

Item 15: Minimize mutability

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is provided when it is created and is fixed for the lifetime of the object. The Java platform libraries contain many immutable classes, including `String`, the boxed primitive classes, and `BigInteger` and `BigDecimal`. There are many good reasons for this: Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.

To make a class immutable, follow these five rules:

1. **Don't provide any methods that modify the object's state** (known as *mutators*).
2. **Ensure that the class can't be extended.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally accomplished by making the class `final`, but there is an alternative that we'll discuss later.
3. **Make all fields `final`.** This clearly expresses your intent in a manner that is enforced by the system. Also, it is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model* [JLS, 17.5; Goetz06 16].
4. **Make all fields `private`.** This prevents clients from obtaining access to mutable objects referred to by fields and modifying these objects directly. While it is technically permissible for immutable classes to have public `final` fields containing primitive values or references to immutable objects, it is not recommended because it precludes changing the internal representation in a later release (Item 13).
5. **Ensure exclusive access to any mutable components.** If your class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain references to these objects. Never initialize such a field to a client-provided object reference or return the object reference from an accessor. Make *defensive copies* (Item 39) in constructors, accessors, and `readObject` methods (Item 76).

Many of the example classes in previous items are immutable. One such class is `PhoneNumber` in Item 9, which has accessors for each attribute but no corresponding mutators. Here is a slightly more complex example:

```
public final class Complex {
    private final double re;
    private final double im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Accessors with no corresponding mutators
    public double realPart() { return re; }
    public double imaginaryPart() { return im; }

    public Complex add(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }

    public Complex subtract(Complex c) {
        return new Complex(re - c.re, im - c.im);
    }

    public Complex multiply(Complex c) {
        return new Complex(re * c.re - im * c.im,
                           re * c.im + im * c.re);
    }

    public Complex divide(Complex c) {
        double tmp = c.re * c.re + c.im * c.im;
        return new Complex((re * c.re + im * c.im) / tmp,
                           (im * c.re - re * c.im) / tmp);
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Complex))
            return false;
        Complex c = (Complex) o;

        // See page 43 to find out why we use compare instead of ==
        return Double.compare(re, c.re) == 0 &&
            Double.compare(im, c.im) == 0;
    }
}
```

```

@Override public int hashCode() {
    int result = 17 + hashDouble(re);
    result = 31 * result + hashDouble(im);
    return result;
}

private int hashDouble(double val) {
    long longBits = Double.doubleToLongBits(re);
    return (int) (longBits ^ (longBits >>> 32));
}

@Override public String toString() {
    return "(" + re + " + " + im + "i";
}
}

```

This class represents a *complex number* (a number with both real and imaginary parts). In addition to the standard Object methods, it provides accessors for the real and imaginary parts and provides the four basic arithmetic operations: addition, subtraction, multiplication, and division. Notice how the arithmetic operations create and return a new Complex instance rather than modifying this instance. This pattern is used in most nontrivial immutable classes. It is known as the *functional* approach because methods return the result of applying a function to their operand without modifying it. Contrast this to the more common *procedural* or *imperative* approach in which methods apply a procedure to their operand, causing its state to change.

The functional approach may appear unnatural if you're not familiar with it, but it enables immutability, which has many advantages. **Immutable objects are simple.** An immutable object can be in exactly one state, the state in which it was created. If you make sure that all constructors establish class invariants, then it is guaranteed that these invariants will remain true for all time, with no further effort on your part or on the part of the programmer who uses the class. Mutable objects, on the other hand, can have arbitrarily complex state spaces. If the documentation does not provide a precise description of the state transitions performed by mutator methods, it can be difficult or impossible to use a mutable class reliably.

Immutable objects are inherently thread-safe; they require no synchronization. They cannot be corrupted by multiple threads accessing them concurrently. This is far and away the easiest approach to achieving thread safety. In fact, no thread can ever observe any effect of another thread on an immutable object. Therefore, **immutable objects can be shared freely.** Immutable classes should take advantage of this by encouraging clients to reuse existing instances wherever

possible. One easy way to do this is to provide public static final constants for frequently used values. For example, the `Complex` class might provide these constants:

```
public static final Complex ZERO = new Complex(0, 0);
public static final Complex ONE  = new Complex(1, 0);
public static final Complex I    = new Complex(0, 1);
```

This approach can be taken one step further. An immutable class can provide static factories (Item 1) that cache frequently requested instances to avoid creating new instances when existing ones would do. All the boxed primitive classes and `BigInteger` do this. Using such static factories causes clients to share instances instead of creating new ones, reducing memory footprint and garbage collection costs. Opting for static factories in place of public constructors when designing a new class gives you the flexibility to add caching later, without modifying clients.

