CHAPTER 4

Classes and Interfaces

CLASSES and interfaces lie at the heart of the Java programming language. They are its basic units of abstraction. The language provides many powerful elements that you can use to design classes and interfaces. This chapter contains guidelines to help you make the best use of these elements so that your classes and interfaces are usable, robust, and flexible.

Item 13: Minimize the accessibility of classes and members

The single most important factor that distinguishes a well-designed module from a poorly designed one is the degree to which the module hides its internal data and other implementation details from other modules. A well-designed module hides all of its implementation details, cleanly separating its API from its implementation. Modules then communicate with one another only through their APIs and are oblivious to each others’ inner workings. This concept, known as information hiding or encapsulation, is one of the fundamental tenets of software design [Parnas72].

Information hiding is important for many reasons, most of which stem from the fact that it decouples the modules that comprise a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation. This speeds up system development because modules can be developed in parallel. It eases the burden of maintenance because modules can be understood quickly and debugged with little fear of harming other modules. While information hiding does not, in and of itself, cause good performance, it enables effective performance tuning. Once a system is complete and profiling has determined which modules are causing performance problems (Item 42), those modules can be optimized without affecting the correctness of other modules. Information hiding increases software reuse because individual modules do not depend on one another and frequently prove useful in contexts other than the ones for which they
were developed. Finally, information hiding decreases the risk in building large systems; individual modules may prove successful even if the system does not.

The Java programming language has many facilities to aid information hiding. One such facility is the *access control* mechanism [JLS, 6.6], which determines the *accessibility* of classes, interfaces, and members. The accessibility of an entity is determined by the location where it is declared and by which, if any, of the access modifiers (*private*, *protected*, and *public*) is present in the entity’s declaration. Proper use of these modifiers is essential to information hiding.

The rule of thumb is that you should **make each class or member as inaccessible as possible**. In other words, you should use the lowest possible access level consistent with the proper functioning of the software that you are writing.

For top-level (non-nested) classes and interfaces, there are only two possible access levels: *package-private* and *public*. If you declare a top-level class or interface with the *public* modifier, it will be public; otherwise, it will be package-private. If a top-level class or interface can be made package-private, it should be. By making it package-private, you make it part of the package’s implementation rather than its exported API, and you can modify it, replace it, or eliminate it in a subsequent release without fear of harming existing clients. If you make it public, you are obligated to support it forever to maintain compatibility.

If a package-private top-level class or interface is used only from within a single class, you should consider making it a private nested class (or interface) of the class in which it is used (Item 22). This further reduces its accessibility. It is, however, not as important to do this as it is to make an unnecessarily public class package-private, because a package-private class is already part of the package’s implementation rather than its API.

For members (fields, methods, nested classes, and nested interfaces) there are four possible access levels, listed here in order of increasing accessibility:

- **private**—The member is accessible only inside the top-level class where it is declared.

- **package-private**—The member is accessible from any class in the package where it is declared. Technically known as *default* access, this is the access level you get if no access modifier is specified.

- **protected**—The member is accessible from subclasses of the class where it is declared (subject to a few restrictions [JLS, 6.6.2]) and from any class in the package where it is declared.

- **public**—The member is accessible from anywhere.
ITEM 13: MINIMIZE THE ACCESSIBILITY OF CLASSES AND MEMBERS

After carefully designing your class’s public API, your reflex should be to make all other members private. Only if another class in the same package really needs to access a member should you remove the private modifier, making the member package-private. If you find yourself doing this often, you should reexamine the design of your system to see if another decomposition might yield classes that are better decoupled from one another. That said, both private and package-private members are part of a class’s implementation and do not normally impact its exported API. These fields can, however, “leak” into the exported API if the class implements Serializable (Item 59, Item 60).

For members of public classes, a huge increase in accessibility occurs when the access level goes from package-private to protected. A protected member is part of the class’s exported API and must be supported forever. Furthermore, a protected member of an exported class represents a public commitment to an implementation detail (Item 17). The need for protected members should be relatively rare.

There is one rule that restricts your ability to reduce the accessibility of methods. If a method overrides a superclass method, it is not permitted to have a lower access level in the subclass than it does in the superclass [JLS, 8.4.8.3]. This is necessary to ensure that an instance of the subclass is usable anywhere that an instance of the superclass is usable. If you violate this rule, the compiler will generate an error message when you try to compile the subclass. A special case of this rule is that if a class implements an interface, all of the class methods that are also present in the interface must be declared public. This is so because all members of an interface are implicitly public [JLS, 9.1.5].

Public classes should rarely, if ever, have public fields. If a field is nonfinal or is a final reference to a mutable object, you give up the ability to limit the values that may be stored in the field by making it public. You also give up the ability to take any action when the field is modified. A simple consequence is that classes with public mutable fields are not thread-safe. Even if a field is final and does not refer to a mutable object, by making the field public you give up the flexibility to switch to a new internal data representation in which the field does not exist.

There is one exception to the rule that public classes should not have public fields. Classes are permitted to expose constants via public static final fields. By convention, such fields have names consisting of capital letters, with words separated by underscores (Item 43). It is critical that these fields contain either primitive values or references to immutable objects (Item 15). A final field containing a reference to a mutable object has all the disadvantages of a nonfinal
field. While the reference cannot be modified, the referenced object can be modified—with disastrous results.

Note that a nonzero-length array is always mutable, so it is nearly always wrong to have public static final array field. If a class has such a field, clients will be able to modify the contents of the array. This is a frequent source of security holes:

```java
// Potential security hole!
public static final Type[] VALUES = { ... };
```

You can replace the public array by a private array and a public immutable list:

```java
private static final Type[] PRIVATE_VALUES = { ... };

public static final List<Type> VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Alternatively, you can replace the public array with a public method that returns a copy of a private array:

```java
private static final Type[] PRIVATE_VALUES = { ... };

public static final Type[] values() {
    return PRIVATE_VALUES.clone();
}
```

To choose between these alternatives, think about what the client is likely to do with the result.

To summarize, you should always reduce accessibility as much as possible. After carefully designing a minimal public API, you should prevent any stray classes, interfaces, or members from becoming a part of the API. With the exception of public static final fields, public classes should have no public fields. Ensure that objects referenced by public static final fields are immutable.
**Item 14: In public classes prefer accessor methods to public fields**

Occasionally, you may be tempted to write degenerate classes that serve no purpose other than to group instance fields:

```java
// Degenerate classes like this should not be public!
class Point {
    public float x;
    public float y;
}
```

Because the data fields of such classes are accessed directly, these classes do not offer the benefits of *encapsulation* (Item 13). You can’t change the representation without changing the API, you can’t enforce invariants, and you can’t take any auxiliary action when a field is accessed. Hard-line object-oriented programmers feel that such classes are anathema and should be replaced by classes with private fields and public accessor methods:

```java
// Encapsulation of data by accessor methods
class Point {
    private float x;
    private float y;

    public Point(float x, float y) {
        this.x = x;
        this.y = y;
    }

    public float getX() { return x; }
    public float getY() { return y; }

    public void setX(float x) { this.x = x; }
    public void setY(float y) { this.y = y; }
}
```

Certainly, the hard-liners are correct when it comes to public classes: If a class is accessible outside the confines of its package, the prudent programmer will provide accessor methods to preserve the flexibility to change the class’s internal representation. If a public class were to expose its data fields, all hope of changing its representation would be lost, as client code for public classes can be distributed all over the known universe.

If, however, a class is package-private, or is a private nested class, there is nothing inherently wrong with directly exposing its data fields—assuming they do
an adequate job of describing the abstraction provided by the class. This approach generates less visual clutter than the accessor-method approach, both in the class definition and in the client code that uses it. While the client code is tied to the class’s internal representation, this code is confined to the package containing the class. If a change in representation becomes desirable, you can make the change without touching any code outside the package. In the case of a private nested class, the scope of the change is further restricted to the enclosing class.

Several classes in the Java platform libraries violate the advice that public classes should not expose fields directly. Prominent examples include the Point and Dimension classes in the java.awt package. Rather than examples to be emulated, these classes should be regarded as cautionary tales. As described in Item 42, the decision to expose the internals of the Dimension class resulted in a serious performance problem that is still with us today.

While it’s never a good idea for a public class to expose fields directly, it is less harmful if the fields are immutable. You can’t change the representation of such a class without changing its API, and you can’t take auxiliary actions when a field is read, but you can enforce invariants. For example, this class guarantees that each instance represents a valid time:

```java
// Public class with exposed immutable fields - questionable
public final class Time {
    private static final int HOURS_PER_DAY = 24;
    private static final int MINUTES_PER_HOUR = 60;

    public final int hour;
    public final int minute;

    public Time(int hour, int minute) {
        if (hour < 0 || hour >= HOURS_PER_DAY)
            throw new IllegalArgumentException("Hour: " + hour);
        if (minute < 0 || minute >= MINUTES_PER_HOUR)
            throw new IllegalArgumentException("Min: " + minute);
        this.hour = hour;
        this.minute = minute;
    }
    ... // Remainder omitted
}
```

In summary, public classes should never expose mutable fields. It is less harmful, though still questionable, for public classes to expose immutable fields. It is sometimes desirable for package-private and private classes to expose fields, whether mutable or immutable.
**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 primitive wrapper 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** (known as *mutators*).
2. **Ensure that no methods may be overridden**. This prevents careless or malicious subclasses from compromising the immutable behavior of the class. Preventing method overrides is generally done by making the class final, but there are alternatives which we’ll discuss later.
3. **Make all fields final**. This clearly expresses your intentions 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].
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 nor return the object reference from an accessor. Make *defensive copies* (Item 29) in constructors, accessors, and `readObject` methods (Item 61).
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:

```java
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);
    }

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Complex))
            return false;
        Complex c = (Complex)o;
        return (double.doubleToIntBits(re) == // See page 42 to
                double.doubleToIntBits(c.re)) && // find out why
               (double.doubleToIntBits(im) == // doubleToIntBits
                double.doubleToIntBits(c.im)); // is used.
    }
}
```
public int hashCode() {
    int result = 17 + double.doubleToIntBits(re);
    result = 37 * result + double.doubleToIntBits(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 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 fre-
consequently used values. For example, the `Complex` class might provide the following constants:

```java
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 that cache frequently requested instances and avoid creating new instances whenever a preexisting instance is requested. The `BigInteger` and `Boolean` classes both have such static factories. Using such static factories causes clients to share preexisting instances rather than creating new ones, reducing memory footprint and garbage collection costs.

A consequence of the fact that immutable objects can be shared freely is that you never have to make defensive copies (Item 29). 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:

```java
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 `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 any of its methods to be overridden. In addition to making a class final, there are two other ways to guarantee this. One way is to make each method of the class, but not the class itself, final. The sole advantage of this approach is that it allows programmers to extend the class by adding new methods built atop the old ones. This approach is not recommended, as it is cleaner and almost as effective to implement the new methods as static methods in a separate, noninstantiable utility class (Item 4).

A second 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:

```java
// 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:

```java
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
ITEM 15: MINIMIZE MUTABILITY

overridden. Unfortunately, this could not be corrected after the fact while preserv-
ing upward 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 defen-
sively copy it under the assumption that it might be mutable (Item 29):

```java
public void foo(BigInteger b) {
    if (b.getClass() != BigInteger.class)
        b = new BigInteger(b.toByteArray());
    ...
}
```

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 53), is also used by
String. No synchronization is necessary, as it is not a problem if the hash value is
recalculated. Here is the general idiom to return a lazily computed function of an
immutable object. Note that a volatile field is used to cache the function’s value,
esential for proper operation in the presence of multiple threads (Item XX):

```java
// Cached, lazily initialized function of an immutable object
private volatile Foo cachedFooVal = UNLIKELY_FOO_VALUE;

public Foo foo() {
    Foo result = cachedFooVal;
    if (result == UNLIKELY_FOO_VALUE)
        result = cachedFooVal = fooVal();
    return result;
}

// Private helper function to calculate our foo value
private Foo fooVal() { ... }
```
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 61.

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 42).

There are some classes for which immutability is impractical, including “process classes” such as `Thread` and `TimerTask`. **If a class cannot be made immutable, you should still 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 constructors should create fully initialized objects with all of their invariants established. You should not provide a public initialization method separate from the constructor unless there is an extremely good reason to do so. Similarly, you should not provide a “reinitialize” method, which enables an object to be reused as if it had been constructed with a different initial state. A reinitialize method generally provides 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 cancelled, 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].
Item 16: Favor composition over inheritance

Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is safe to use inheritance within a package, where the subclass and the superclass implementation are under the control of the same programmers. It is also safe to use inheritance when extending classes specifically designed and documented for extension (Item 17). Inheriting from ordinary concrete classes across package boundaries, however, is dangerous. As a reminder, this book uses the word “inheritance” to mean implementation inheritance (when one class extends another). The problems discussed in this item do not apply to interface inheritance (when a class implements an interface or where one interface extends another).

Unlike method invocation, inheritance violates encapsulation [Snyder86]. In other words, a subclass depends on the implementation details of its superclass for its proper function. The superclass’s implementation may change from release to release, and if it does, the subclass may break, even though its code has not been touched. As a consequence, a subclass must evolve in tandem with its superclass, unless the superclass’s authors have designed and documented it specifically for the purpose of being extended.

To make this concrete, let’s suppose we have a program that uses a HashSet. To tune the performance of our program, we need to query the HashSet as to how many elements have been added since it was created (not to be confused with its current size, which goes down when an element is removed). To provide this functionality, we write a HashSet variant that keeps count of the number of attempted element insertions and exports an accessor for this count. The HashSet class contains two methods capable of adding elements, add and addAll, so we override both of these methods:

```java
// Broken - Inappropriate use of inheritance!
public class InstrumentedHashSet<E> extends HashSet<E> {
    // The number of attempted element insertions
    private int addCount = 0;

    public InstrumentedHashSet() {
    }

    public InstrumentedHashSet(Collection<? extends E> c) {
        super(c);
    }
```
public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
}

@Override public boolean add(E e) {
    addCount++;
    return super.add(e);
}

@Override public boolean addAll(Collection<? extends E> c) {
    addCount += c.size();
    return super.addAll(c);
}

public int getAddCount() {
    return addCount;
}
}

This class looks reasonable, but it doesn’t work. Suppose we create an instance and add three elements using the addAll method:

InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>());
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));

We would expect the getAddCount method to return three at this point, but it returns six. What went wrong? Internally, HashSet’s addAll method is implemented on top of its add method, although HashSet, quite reasonably, does not document this implementation detail. The addAll method in InstrumentedHashSet added three to addCount and then invoked HashSet’s addAll implementation using super.addAll. This in turn invoked the add method, as overridden in InstrumentedHashSet, once for each element. Each of these three invocations added one more to addCount, for a total increase of six: each element added with the addAll method is double-counted.

We could “fix” the subclass by eliminating its override of the addAll method. While the resulting class would work, it would depend for its proper function on the fact that HashSet’s addAll method is implemented on top of its add method. This “self-use” is an implementation detail, not guaranteed to hold in all implementations of the Java platform and subject to change from release to release. Therefore, the resulting InstrumentedHashSet class would be fragile.

It would be slightly better to override the addAll method to iterate over the specified collection, calling the add method once for each element. This would
guarantee the correct result whether or not HashSet’s `addAll` method were implemented atop its `add` method, because `HashSet`’s `addAll` implementation would no longer be invoked. This technique, however, does not solve all our problems. It amounts to reimplementing superclass methods that may or may not result in self-use, which is difficult, time-consuming, and error prone. Additionally, it isn’t always possible, as some methods cannot be implemented without access to private fields inaccessible to the subclass.

A related cause of fragility in subclasses is that their superclass can acquire new methods in subsequent releases. Suppose a program depends for its security on the fact that all elements inserted into some collection satisfy some predicate. This can be guaranteed by subclassing the collection and overriding each method capable of adding an element to ensure that the predicate is satisfied before adding the element. This works fine until a new method capable of inserting an element is added to the superclass in a subsequent release. Once this happens, it becomes possible to add an “illegal” element to an instance of the subclass merely by invoking the new method, which is not overridden in the subclass. This is not a purely theoretical problem. Several security holes of this nature had to be fixed when `Hashtable` and `Vector` were retrofitted to participate in the Collections Framework.

Both of the above problems stem from overriding methods. You might think that it is safe to extend a class if you merely add new methods and refrain from overriding existing methods. While this sort of extension is much safer, it is not without risk. If the superclass acquires a new method in a subsequent release and you have the bad luck to have given the subclass a method with the same signature and a different return type, your subclass will no longer compile [JLS, 8.4.8.3]. If you’ve given the subclass a method with the same signature and return type as the new superclass method, then you’re now overriding it, so you’re subject to the two problems described above. Furthermore, it is doubtful that your method will fulfill the contract of the new superclass method, as that contract had not yet been written when you wrote the subclass method.

Luckily, there is a way to avoid all of the problems described earlier. Instead of extending an existing class, give your new class a private field that references an instance of the existing class. This design is called composition because the existing class becomes a component of the new one. Each instance method in the new class invokes the corresponding method on the contained instance of the existing class and returns the results. This is known as forwarding, and the methods in the new class are known as forwarding methods. The resulting class will be rock solid, with no dependencies on the implementation details of the existing
class. Even adding new methods to the existing class will have no impact on the new class. To make this concrete, here's a replacement for InstrumentedHashSet that uses the composition-and-forwarding approach:

```java
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> implements Set<E> {
    private final Set<E> s;
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        this.s = s;
    }

    public boolean add(E e) {
        addCount++;
        return s.add(e);
    }

    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }

    // Forwarding methods
    public void clear() { s.clear(); }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty() { return s.isEmpty(); }
    public int size() { return s.size(); }
    public Iterator<E> iterator() { return s.iterator(); }
    public boolean remove(Object o) { return s.remove(o); }
    public boolean containsAll(Collection<?> c) {
        return s.containsAll(c);
    }
    public boolean removeAll(Collection<?> c) {
        return s.removeAll(c);
    }
    public boolean retainAll(Collection<?> c) {
        return s.retainAll(c);
    }
    public Object[] toArray() { return s.toArray(); }
    public <T> T[] toArray(T[] a) { return s.toArray(a); }
    public boolean equals(Object o) { return s.equals(o); }
    public int hashCode() { return s.hashCode(); }
    public String toString() { return s.toString(); }
}
```
The design of the InstrumentedSet class is enabled by the existence of the Set interface, which captures the functionality of the HashSet class. Besides being robust, this design is extremely flexible. The InstrumentedSet class implements the Set interface and has a single constructor whose argument is also of type Set. In essence, the class transforms one Set into another, adding the instrumentation functionality. Unlike the inheritance-based approach, which works only for a single concrete class and requires a separate constructor for each supported constructor in the superclass, the wrapper class can be used to instrument any Set implementation and will work in conjunction with any preexisting constructor:

```java
Set<String> s1 = new InstrumentedSet(new TreeSet<String>(list));
Set<E> s2 = new InstrumentedSet<E>(new HashSet<E>(capacity));
```

The InstrumentedSet class can even be used to temporarily instrument a set instance that has already been used without instrumentation:

```java
static void f(Set<E> set) {
    InstrumentedSet<E> instSet = new InstrumentedSet<E>(set);
    ... // Within this method use instSet instead of set
}
```

The InstrumentedSet class is known as a wrapper class because each InstrumentedSet instance contains ("wraps") another Set instance. This is also known as the Decorator pattern [Gamma95, p. 175], because the InstrumentedSet class "decorates" a set by adding instrumentation. Sometimes the combination of composition and forwarding is erroneously referred to as delegation. Technically, it’s not delegation unless the wrapper object passes itself to the wrapped object [Gamma95, p. 20].

The disadvantages of wrapper classes are few. One caveat is that wrapper classes are not suited for use in callback frameworks, wherein objects pass self-references to other objects for subsequent invocations ("callbacks"). Because a wrapped object doesn’t know of its wrapper, it passes a reference to itself (this) and callbacks elude the wrapper. This is known as the SELF problem [Lieberman86]. Some people worry about the performance impact of forwarding method invocations or the memory footprint impact of wrapper objects. Neither turn out to have much impact in practice. It is tedious to write forwarding methods, but this is tempered by the fact that you have to write only one constructor.

Inheritance is appropriate only in circumstances where the subclass really is a subtype of the superclass. In other words, a class B should extend a class A only if
an “is-a” relationship exists between the two classes. If you are tempted to have a class \( B \) extend a class \( A \), ask yourself the question: “Is every \( B \) really an \( A \)?” If you cannot truthfully answer yes to this question, \( B \) should not extend \( A \). If the answer is no, it is often the case that \( B \) should contain a private instance of \( A \) and expose a smaller and simpler API: \( A \) is not an essential part of \( B \), merely a detail of its implementation.

There are a number of obvious violations of this principle in the Java platform libraries. For example, a stack is not a vector, so \( \text{Stack} \) should not extend \( \text{Vector} \). Similarly, a property list is not a hash table so \( \text{Properties} \) should not extend \( \text{Hashtable} \). In both cases, composition would have been appropriate.

If you use inheritance where composition is appropriate, you needlessly expose implementation details. The resulting API ties you to the original implementation, forever limiting the performance of your class. More seriously, by exposing the internals you let the client access them directly. At the very least, this can lead to confusing semantics. For example, if \( p \) refers to a \( \text{Properties} \) instance, then \( p \).\text{getProperty(key)} \) may yield different results from \( p \).\text{get(key)} \): the former method takes defaults into account, while the latter method, which is inherited from \( \text{Hashtable} \), does not. Most seriously, the client may be able to corrupt invariants of the subclass by modifying the superclass directly. In the case of \( \text{Properties} \), the designers intended that only strings be allowed as keys and values, but direct access to the underlying \( \text{Hashtable} \) allows this invariant to be violated. Once this invariant is violated, it is no longer possible to use other parts of the \( \text{Properties} \) API (\text{load} and \text{store}). By the time this problem was discovered, it was too late to correct it because clients depended on the use of nonstring keys and values.

There is one last set of questions you should ask yourself before deciding to use inheritance in place of composition. Does the class that you contemplate extending have any flaws in its API? If so, are you comfortable propagating those flaws into your class’s API? Inheritance propagates any flaws in the superclass’s API, while composition lets you design a new API that hides these flaws.

To summarize, inheritance is powerful, but it is problematic because it violates encapsulation. It is appropriate only when a genuine subtype relationship exists between the subclass and the superclass. Even then, inheritance may lead to fragility if the subclass is in a different package from the superclass and the superclass is not designed for extension. To avoid this fragility, use composition and forwarding instead of inheritance, especially if an appropriate interface to implement a wrapper class exists. Not only are wrapper classes more robust than subclasses, they are also more powerful.
Item 17: Design and document for inheritance or else prohibit it

Item 16 alerted you to the dangers of subclassing a “foreign” class that was not designed and documented for inheritance. So what does it mean for a class to be designed and documented for inheritance?

First, the class must document precisely the effects of overriding any method. In other words, the class must document its self-use of overridable methods: For each public or protected method or constructor, the documentation must indicate which overridable methods the method or constructor invokes, in what sequence, and how the results of each invocation affect subsequent processing. (By overridable, we mean nonfinal and either public or protected.) More generally, a class must document any circumstances under which it might invoke an overridable method. For example, invocations might come from background threads or static initializers.

By convention, a method that invokes overridable methods contains a description of these invocations at the end of its documentation comment. The description begins with the phrase, “This implementation.” This phrase should not be taken to indicate that the behavior may change from release to release. It connotes that the description concerns the inner workings of the method. Here’s an example, copied from the specification for java.util.AbstractCollection:

```java
public boolean remove(Object o)
```

Removes a single instance of the specified element from this collection, if it is present (optional operation). More formally, removes an element e such that `(o==null ? e==null : o.equals(e))`, if the collection contains one or more such elements. Returns true if the collection contained the specified element (or equivalently, if the collection changed as a result of the call).

This implementation iterates over the collection looking for the specified element. If it finds the element, it removes the element from the collection using the iterator’s remove method. Note that this implementation throws an UnsupportedOperationException if the iterator returned by this collection’s iterator method does not implement the remove method.

This documentation leaves no doubt that overriding the iterator method will affect the behavior of the remove method. Furthermore, it describes exactly how the behavior of the Iterator returned by the iterator method will affect the behavior of the remove method. Contrast this to the situation in Item 16, where
the programmer subclassing HashSet simply could not say whether overriding the add method would affect the behavior of the addAll method.

But doesn’t this violate the dictum that good API documentation should describe what a given method does and not how it does it? Yes it does! This is an unfortunate consequence of the fact that inheritance violates encapsulation. To document a class so that it can be safely subclassed, you must describe implementation details that should otherwise be left unspecified.

Design for inheritance involves more than just documenting patterns of self-use. To allow programmers to write efficient subclasses without undue pain, a class may have to provide hooks into its internal workings in the form of judiciously chosen protected methods or, in rare instances, protected fields. For example, consider the removeRange method from java.util.AbstractList:

```java
protected void removeRange(int fromIndex, int toIndex)
```

Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. Shifts any succeeding elements to the left (reduces their index). This call shortens the ArrayList by (toIndex - fromIndex) elements. (If toIndex == fromIndex, this operation has no effect.)

This method is called by the clear operation on this list and its sublists. Overriding this method to take advantage of the internals of the list implementation can substantially improve the performance of the clear operation on this list and its subLists.

This implementation gets a list iterator positioned before fromIndex and repeatedly calls ListIterator.next followed by ListIterator.remove, until the entire range has been removed. Note: If ListIterator.remove requires linear time, this implementation requires quadratic time.

Parameters:

- fromIndex: index of first element to be removed.
- toIndex: index after last element to be removed.

This method is of no interest to end users of a List implementation. It is provided solely to make it easy for subclasses to provide a fast clear method on sublists. In the absence of the removeRange method, subclasses would have to make do with quadratic performance when the clear method was invoked on sublists or rewrite the entire subList mechanism from scratch—not an easy task!
So how do you decide what protected members to expose when you design a class for inheritance? Unfortunately, there is no magic bullet. The best you can do is to think hard, take your best guess, and then test it by writing some subclasses. You should expose as few protected members as possible because each one represents a commitment to an implementation detail. On the other hand, you must not expose too few, as a missing protected member can render a class practically unusable for inheritance.

The importance of testing extensible classes by writing subclasses cannot be overemphasized. If you omit a crucial protected member, trying to write subclasses will make the omission painfully obvious. Conversely, if none of your subclasses use a protected member, make it private. Experience shows that three subclasses are generally sufficient to test an extensible class.

When you design for inheritance a class that is likely to achieve wide use, realize that you are committing forever to the self-use patterns that you document and to the implementation decisions implicit in its protected methods and fields. These commitments can make it difficult or impossible to improve the performance or functionality of the class in a subsequent release. Therefore, you must test your class by writing subclasses before you release it.

Also, note that the special documentation required for inheritance clutters up normal documentation, which is designed for programmers who create instances of your class and invoke methods on them. As of this writing, there is little in the way of tools or commenting conventions to separate ordinary API documentation from information of interest only to programmers implementing subclasses.

There are a few more restrictions that a class must obey to allow inheritance. Constructors must not invoke overridable methods, directly or indirectly. If you violate this rule, program failure will result. The superclass constructor runs before the subclass constructor, so the overriding method in the subclass will get invoked before the subclass constructor has run. If the overriding method depends on any initialization performed by the subclass constructor, the method will not behave as expected. To make this concrete, here’s a class that violates this rule:

```java
public class Super {
    // Broken - constructor invokes overridable method
    public Super() {
        m();
    }

    public void m() {
    }
}
```
Here’s a subclass that overrides \texttt{m}, which is erroneously invoked by \texttt{Super}’s sole constructor:

