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 15: Minimize the accessibility of classes and members

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

Information hiding is important for many reasons, most of which stem from the fact that it *decouples* the components that comprise a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation. This speeds up system development because components can be developed in parallel. It eases the burden of maintenance because components can be understood more quickly and debugged or replaced with little fear of harming other components. 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 components are causing performance problems (Item 67), those components can be optimized without affecting the correctness of others. Information hiding increases software reuse because components that aren't tightly coupled often prove useful in other contexts besides the ones for which they were

developed. Finally, information hiding decreases the risk in building large systems because individual components may prove successful even if the system does not.

Java has many facilities to aid in information hiding. The *access control* mechanism [JLS, 6.6] specifies the *accessibility* of classes, interfaces, and members. The accessibility of an entity is determined by the location of its declaration and by which, if any, of the access modifiers (`private`, `protected`, and `public`) is present on the declaration. Proper use of these modifiers is essential to information hiding.

The rule of thumb is simple: **make each class or member as inaccessible as possible.** In other words, 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 implementation rather than the 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 by only one class, consider making the top-level class a private static nested class of the sole class that uses it (Item 24). This reduces its accessibility from all the classes in its package to the one class that uses it. But it is far more important to reduce the accessibility of a gratuitously public class than of a package-private top-level class: the public class is part of the package's API, while the package-private top-level class is already part of its implementation.

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 from 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 (except for interface members, which are public by default).
- **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.

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` (Items 86 and 87).

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. Also, a protected member of an exported class represents a public commitment to an implementation detail (Item 19). The need for protected members should be relatively rare.

There is a key rule that restricts your ability to reduce the accessibility of methods. If a method overrides a superclass method, it cannot have a more restrictive access level in the subclass than 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 (the *Liskov substitution principle*, see Item 15). 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 in the interface must be declared public in the class.

To facilitate testing your code, you may be tempted to make a class, interface, or member more accessible than otherwise necessary. This is fine up to a point. It is acceptable to make a private member of a public class package-private in order to test it, but it is not acceptable to raise the accessibility any higher. In other words, it is not acceptable to make a class, interface, or member a part of a package's exported API to facilitate testing. Luckily, it isn't necessary either because tests can be made to run as part of the package being tested, thus gaining access to its package-private elements.

Instance fields of public classes should rarely be public (Item 16). If an instance field is nonfinal or is a reference to a mutable object, then by making it public, you give up the ability to limit the values that can be stored in the field. This means you give up the ability to enforce invariants involving the field. Also, you give up the ability to take any action when the field is modified, so **classes with public mutable fields are not generally thread-safe.** Even if a field is final and refers to an immutable object, by making it public you give up the flexibility to switch to a new internal data representation in which the field does not exist.

The same advice applies to static fields, with one exception. You can expose constants via public static final fields, assuming the constants form an integral part of the abstraction provided by the class. By convention, such fields have names consisting of capital letters, with words separated by underscores (Item 68). It is critical that these fields contain either primitive values or references to immutable objects (Item 17). a 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 wrong for a class to have a public static final array field, or an accessor that returns such a field.** If a class has such a field or accessor, clients will be able to modify the contents of the array. This is a frequent source of security holes:

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

Beware of the fact that some IDEs generate accessors that return references to private array fields, resulting in exactly this problem. There are two ways to fix the problem. You can make the public array private and add a public immutable list:

```
private static final Thing[] PRIVATE_VALUES = { ... };
public static final List<Thing> VALUES =
   Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

Alternatively, you can make the array private and add a public method that returns a copy of a private array:

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

To choose between these alternatives, think about what the client is likely to do with the result. Which return type will be more convenient? Which will give better performance?

As of Java 9, there are two additional, implicit access levels introduced as part of the *module system*. A module is a grouping of packages, like a package is a grouping of classes. A module may explicitly export some of its packages via *export declarations* in its *module declaration* (which is by convention contained in a source file named `module-info.java`). Public and protected members of unexported packages in a module are inaccessible outside the module; within the

module, accessibility is unaffected by export declarations. Using the module system allows you to share classes among packages within a module without making them visible to the entire world. Public and protected members of public classes in unexported packages give rise to the two implicit access levels, which are intramodular analogues of the normal public and protected levels. The need for this kind of sharing is relatively rare and can often be eliminated by rearranging the classes within your packages.

Unlike the four main access levels, the two module-based levels are largely advisory. If you place a module's JAR file on your application's class path instead of its module path, the packages in the module revert to their non-modular behavior: all of the public and protected members of the packages' public classes have their normal accessibility, regardless of whether the packages are exported by the module [Reinhold, 1.2]. The one place where the newly introduced access levels are strictly enforced is the JDK itself: the unexported packages in the Java libraries are truly inaccessible outside of their modules.

Not only is the access protection afforded by modules of limited utility to the typical Java programmer, and largely advisory in nature; in order to take advantage of it, you must group your packages into modules, make all of their dependencies explicit in module declarations, rearrange your source tree, and take special actions to accommodate any access to non-modularized packages from within your modules [Reinhold, 3]. It is too early to say whether modules will achieve widespread use outside of the JDK itself. In the meantime, it seems best to avoid them unless you have a compelling need.

To summarize, you should reduce accessibility of program elements as much as possible (within reason). After carefully designing a minimal public API, you should prevent any stray classes, interfaces, or members from becoming part of the API. With the exception of public static final fields, which serve as constants, public classes should have no public fields. Ensure that objects referenced by public static final fields are immutable.

Item 16: In public classes, use accessor methods, not public fields

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

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

Because the data fields of such classes are accessed directly, these classes do not offer the benefits of *encapsulation* (Item 15). You can't change the representation without changing the API, you can't enforce invariants, and you can't take auxiliary action when a field is accessed. Hard-line object-oriented programmers feel that such classes are anathema and should always be replaced by classes with private fields and public *accessor methods* (getters) and, for mutable classes, *mutators* (setters):