A consequence of the fact that immutable objects can be shared freely is that you never have to make *defensive copies* (Item 39). In fact, you never have to make any copies at all because the copies would be forever equivalent to the originals. Therefore, you need not and should not provide a `clone` method or *copy constructor* (Item 11) on an immutable class. This was not well understood in the early days of the Java platform, so the `String` class does have a copy constructor, but it should rarely, if ever, be used (Item 5).

Not only can you share immutable objects, but you can share their internals. For example, the `BigInteger` class uses a sign-magnitude representation internally. The sign is represented by an `int`, and the magnitude is represented by an `int` array. The `negate` method produces a new `BigInteger` of like magnitude and opposite sign. It does not need to copy the array; the newly created `BigInteger` points to the same internal array as the original.

Immutable objects make great building blocks for other objects, whether mutable or immutable. It's much easier to maintain the invariants of a complex object if you know that its component objects will not change underneath it. A special case of this principle is that immutable objects make great map keys and set elements: you don't have to worry about their values changing once they're in the map or set, which would destroy the map or set's invariants.

The only real disadvantage of immutable classes is that they require a separate object for each distinct value. Creating these objects can be costly, especially if they are large. For example, suppose that you have a million-bit `BigInteger` and you want to change its low-order bit:

```
BigInteger moby = ...;
moby = moby.flipBit(0);
```

The `flipBit` method creates a new `BigInteger` instance, also a million bits long, that differs from the original in only one bit. The operation requires time and space proportional to the size of the `BigInteger`. Contrast this to `java.util.BitSet`. Like `BigInteger`, `BitSet` represents an arbitrarily long sequence of bits, but unlike `BigInteger`, `BitSet` is mutable. The `BitSet` class provides a method that allows you to change the state of a single bit of a million-bit instance in constant time.

The performance problem is magnified if you perform a multistep operation that generates a new object at every step, eventually discarding all objects except the final result. There are two approaches to coping with this problem. The first is to guess which multistep operations will be commonly required and provide them as primitives. If a multistep operation is provided as a primitive, the immutable class does not have to create a separate object at each step. Internally, the immutable class can be arbitrarily clever. For example, `BigInteger` has a package-private mutable “companion class” that it uses to speed up multistep operations such as modular exponentiation. It is much harder to use the mutable companion class than to use `BigInteger` for all of the reasons outlined earlier, but luckily you don’t have to: the implementors of `BigInteger` did the hard work for you.

The package-private mutable companion class approach works fine if you can accurately predict which complex multistage operations clients will want to perform on your immutable class. If not, then your best bet is to provide a *public* mutable companion class. The main example of this approach in the Java platform libraries is the `String` class, whose mutable companion is `StringBuilder` (and the largely obsolete `StringBuffer`). Arguably, `BitSet` plays the role of mutable companion to `BigInteger` under certain circumstances.

Now that you know how to make an immutable class and you understand the pros and cons of immutability, let’s discuss a few design alternatives. Recall that to guarantee immutability, a class must not permit itself to be subclassed. Typically this is done by making the class `final`, but there is another, more flexible way to do it. The alternative to making an immutable class `final` is to make all of its constructors private or package-private, and to add public *static factories* in place of the public constructors (Item 1).

To make this concrete, here's how `Complex` would look if you took this approach:

```
// Immutable class with static factories instead of constructors
public class Complex {
    private final double re;
    private final double im;

    private Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public static Complex valueOf(double re, double im) {
        return new Complex(re, im);
    }

    ... // Remainder unchanged
}
```

While this approach is not commonly used, it is often the best alternative. It is the most flexible because it allows the use of multiple package-private implementation classes. To its clients that reside outside its package, the immutable class is effectively final because it is impossible to extend a class that comes from another package and that lacks a public or protected constructor. Besides allowing the flexibility of multiple implementation classes, this approach makes it possible to tune the performance of the class in subsequent releases by improving the object-caching capabilities of the static factories.

