Many of the example classes in previous items are immutable. One such class is `PhoneNumber` in Item 8, which has accessors for each attribute but no corresponding mutators. Here is a slightly more complex example:

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

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

    // Accessors with no corresponding mutators
    public float realPart()      { return re; }
    public float 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) {
        float 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);
    }

    public boolean equals(Object o) {
       if (o == this)
           return true;
       if (!(o instanceof Complex))
           return false;
       Complex c = (Complex)o;
       return (Float.floatToIntBits(re) ==    // See page 33 to
               Float.floatToIntBits(c.re)) && // find out why
              (Float.floatToIntBits(im) ==    // floatToIntBits
               Float.floatToIntBits(c.im));   // is used.
    }

    public int hashCode() {
        int result = 17 + Float.floatToIntBits(re);
        result = 37*result + Float.floatToIntBits(im);
        return result;
    }

    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* 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 fre-