```
// Encapsulation of data by accessor methods and mutators
class Point {
    private double x;
    private double y;

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

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

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

Certainly, the hard-liners are correct when it comes to public classes: **if a class is accessible outside its package, provide accessor methods** to preserve the flexibility to change the class's internal representation. If a public class exposes its data fields, all hope of changing its representation is lost because client code can be distributed far and wide.

However, **if a class is package-private or is a private nested class, there is nothing inherently wrong with 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 67, 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:

```
// 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, however, sometimes desirable for package-private or private nested classes to expose fields, whether mutable or immutable.

Item 17: Minimize mutability

An immutable class is simply a class whose instances cannot be modified. All of the information contained in each instance is fixed for the lifetime of the object, so no changes can ever be observed. The Java platform libraries contain many immutable classes, including `String`, the boxed primitive classes, and `BigInteger` and `BigDecimal`. There are many good reasons for this: Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.

To make a class immutable, follow these five rules:

1. **Don't provide methods that modify the object's state** (known as *mutators*).
2. **Ensure that the class can't be extended.** This prevents careless or malicious subclasses from compromising the immutable behavior of the class by behaving as if the object's state has changed. Preventing subclassing is generally accomplished by making the class final, but there is an alternative that we'll discuss later.
3. **Make all fields final.** This clearly expresses your intent in a manner that is enforced by the system. Also, it is necessary to ensure correct behavior if a reference to a newly created instance is passed from one thread to another without synchronization, as spelled out in the *memory model* [JLS, 17.5; Goetz06, 16].
4. **Make all fields private.** This prevents clients from obtaining access to mutable objects referred to by fields and modifying these objects directly. While it is technically permissible for immutable classes to have public final fields containing primitive values or references to immutable objects, it is not recommended because it precludes changing the internal representation in a later release (Items 15 and 16).
5. **Ensure exclusive access to any mutable components.** If your class has any fields that refer to mutable objects, ensure that clients of the class cannot obtain references to these objects. Never initialize such a field to a client-provided object reference or return the field from an accessor. Make *defensive copies* (Item 50) in constructors, accessors, and `readObject` methods (Item 88).

Many of the example classes in previous items are immutable. One such class is `PhoneNumber` in Item 11, which has accessors for each attribute but no corresponding mutators. Here is a slightly more complex example:

```
// Immutable complex number class
public final class Complex {
    private final double re;
    private final double im;

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

    public double realPart()      { return re; }
    public double imaginaryPart() { return im; }

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

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

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

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

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

       // See page 47 to find out why we use compare instead of ==
       return Double.compare(c.re, re) == 0
           && Double.compare(c.im, im) == 0;
    }
    @Override public int hashCode() {
        return 31 * Double.hashCode(re) + Double.hashCode(im);
    }

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

This class represents a *complex number* (a number with both real and imaginary parts). In addition to the standard `Object` methods, it provides accessors for the real and imaginary parts and provides the four basic arithmetic operations: addition, subtraction, multiplication, and division. Notice how the arithmetic operations create and return a new `Complex` instance rather than modifying this instance. This pattern is known as the *functional* approach because methods return the result of applying a function to their operand, without modifying it. Contrast it to the *procedural* or *imperative* approach in which methods apply a procedure to their operand, causing its state to change. Note that the method names are prepositions (such as `plus`) rather than verbs (such as `add`). This emphasizes the fact that methods don't change the values of the objects. The `BigInteger` and `BigDecimal` classes did *not* obey this naming convention, and it led to many usage errors.

