failing to clone an array or object before storing a reference to it. When writing a class, it is always worthwhile to ask yourself whether this class could be effectively implemented as an immutable class. You might be surprised at how often the answer is yes.

## Resources

- *Favor Immutability*, Item 13 of Joshua Bloch's *Effective Java Programming Language Guide* (Addison-Wesley, 2001), covers the benefits of immutability in greater detail.
- Mutable or immutable? (*JavaWorld*, December 2002) discusses the benefits of immutability.
- Praxis 63 of Peter Haggar's *Practical Java Programming Language Guide* (Addison-Wesley, 2000) encourages the use of immutable classes.
- Read more about the Flyweight pattern in the "Gang of Four" book *Design Patterns* (Addison-Wesley, 1995).
- Learn more about the CopyOnWriteArrayList class and the rest of the util.concurrent library.
- The November 2002 *Java theory and practice* article, Concurrency made simple (sort of), offers an introduction to the `util.concurrent` package.
- Read the complete *Java theory and practice* series, including details on safe construction techniques and the use and misuse of the `final` keyword.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.

## About the author

### Brian Goetz

Brian Goetz is a software consultant and has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California. See Brian's published and upcoming articles in popular industry publications.