Static factories have many other advantages over constructors, as discussed in Item 1. For example, suppose that you want to provide a means of creating a complex number based on its polar coordinates. This would be very messy using constructors because the natural constructor would have the same signature that we already used: `Complex(double, double)`. With static factories it's easy. Just add a second static factory with a name that clearly identifies its function:

```
public static Complex valueOfPolar(double r, double theta) {
    return new Complex(r * Math.cos(theta),
                       r * Math.sin(theta));
}
```

It was not widely understood that immutable classes had to be effectively final when `BigInteger` and `BigDecimal` were written, so all of their methods may be

overridden. Unfortunately, this could not be corrected after the fact while preserving backward compatibility. If you write a class whose security depends on the immutability of a `BigInteger` or `BigDecimal` argument from an untrusted client, you must check to see that the argument is a “real” `BigInteger` or `BigDecimal`, rather than an instance of an untrusted subclass. If it is the latter, you must defensively copy it under the assumption that it might be mutable (Item 39):

```
public static BigInteger safeInstance(BigInteger val) {
    if (val.getClass() != BigInteger.class)
        return new BigInteger(val.toByteArray());
    return val;
}
```

The list of rules for immutable classes at the beginning of this item says that no methods may modify the object and that all its fields must be final. In fact these rules are a bit stronger than necessary and can be relaxed to improve performance. In truth, no method may produce an *externally visible* change in the object’s state. However, some immutable classes have one or more nonfinal fields in which they cache the results of expensive computations the first time they are needed. If the same value is requested again, the cached value is returned, saving the cost of recalculation. This trick works precisely because the object is immutable, which guarantees that the computation would yield the same result if it were repeated.

For example, `PhoneNumber`’s `hashCode` method (Item 9, page 49) computes the hash code the first time it’s invoked and caches it in case it’s invoked again. This technique, an example of *lazy initialization* (Item 71), is also used by `String`.

One caveat should be added concerning serializability. If you choose to have your immutable class implement `Serializable` and it contains one or more fields that refer to mutable objects, you must provide an explicit `readObject` or `readResolve` method, or use the `ObjectOutputStream.writeUnshared` and `ObjectInputStream.readUnshared` methods, even if the default serialized form is acceptable. Otherwise an attacker could create a mutable instance of your not-quite-immutable class. This topic is covered in detail in Item 76.

To summarize, resist the urge to write a `set` method for every `get` method. **Classes should be immutable unless there’s a very good reason to make them mutable.** Immutable classes provide many advantages, and their only disadvantage is the potential for performance problems under certain circumstances. You should always make small value objects, such as `PhoneNumber` and `Complex`, immutable. (There are several classes in the Java platform libraries, such as

`java.util.Date` and `java.awt.Point`, that should have been immutable but aren't.) You should seriously consider making larger value objects, such as `String` and `BigInteger`, immutable as well. You should provide a public mutable companion class for your immutable class *only* once you've confirmed that it's necessary to achieve satisfactory performance (Item 55).

There are some classes for which immutability is impractical. **If a class cannot be made immutable, limit its mutability as much as possible.** Reducing the number of states in which an object can exist makes it easier to reason about the object and reduces the likelihood of errors. Therefore, **make every field final unless there is a compelling reason to make it nonfinal.**

Constructors should create fully initialized objects with all of their invariants established. Don't provide a public initialization method separate from the constructor or static factory unless there is a compelling reason to do so. Similarly, don't provide a "reinitialize" method that enables an object to be reused as if it had been constructed with a different initial state. Such methods generally provide little if any performance benefit at the expense of increased complexity.

The `TimerTask` class exemplifies these principles. It is mutable, but its state space is kept intentionally small. You create an instance, schedule it for execution, and optionally cancel it. Once a timer task has run to completion or has been canceled, you may not reschedule it.

A final note should be added concerning the `Complex` class in this item. This example was meant only to illustrate immutability. It is not an industrial-strength complex number implementation. It uses the standard formulas for complex multiplication and division, which are not correctly rounded and provide poor semantics for complex NaNs and infinities [Kahan91, Smith62, Thomas94].