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 achieve thread safety. Since no thread can ever observe any effect of another thread on an immutable object, **immutable objects can be shared freely.** Immutable classes should therefore encourage clients to reuse existing instances wherever possible. One easy way to do this is to provide public static final constants for commonly used values. For example, the `Complex` class might provide these constants:

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

This approach can be taken one step further. An immutable class can provide static factories (Item 1) that cache frequently requested instances to avoid creating new instances when existing ones would do. All the boxed primitive classes and `BigInteger` do this. Using such static factories causes clients to share instances instead of creating new ones, reducing memory footprint and garbage collection

costs. Opting for static factories in place of public constructors when designing a new class gives you the flexibility to add caching later, without modifying clients.

A consequence of the fact that immutable objects can be shared freely is that you never have to make *defensive copies* of them (Item 50). 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 13) 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 6).

Not only can you share immutable objects, but they 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 even though it is mutable; 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.

Immutable objects provide failure atomicity for free (Item 76). Their state never changes, so there is no possibility of a temporary inconsistency.

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

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

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

```
BitSet moby = ...;
moby.flip(0);
```

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 to provide them as primitives. If a multistep operation is provided as a primitive, the immutable class does not have to create a separate object at each step. Internally, the immutable class can be arbitrarily clever. For example, `BigInteger` has a package-private mutable "companion class" that it uses to speed up multistep operations such as modular exponentiation. It is much harder to use the mutable companion class than to use `BigInteger`, for all of the reasons outlined earlier. Luckily, you don't have to use it: 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 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 its obsolete predecessor, `StringBuffer`).

Now that you know how to make an immutable class and you understand the pros and cons of immutability, let's discuss a few design alternatives. Recall that to guarantee immutability, a class must not permit itself to be subclassed. This can be done by making the class final, but there is another, more flexible alternative. Instead of making an immutable class final, you can make all of its constructors private or package-private and add public static factories in place of the public constructors (Item 1). To make this concrete, here's how `Complex` would look if you took this approach:

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

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

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

    ... // Remainder unchanged
}
```

This approach 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.

It was not widely understood that immutable classes had to be effectively final when `BigInteger` and `BigDecimal` were written, so all of their methods may be overridden. Unfortunately, this could not be corrected after the fact while preserving backward compatibility. If you write a class whose security depends on the immutability of a `BigInteger` or `BigDecimal` argument from an untrusted client, you must check to see that the argument is a "real" `BigInteger` or `BigDecimal`, rather than an instance of an untrusted subclass. If it is the latter, you must defensively copy it under the assumption that it might be mutable (Item 50):

```
public static BigInteger safeInstance(BigInteger val) {
    return val.getClass() == BigInteger.class ?
            val : new BigInteger(val.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 11, page 53) 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 83), is also used by `String`.

One caveat should be added concerning serializability. If you choose to have your immutable class implement `Serializable` and it contains one or more fields that refer to mutable objects, you must provide an explicit `readObject` or `readResolve` method, or use the `ObjectOutputStream.writeUnshared` and

`ObjectInputStream.readUnshared` methods, even if the default serialized form is acceptable. Otherwise an attacker could create a mutable instance of your class. This topic is covered in detail in Item 88.

To summarize, resist the urge to write a setter for every getter. **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 67).

There are some classes for which immutability is impractical. **If a class cannot be made immutable, limit its mutability as much as possible.** Reducing the number of states in which an object can exist makes it easier to reason about the object and reduces the likelihood of errors. Therefore, make every field final unless there is a compelling reason to make it nonfinal. Combining the advice of this item with that of Item 15, your natural inclination should be to **declare every field `private final` unless there's a good reason to do otherwise.**

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

The `CountDownLatch` class exemplifies these principles. It is mutable, but its state space is kept intentionally small. You create an instance, use it once, and it's done: once the countdown latch's count has reached zero, you may not reuse 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 18: 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 implementations 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 19). 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 when 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:

```
// 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(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. Incidentally, note that we create a list using the static factory method `List.of`, which was added in Java 9; if you're using an earlier release, use `Arrays.asList` instead:

```
InstrumentedHashSet<String> s = new InstrumentedHashSet<>();
s.addAll(List.of("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, error-prone, and may reduce performance. Additionally, it isn't always possible because 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 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 these 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 problems described earlier. Furthermore, it is doubtful that your method will fulfill the contract of the new superclass method, because 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 above. 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. Note that the implementation is broken into two pieces, the class itself and a reusable *forwarding class,* which contains all of the forwarding methods and nothing else:

```
// Wrapper class - uses composition in place of inheritance
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @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;
    }
}

// Reusable forwarding class
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;
    public ForwardingSet(Set<E> s) { this.s = s; }

    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 add(E e)           { return s.add(e);      }
    public boolean remove(Object o)   { return s.remove(o);   }
    public boolean containsAll(Collection<?> c)
                                      { return s.containsAll(c); }
    public boolean addAll(Collection<? extends E> c)
                                      { return s.addAll(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); }
    @Override public boolean equals(Object o)
                                       { return s.equals(o);  }
    @Override public int hashCode()    { return s.hashCode(); }
    @Override 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:

```
Set<Instant> times = new InstrumentedSet<>(new TreeSet<>(cmp));
Set<E> s = new InstrumentedSet<>(new HashSet<>(INIT_CAPACITY));
```

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

```
static void walk(Set<Dog> dogs) {
    InstrumentedSet<Dog> iDogs = new InstrumentedSet<>(dogs);
    ... // Within this method use iDogs instead of dogs
}
```

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] because the `InstrumentedSet` class "decorates" a set by adding instrumentation. Sometimes the combination of composition and forwarding is loosely referred to as *delegation.* Technically it's not delegation unless the wrapper object passes itself to the wrapped object [Lieberman86; Gamma95].

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's tedious to write forwarding methods, but you have to write the reusable forwarding class for each interface only once, and forwarding classes may be provided for you. For example, Guava provides forwarding classes for all of the collection interfaces [Guava].

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 different 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 `Stack` should not extend `Vector`. Similarly, a property list is not a hash table, so `Properties` should not extend `Hashtable`. In both cases, composition would have been preferable.

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 clients access them directly. At the very least, it can lead to confusing semantics. For example, if `p` refers to a `Properties` instance, then `p.getProperty(key)` may yield different results from `p.get(key)`: the former method takes defaults into account, while the latter method, which is inherited from `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 `Properties`, the designers intended that only strings be allowed as keys and values, but direct access to the underlying `Hashtable` allows this invariant to be violated. Once violated, it is no longer possible to use other parts of the `Properties` API (`load` and `store`). By the time this problem was discovered, it was too late to correct it because clients depended on the use of non-string 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 inheritance. 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 19: Design and document for inheritance or else prohibit it

Item 18 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, the documentation must indicate which overridable methods the method 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.

A method that invokes overridable methods contains a description of these invocations at the end of its documentation comment. The description is in a special section of the specification, labeled "Implementation Requirements," which is generated by the Javadoc tag `@implSpec`. This section describes the inner workings of the method. Here's an example, copied from the specification for `java.util.AbstractCollection`:

```
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 `Objects.equals(o, e)`, if this collection contains one or more such elements. Returns `true` if this collection contained the specified element (or equivalently, if this collection changed as a result of the call).
>
> **Implementation Requirements:** 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 and this collection contains the specified object.

This documentation leaves no doubt that overriding the `iterator` method will affect the behavior of the `remove` method. It also 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 18, 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.

The `@implSpec` tag was added in Java 8 and used heavily in Java 9. This tag should be enabled by default, but as of Java 9, the Javadoc utility still ignores it unless you pass the command line switch `-tag "apiNote:a:API Note:"`.

Designing 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`:

> `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 list 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.
>
> **Implementation Requirements:** 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 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 because a missing protected member can render a class practically unusable for inheritance.

The *only* way to test a class designed for inheritance is to write subclasses. If you omit a crucial protected member, trying to write a subclass will make the omission painfully obvious. Conversely, if several subclasses are written and none uses a protected member, you should probably make it private. Experience shows that three subclasses are usually sufficient to test an extendable class. One or more of these subclasses should be written by someone other than the superclass author.

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 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:

```
public class Super {
    // Broken - constructor invokes an overridable method
    public Super() {
        overrideMe();
    }
    public void overrideMe() {
    }
}
```

Here's a subclass that overrides the `overrideMe` method, which is erroneously invoked by `Super`'s sole constructor:

```
public final class Sub extends Super {
    // Blank final, set by constructor
    private final Instant instant;

    Sub() {
        instant = Instant.now();
    }

    // Overriding method invoked by superclass constructor
    @Override public void overrideMe() {
        System.out.println(instant);
    }

    public static void main(String[] args) {
        Sub sub = new Sub();
        sub.overrideMe();
    }
}
```

You might expect this program to print out the instant twice, but it prints out `null` the first time because `overrideMe` is invoked by the `Super` constructor before the `Sub` constructor has a chance to initialize the `instant` field. Note that this program observes a final field in two different states! Note also that if `overrideMe` had invoked any method on `instant`, it would have thrown a `NullPointerException` when the `Super` constructor invoked `overrideMe`. The only reason this program doesn't throw a `NullPointerException` as it stands is that the `println` method tolerates null parameters.