```java
final class Sub extends Super {
    private final Date date; // Blank final, set by constructor
    Sub() {
        date = new Date();
    }
    // Overrides \texttt{Super.m}, invoked by the constructor \texttt{Super()}
    public void m() {
        System.out.println(date);
    }
    public static void main(String[] args) {
        Sub s = new Sub();
        s.m();
    }
}
```

You might expect this program to print out the date twice, but it prints out \texttt{null} the first time because the method \texttt{m} is invoked by the constructor \texttt{Super()} before the constructor \texttt{Sub()} has a chance to initialize the \texttt{date} field. Note that this program observes a final field in two different states.

The \texttt{Cloneable} and \texttt{Serializable} interfaces present special difficulties when designing for inheritance. It is generally not a good idea for a class designed for inheritance to implement either of these interfaces, as they place a substantial burden on programmers who extend the class. There are, however, special actions that you can take to allow subclasses to implement these interfaces without mandating that they do so. These actions are described in Item 11 and Item 59.

If you do decide to implement \texttt{Cloneable} or \texttt{Serializable} in a class designed for inheritance, you should be aware that because the \texttt{clone} and \texttt{readObject} methods behave a lot like constructors, a similar restriction applies: \textbf{Neither \texttt{clone} nor \texttt{readObject} may invoke an overridable method, directly or indirectly.} In the case of the \texttt{readObject} method, the overriding method will run before the subclass’s state has been deserialized. In the case of the \texttt{clone} method, the overriding method will run before the subclass’s \texttt{clone} method has a chance to fix the clone’s state. In either case, a program failure is likely to follow. In the case of the \texttt{clone} method, the failure can do damage to the object being cloned as well as to the clone itself.
Finally, if you decide to implement `Serializable` in a class designed for inheritance and the class has a `readResolve` or `writeReplace` method, you must make the `readResolve` or `writeReplace` method protected rather than private. If these methods are private, they will be silently ignored by subclasses. This is one more case where an implementation detail becomes part of a class’s API to permit inheritance.

By now, it should be apparent that designing a class for inheritance places substantial limitations on the class. This is not a decision to be undertaken lightly. There are some situations where it is clearly the right thing to do, such as abstract classes, including skeletal implementations of interfaces (Item 18). There are other situations where it is clearly the wrong thing to do, such as immutable classes (Item 15).

But what about ordinary concrete classes? Traditionally, they are neither final nor designed and documented for subclassing, but this state of affairs is dangerous. Each time a change is made in such a class, there is a chance that client classes that extend the class will break. This is not just a theoretical problem. It is not uncommon to receive subclassing-related bug reports after modifying the internals of a nonfinal concrete class that was not designed and documented for inheritance.

The best solution to this problem is to prohibit subclassing in classes that are not designed and documented to be safely subclassed. There are two ways to prohibit subclassing. The easier of the two is to declare the class final. The alternative is to make all the constructors private or package-private and to add public static factories in place of the constructors. This alternative, which provides the flexibility to use subclasses internally, is discussed in Item 15. Either approach is acceptable.

This advice may be somewhat controversial, as many programmers have grown accustomed to subclassing ordinary concrete classes to add facilities such as instrumentation, notification, and synchronization or to limit functionality. If a class implements some interface that captures its essence, such as `Set`, `List`, or `Map`, then you should feel no compunction about prohibiting subclassing. The wrapper class pattern, described in Item 16, provides a superior alternative to inheritance for augmenting the functionality.

If a concrete class does not implement a standard interface, then you may inconvenience some programmers by prohibiting inheritance. If you feel that you must allow inheritance from such a class, one reasonable approach is to ensure that the class never invokes any of its overridable methods and to document this fact. In other words, eliminate the class’s self-use of overridable methods entirely.
In doing so, you’ll create a class that is reasonably safe to subclass. Overriding a method will never affect the behavior of any other method.

You can eliminate a class’s self-use of overridable methods mechanically, without changing its behavior. Move the body of each overridable method to a private “helper method” and have each overridable method invoke its private helper method. Then replace each self-use of an overridable method with a direct invocation of the overridable method’s private helper method.
Item 18: Prefer interfaces to abstract classes

The Java programming language provides two mechanisms for defining a type that permits multiple implementations: interfaces and abstract classes. The most obvious difference between the two mechanisms is that abstract classes are permitted to contain implementations for some methods while interfaces are not. A more important difference is that to implement the type defined by an abstract class, a class must be a subclass of the abstract class. Any class that defines all of the required methods and obeys the general contract is permitted to implement an interface, regardless of where the class resides in the class hierarchy. Because Java permits only single inheritance, this restriction on abstract classes severely constrains their use as type definitions.

Existing classes can be easily retrofitted to implement a new interface. All you have to do is add the required methods if they don’t yet exist and add an implements clause to the class declaration. For example, many existing classes were retrofitted to implement the Comparable interface when it was introduced into the platform. Existing classes cannot, in general, be retrofitted to extend a new abstract class. If you want to have two classes extend the same abstract class, you have to place the abstract class high up in the type hierarchy where it subclasses an ancestor of both classes. Unfortunately, this causes great collateral damage to the type hierarchy, forcing all descendants of the common ancestor to extend the new abstract class whether or not it is appropriate for them to do so.

Interfaces are ideal for defining mixins. A mixin is a type that a class can implement in addition to its “primary type” to declare that it provides some optional behavior. For example, Comparable is a mixin interface that allows a class to declare that its instances are ordered with respect to other mutually comparable objects. Such an interface is called a mixin because it allows the optional functionality to be “mixed in” to the type’s primary functionality. Abstract classes cannot be used to define mixins for the same reason that they can’t be retrofitted onto existing classes: a class cannot have more than one parent, and there is no reasonable place in the class hierarchy to put a mixin.

Interfaces allow the construction of nonhierarchical type frameworks. Type hierarchies are great for organizing some things, but other things don’t fall neatly into a rigid hierarchy. For example, suppose we have an interface representing a singer and another representing a songwriter:

```java
public interface Singer {
    AudioClip sing(Song s);
}
```
public interface Songwriter {
    Song compose(boolean hit);
}

In real life, some singers are also songwriters. Because we used interfaces rather than abstract classes to define these types, it is perfectly permissible for a single class to implement both Singer and Songwriter. In fact, we can define a third interface that extends both Singer and Songwriter and adds new methods that are appropriate to the combination:

public interface SingerSongwriter extends Singer, Songwriter {
    AudioClip strum();
    void actSensitive();
}

You don’t always need this level of flexibility, but when you do, interfaces are a lifesaver. The alternative is a bloated class hierarchy containing a separate class for every supported combination of attributes. If there are \( n \) attributes in the type system, there are \( 2^n \) possible combinations that you might have to support. This is what’s known as a \textit{combinatorial explosion}. Bloated class hierarchies can lead to bloated classes containing many methods that differ only in the type of their arguments, as there are no types in the class hierarchy to capture common behaviors.

\textbf{Interfaces enable safe, powerful functionality enhancements} via the \textit{-wrapper class} idiom, described in Item 16. If you use abstract classes to define types, you leave the programmer who wants to add functionality with no alternative but to use inheritance. The resulting classes are less powerful and more fragile than wrapper classes.

While interfaces are not permitted to contain method implementations, using interfaces to define types does not prevent you from providing implementation assistance to programmers. \textbf{You can combine the virtues of interfaces and abstract classes by providing an abstract skeletal implementation class to go with each nontrivial interface that you export}. The interface still defines the type, but the skeletal implementation takes all of the work out of implementing it.

By convention, skeletal implementations are called \texttt{AbstractInterface}, where \texttt{Interface} is the name of the interface they implement. For example, the Collections Framework provides a skeletal implementation to go along with each main collection interface: \texttt{AbstractCollection}, \texttt{AbstractSet}, \texttt{AbstractList}, and \texttt{AbstractMap}. Arguably it would have made sense to call them \texttt{SkeletalCollection}, \texttt{SkeletalSet}, \texttt{SkeletalList}, and \texttt{SkeletalMap}, but the \texttt{Abstract} convention is now firmly established.
When properly designed, skeletal implementations make it very easy for programmers to provide their own implementations of your interfaces. For example, here’s a static factory method containing a complete, fully functional `List` implementation:

```java
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(final int[] a) {
    if (a == null)
        throw new NullPointerException();

    return new AbstractList<Integer>() {
        public Integer get(int i) {
            return a[i]; // Autoboxing
        }

        public int size() {
            return a.length;
        }

        public Integer set(int i, Integer val) {
            int oldVal = a[i];
            a[i] = val; // Auto-unboxing
            return oldVal; // Autoboxing
        }
    };
}
```

When you consider all that a `List` implementation does for you, this example is an impressive demonstration of the power of skeletal implementations. Incidentally, the example is an `Adapter` [Gamma95, p. 139] that allows an `int` array to be viewed as a list of `Integer` instances. Because of all the translation back and forth between `int` values and `Integer` instances (boxing and unboxing), the performance is not terribly good. Note that a static factory is provided and that the class is an inaccessible `anonymous class` (Item 22) hidden inside the static factory.

The beauty of skeletal implementations is that they provide the implementation assistance of abstract classes without imposing the severe constraints that abstract classes impose when they serve as type definitions. For most implementors of an interface, extending the skeletal implementation is the obvious choice, but it is strictly optional. If a preexisting class cannot be made to extend the skeletal implementation, the class can always implement the interface manually. Furthermore, the skeletal implementation can still aid the implementor’s task. The class implementing the interface can forward invocations of interface methods to a contained instance of a private inner class that extends the skeletal implementa-
tion. This technique, known as *simulated multiple inheritance*, is closely related to the wrapper class idiom discussed in Item 16. It provides most of the benefits of multiple inheritance, while avoiding the pitfalls.

Writing a skeletal implementation is a relatively simple, if somewhat tedious, process. First you must study the interface and decide which methods are the primitives in terms of which the others can be implemented. These primitives will be the abstract methods in your skeletal implementation. Then you must provide concrete implementations of all the other methods in the interface. For example, here’s a skeletal implementation of the `Map.Entry` interface:

```java
// Skeletal Implementation
public abstract class AbstractMapEntry<K, V> implements Map.Entry<K, V> {
    // Primitive operations
    public abstract K getKey();
    public abstract V getValue();

    // Entries in modifiable maps must override this method
    public V setValue(V value) {
        throw new UnsupportedOperationException();
    }

    // Implements the general contract of Map.Entry.equals
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (! (o instanceof Map.Entry))
            return false;
        Map.Entry<?, ?> arg = (Map.Entry) o;
        return equals(getKey(), arg.getKey()) &&
                equals(getValue(), arg.getValue());
    }

    private static boolean equals(Object o1, Object o2) {
        return o1 == null ? o2 == null : o1.equals(o2);
    }

    // Implements the general contract of Map.Entry.hashCode
    public int hashCode() {
        return hashCode(getKey()) ^ hashCode(getValue());
    }

    private static int hashCode(Object obj) {
        return obj == null ? 0 : obj.hashCode();
    }
}
```
Because skeletal implementations are designed for inheritance, you should follow all of the design and documentation guidelines in Item 17. For brevity’s sake, the documentation comments were omitted from the previous example, but good documentation is absolutely essential for skeletal implementations.

A minor variant on the skeletal implementation is the simple implementation, exemplified by AbstractMap.SimpleEntry. A simple implementation is like a skeletal implementation in that it implements an interface and is designed for extension, but differs in that it isn’t abstract: it is the simplest possible working implementation. You can use it as it stands or subclass it as circumstances warrant.

Using abstract classes to define types that permit multiple implementations has one great advantage over using interfaces: It is far easier to evolve an abstract class than an interface. If, in a subsequent release, you want to add a new method to an abstract class, you can always add a concrete method containing a reasonable default implementation. All existing implementations of the abstract class will then provide the new method. This does not work for interfaces.

It is, generally speaking, impossible to add a method to a public interface without breaking all existing programs that use the interface. Classes that previously implemented the interface will be missing the new method and won’t compile anymore. You could limit the damage somewhat by adding the new method to the skeletal implementation at the same time as you added it to the interface, but this really wouldn’t solve the problem. Any implementation that didn’t inherit from the skeletal implementation would still be broken.

Public interfaces, therefore, must be designed carefully. Once an interface is released and widely implemented, it is almost impossible to change it. You really must get it right the first time. If an interface contains a minor flaw, it will irritate you and its users forever. If an interface is severely deficient, it can doom an API. The best thing to do when releasing a new interface is to have as many programmers as possible implement the interface in as many ways as possible before the interface is frozen. This will allow you to discover flaws while you can still correct them.

To summarize, an interface is generally the best way to define a type that permits multiple implementations. An exception to this rule is the case where ease of evolution is deemed more important than flexibility and power. Under these circumstances, you should use an abstract class to define the type, but only if you understand and can accept the limitations. If you export a nontrivial interface, you should strongly consider providing a skeletal implementation to go with it. Finally, you should design all of your public interfaces with the utmost care and test them thoroughly by writing multiple implementations.
Item 19: Use interfaces only to define types

When a class implements an interface, the interface serves as a type that can be used to refer to instances of the class. That a class implements an interface should therefore say something about what a client can do with instances of the class. It is inappropriate to define an interface for any other purpose.

One kind of interface that fails this test is the so-called constant interface. Such an interface contains no methods; it consists solely of static final fields, each exporting a constant. Classes using these constants implement the interface to avoid the need to qualify constant names with a class name. Here is an example:

```java
// Constant interface pattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER = 6.02214199e23;
    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    // Mass of the electron (kg)
    static final double ELECTRON_MASS = 9.10938188e-31;
}
```

The constant interface pattern is a poor use of interfaces. That a class uses some constants internally is an implementation detail. Implementing a constant interface causes this implementation detail to leak into the class’s exported API. It is of no consequence to the users of a class that the class implements a constant interface. In fact, it may even confuse them. Worse, it represents a commitment: if in a future release the class is modified so that it no longer needs to use the constants, it still must implement the interface to ensure binary compatibility. If a nonfinal class implements a constant interface, all of its subclasses will have their namespaces polluted by the constants in the interface.

There are several constant interfaces in the Java platform libraries, such as java.io.ObjectStreamConstants. These interfaces should be regarded as anomalies and should not be emulated.

If you want to export constants, there are several reasonable choices. If the constants are strongly tied to an existing class or interface, you should add them to the class or interface. For example, all of the numerical wrapper classes in the Java platform libraries, such as Integer and Float, export MIN_VALUE and MAX_VALUE constants. If the constants are best viewed as members of an enumer-
ated type, you should export them with an enum type (Item 22). Otherwise, you should export the constants with a noninstantiable utility class (Item 4). Here is a utility class version of the PhysicalConstants example above:

```java
// Constant utility class
package com.effectivejava.science;

public class PhysicalConstants {
    private PhysicalConstants() {} // Prevents instantiation

    public static final double AVOGADROS_NUMBER   = 6.02214199e23;
    public static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    public static final double ELECTRON_MASS      = 9.10938188e-31;
}
```

Normally a utility class requires clients to qualify constant names with a class name, for example PhysicalConstants.AVOGADROS_NUMBER. If you make heavy use of the constants exported by a utility class, you can avoid the need for qualifying the constants with the class name by making use of the static import facility, introduced in release 1.5:

```java
// Use of static import to avoid qualifying constants
import static com.effectivejava.science.PhysicalConstants.*;

public class Test {
    double atoms(double mols) {
        return AVOGADROS_NUMBER * mols;
    }
    ...
    // Many more uses of PhysicalConstants justify static import
}
```

In summary, interfaces should be used only to define types. They should not be used to export constants.
Item 20: Prefer class hierarchies to tagged classes

Occasionally you may run across a class whose instances come in two or more flavors and contain a tag field indicating the flavor of the instance. For example consider this class, which is capable of representing a circle or rectangle:

```java
// Tagged class - vastly inferior to a class hierarchy!
class Figure {
    enum Shape { RECTANGLE, CIRCLE };

    // Tag field - the shape of this figure
    final Shape shape;

    // These fields are used only if shape is RECTANGLE
    double length;
    double width;

    // This field is used only if shape is CIRCLE
    double radius;

    // Constructor for circle
    Figure(double radius) {
        shape = Shape.CIRCLE;
        this.radius = radius;
    }

    // Constructor for rectangle
    Figure(double length, double width) {
        shape = Shape.RECTANGLE;
        this.length = length;
        this.width = width;
    }

    double area() {
        switch(shape) {
            case RECTANGLE:
                return length * width;
            case CIRCLE:
                return Math.PI * (radius * radius);
            default:
                throw new AssertionError();
        }
    }
}
```
Such tagged classes have numerous shortcomings. They are cluttered with boilerplate including enum declarations, tag fields, and switch statements. Readability is further harmed because multiple implementations are jumbled together in a single class. Memory footprint is increased because instances are burdened with irrelevant fields belonging to other flavors. Fields can’t be made final unless constructors initialize irrelevant fields, resulting in more boilerplate. Constructors must set the tag field and initialize the right data fields with no help from the compiler: if you initialize the wrong fields, the program will fail at run time. You can’t add a flavor to a tagged class unless you can modify its source file. If you do add a flavor, you must remember to add a case to every switch statement, or the class will fail at run time. Finally, the data type of an instance gives no clue as to its flavor. In short, tagged classes are verbose, error-prone, and inefficient.

Luckily, object-oriented languages such as Java offer a far better alternative for defining a single data type capable of representing objects of multiple flavors: subtyping. A tagged class is just a pallid imitation of a class hierarchy.

To transform a tagged class into a class hierarchy, first define an abstract class containing an abstract method for each method in the tagged class whose behavior depends on the tag value. In the Figure class, there is only one such method, which is area. This abstract class is the root of the class hierarchy. If there are any methods whose behavior does not depend on the value of the tag, put them in this class. Similarly, if there are any data fields used by all the flavors, put them in this class. There are no such flavor-independent methods or fields in the Figure class.

Next, define a concrete subclass of the root class for each flavor of the original tagged class. In our example, there are two: circle and rectangle. Include in each subclass the data fields particular to its flavor. In our example, radius is particular to circle, and length and width are particular to rectangle. Also include in each subclass the appropriate implementation of each abstract method in the root class. Here is the class hierarchy corresponding to the original Figure class:

```java
// Class hierarchy replacement for a tagged class
abstract class Figure {
    abstract double area();
}

class Circle extends Figure {
    final double radius;

    Circle(double radius) { this.radius = radius; }

    double area() { return Math.PI * radius * radius; }
}
```
class Rectangle extends Figure {
    final double length;
    final double width;

    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }
    double area() { return length * width; }
}

This class hierarchy corrects every shortcoming of tagged classes noted previously. The code is simple and clear, containing none of the boilerplate found in the original. The implementation of each flavor is allotted its own class, and none of these classes are encumbered by irrelevant data fields. All fields are final. The compiler ensures that each class’s constructor initializes its data fields, and that each class has an implementation for every abstract method declared in the root class. This eliminates the possibility of a runtime failure due to a missing switch case. Multiple programmers can extend the hierarchy independently and interoperably without access to the source for the root class. There is a separate data type associated with each flavor, allowing programmers to indicate the flavor of a variable and to restrict variables and input parameters to a particular flavor.

Another advantage of class hierarchies is that they can be made to reflect natural hierarchical relationships among types, allowing for increased flexibility and better compile-time type checking. Suppose the tagged class in the original example also allowed for squares. The class hierarchy could be made to reflect the fact that a square is a special kind of rectangle (assuming both are immutable):

class Square extends Rectangle {
    Square(double side) {
        super(side, side);
    }
}

Note that the classes in the above hierarchy are accessed by their fields rather than by accessor methods. This was done for brevity and would be unacceptable if the hierarchy were public (Item 20).

In summary, tagged classes are seldom appropriate. If you’re tempted to write a class with an explicit tag field, think about whether the tag could be eliminated and the class replaced by a hierarchy. When you encounter an existing class with a tag field, consider refactoring it into a hierarchy.
Item 21: Use function objects to represent strategies

Some languages support function pointers, delegates, lambda expressions, or similar facilities that allow programs to store and transmit the ability to invoke a particular function. Such facilities are typically used to allow the caller of a function to specialize its behavior by passing in a second function. For example, the `qsort` function in C’s standard library takes a pointer to a comparator function, which `qsort` uses to compare the elements to be sorted. The comparator function takes two parameters, each of which is a pointer to an element. It returns a negative integer if the element pointed to by the first parameter is less than the one pointed to by the second, zero if the two elements are equal, and a positive integer if the element pointed to by the first parameter is greater than the one pointed to by the second. Different sort orders can be obtained by passing in different comparator functions. This is an example of the Strategy pattern [Gamma95, p. 315]; the comparator function represents a strategy for sorting elements.

Function pointers were omitted from Java because object references can be used to provide the same functionality. Invoking a method on an object typically performs some operation on that object. However, it is possible to define an object whose methods perform operations on other objects, passed explicitly to the methods. An instance of a class that exports exactly one such method is effectively a pointer to that method. Such instances are known as function objects. For example, consider the following class:

```java
class StringLengthComparator {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

This class exports a single method that takes two strings and returns a negative integer if the first string is shorter than the second, zero if the two strings are of equal length, and a positive integer if the first string is longer. This method is a comparator that orders strings based on their length instead of the more typical lexicographic ordering. A reference to a `StringLengthComparator` object serves as a “function pointer” to this comparator, allowing it to be invoked on arbitrary pairs of strings. In other words, a `StringLengthComparator` instance is a concrete strategy for string comparison.

As is typical for concrete strategy classes, the `StringLengthComparator` class is stateless: It has no fields, hence all instances of the class are functionally
equivalent. Thus it could just as well be a singleton to save on unnecessary object creation costs (Item 5, Item 2):

class StringLengthComparator {
    private StringLengthComparator() { }

    public static final StringLengthComparator INSTANCE = new StringLengthComparator();

    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}

To pass a StringLengthComparator instance to a method, we need an appropriate type for the parameter. It would do no good to use StringLengthComparator because clients would be unable to pass any other comparison strategy. Instead, we need to define a Comparator interface and modify StringLengthComparator to implement this interface. In other words, we need to define a strategy interface to go with the concrete strategy class. Here it is:

// Strategy interface
public interface Comparator<T> {
    public int compare(T t1, T t2);
}

This definition of the Comparator interface happens to come from the java.util package, but there’s nothing magic about it: you could just as well have written it yourself. The Comparator interface is parameterized (Item XXX) so that it is applicable to comparators for objects other than strings. Its compare method takes two parameters of type T (its formal type parameter) rather than String. The StringLengthComparator class shown above can be made to implement Comparator<String> merely by declaring it to do so:

class StringLengthComparator implements Comparator<String> {
    ...
    // class body is identical to the one shown above
}
Concrete strategy classes are often declared using anonymous classes (Item 22). The following statement sorts an array of strings according to length:

```java
Arrays.sort(stringArray, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

Because the strategy interface serves as a type for all of its concrete strategy instances, a concrete strategy class needn’t be made public to export a concrete strategy. Instead, a “host class” can export a public static field (or static factory method) whose type is the strategy interface, and the concrete strategy class can be a private nested class of the host. In the example that follows, a static member class is used in preference to an anonymous class to allow the concrete strategy class to implement a second interface, `Serializable`:

```java
// Exporting a concrete strategy
class Host {
    ... // Bulk of class omitted

    private static class StrLenCmp implements Comparator<String>, Serializable {
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }

    // Returned comparator is serializable
    public static final Comparator<String> STRING_LENGTH_COMPARATOR = new StrLenCmp();
}
```

The `String` class uses this pattern to export a case-independent string comparator via its `CASE_INSENSITIVE_ORDER` field.

To summarize, a primary use of function pointers is to implement the Strategy pattern. To implement this pattern in Java, declare an interface to represent the strategy and a class that implements this interface for each concrete strategy. When a concrete strategy is used only once, it is typically declared and instantiated as an anonymous class. When a concrete strategy is designed for repeated use, it is generally implemented as a private static member class and exported in a public static final field whose type is the strategy interface.
Item 22: Favor static member classes over nonstatic

A nested class is a class defined within another class. A nested class should exist only to serve its enclosing class. If a nested class would be useful in some other context, then it should be a top-level class. There are four kinds of nested classes: static member classes, nonstatic member classes, anonymous classes, and local classes. All but the first kind are known as inner classes. This item tells you when to use which kind of nested class and why.

A static member class is the simplest kind of nested class. It is best thought of as an ordinary class that happens to be declared inside another class and has access to all of the enclosing class’s members, even those declared private. A static member class is a static member of its enclosing class and obeys the same accessibility rules as other static members. If it is declared private, it is accessible only within the enclosing class, and so forth.

One common use of a static member class is as a public auxiliary class, useful only in conjunction with its outer class. For example, consider an enum describing the operations supported by a calculator (Item 23). The Operation enum should be a public static member class of the Calculator class. Clients of Calculator could then refer to operations using names like Calculator.Operation.PLUS and Calculator.Operation.MINUS.

Syntactically, the only difference between static and nonstatic member classes is that static member classes have the modifier static in their declarations. Despite the syntactic similarity, these two kinds of nested classes are very different. Each instance of a nonstatic member class is implicitly associated with an enclosing instance of its containing class. Within instance methods of a nonstatic member class, you can invoke methods on the enclosing instance, or obtain a reference to the enclosing instance using the qualified this construct [JLS, 15.8.4]. If an instance of a nested class can exist in isolation from an instance of its enclosing class, then the nested class cannot be a nonstatic member class: It is impossible to create an instance of a nonstatic member class without an enclosing instance.

The association between a nonstatic member class instance and its enclosing instance is established when the former is created; it cannot be modified thereafter. Normally, the association is established automatically by invoking a nonstatic member class constructor from within an instance method of the enclosing class. It is possible, although rare, to establish the association manually using the expression enclosingInstance.new MemberClass(args). As you would expect, the association takes up space in the nonstatic member class instance and adds time to its construction.
One common use of a nonstatic member class is to define an Adapter [Gamma95, p. 139] that allows an instance of the outer class to be viewed as an instance of some unrelated class. For example, implementations of the Map interface typically use nonstatic member classes to implement their collection views, which are returned by Map's keySet, entrySet, and values methods. Similarly, implementations of the collection interfaces, such as Set and List, typically use nonstatic member classes to implement their iterators:

```java
// Typical use of a nonstatic member class
public class MySet<E> extends AbstractSet<E> {
    ...
    // Bulk of the class omitted

    public Iterator<E> iterator() {
        return new MyIterator();
    }

    private class MyIterator implements Iterator<E> {
        ...
    }
}
```

If you declare a member class that does not require access to an enclosing instance, remember to put the static modifier in its declaration, making it a static rather than a nonstatic member class. If you omit the static modifier, each instance will have an extraneous reference to the enclosing object. Storing this reference costs time and space, with no corresponding benefits. Should you ever need to allocate an instance without an enclosing instance, you'll be unable to do so, as nonstatic member class instances are required to have an enclosing instance.

A common use of private static member classes is to represent components of the object represented by their enclosing class. For example, consider a Map instance, which associates keys with values. Many Map implementations have an internal Entry object for each key-value pair in the map. While each entry is associated with a map, the methods on an entry (getKey, getValue, and setValue) do not need access to the map. Therefore it would be wasteful to use a nonstatic member class to represent entries: a private static member class is best. If you accidentally omit the static modifier in the entry declaration, the map will still work, but each entry will contain a superfluous reference to the map, which wastes space and time.

It is doubly important to choose correctly between a static and nonstatic member class if the class in question is a public or protected member of an exported class. In this case, the member class is an exported API element and may not be
changed from a nonstatic to a static member class in a subsequent release without violating binary compatibility.

Anonymous classes are unlike anything else in the Java programming language. As you would expect, an anonymous class has no name. It is not a member of its enclosing class. Rather than being declared along with other members, it is simultaneously declared and instantiated at the point of use. Anonymous classes are permitted at any point in the code where an expression is legal. Anonymous classes behave like static or nonstatic member classes depending on where they occur: they have enclosing instances if they occur in a nonstatic context.

There are several limitations on the applicability of anonymous classes. Because they are simultaneously declared and instantiated, an anonymous class may be used only if it is instantiated at a single point in the code. Because anonymous classes have no name, they may be used only if there is no need to refer to them after they are instantiated. Anonymous classes typically implement only methods in their interface or superclass. They do not declare any new methods, as there is no nameable type to access new methods. Because anonymous classes occur in the midst of expressions, they should be very short, perhaps twenty lines or less. Longer anonymous classes would harm the readability of the program.

One common use of an anonymous class is to create a function object (Item 21) on the fly. For example, the sort method invocation on page 105 sorts an array of strings according to their length using an anonymous Comparator instance. Another common use of an anonymous class is to create a process object, such as a Thread, Runnable, or TimerTask instance. A third common use is within a static factory method (see the intArrayAsList method in Item 18).

Local classes are probably the least frequently used of the four kinds of nested classes. A local class may be declared anywhere that a local variable may be declared and obeys the same scoping rules. Local classes have attributes in common with each of the other kinds of nested classes. Like member classes, they have names and may be used repeatedly. Like anonymous classes, they have enclosing instances only if they are defined in a nonstatic context. Like anonymous classes, they should be short so as not to harm readability.

To recap, there are four different kinds of nested classes, and each has its place. If a nested class needs to be visible outside of a single method or is too long to fit comfortably inside a method, use a member class. If each instance of the member class needs a reference to its enclosing instance, make it nonstatic; otherwise make it static. Assuming the class belongs inside a method, if you need to create instances from only one location and there is a preexisting type that characterizes the class, make it an anonymous class; otherwise, make it a local class.