Note that it *is* safe to invoke private methods, final methods, and static methods, none of which are overridable, from a constructor.

The `Cloneable` and `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 because 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 13 and Item 86.

If you do decide to implement either `Cloneable` or `Serializable` in a class that is designed for inheritance, you should be aware that because the `clone` and `readObject` methods behave a lot like constructors, a similar restriction applies: **neither `clone` nor `readObject` may invoke an overridable method, directly or indirectly.** In the case of `readObject`, the overriding method will run before the

subclass's state has been deserialized. In the case of `clone`, the overriding method will run before the subclass's `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 `clone`, the failure can damage the original object as well as the clone. This can happen, for example, if the overriding method assumes it is modifying the clone's copy of the object's deep structure, but the copy hasn't been made yet.

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 requires great effort and 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 20). There are other situations where it is clearly the wrong thing to do, such as immutable classes (Item 17).

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 subclasses extending 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 17. Either approach is acceptable.

This advice may be somewhat controversial because 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 18, 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.

In summary, designing a class for inheritance is hard work. You must document all of its self-use patterns, and once you've documented them, you must commit to them for the life of the class. If you fail to do this, subclasses may become dependent on implementation details of the superclass and may break if the implementation of the superclass changes. To allow others to write *efficient* subclasses, you may also have to export one or more protected methods. Unless you know there is a real need for subclasses, you are probably better off prohibiting inheritance by declaring your class final or ensuring that there are no accessible constructors.

Item 20: Prefer interfaces to abstract classes

Java has two mechanisms to define a type that permits multiple implementations: interfaces and abstract classes. Since the introduction of *default methods* for interfaces in Java 8 [JLS 9.4.3], both mechanisms allow you to provide implementations for some instance methods. A major difference is that to implement the type defined by an abstract class, a class must be a subclass of the abstract class. Because Java permits only single inheritance, this restriction on abstract classes severely constrains their use as type definitions. Any class that defines all the required methods and obeys the general contract is permitted to implement an interface, regardless of where the class resides in the class hierarchy.

Existing classes can easily be retrofitted to implement a new interface. All you have to do is to add the required methods, if they don't yet exist, and to add an `implements` clause to the class declaration. For example, many existing classes were retrofitted to implement the `Comparable`, `Iterable`, and `Autocloseable` interfaces when they were added to 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 it high up in the type hierarchy where it is an ancestor of both classes. Unfortunately, this can cause great collateral damage to the type hierarchy, forcing all descendants of the new abstract class to subclass it, whether or not it is appropriate.

Interfaces are ideal for defining mixins. Loosely speaking, 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 can't 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 insert a mixin.

Interfaces allow for 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:

```
public interface Singer {
    AudioClip sing(Song s);
}
```

```
public interface Songwriter {
    Song compose(int chartPosition);
}
```

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 *combinatorial explosion*. Bloated class hierarchies can lead to bloated classes with many methods that differ only in the type of their arguments because there are no types in the class hierarchy to capture common behaviors.

Interfaces enable safe, powerful functionality enhancements via the *wrapper class* idiom (Item 18). If you use abstract classes to define types, you leave the programmer who wants to add functionality with no alternative but inheritance. The resulting classes are less powerful and more fragile than wrapper classes.

When there is an obvious implementation of an interface method in terms of other interface methods, consider providing implementation assistance to programmers in the form of a default method. For an example of this technique, see the `removeIf` method on page 104. If you provide default methods, be sure to document them for inheritance using the `@implSpec` Javadoc tag (Item 19).

There are limits on how much implementation assistance you can provide with default methods. Although many interfaces specify the behavior of `Object` methods such as `equals` and `hashCode`, you are not permitted to provide default methods for them. Also, interfaces are not permitted to contain instance fields or nonpublic static members (with the exception of private static methods). Finally, you can't add default methods to an interface that you don't control.

You can, however, combine the advantages of interfaces and abstract classes by providing an abstract *skeletal implementation class* to go with an interface. The interface defines the type, perhaps providing some default methods, while the skeletal implementation class implements the remaining non-primitive interface

methods atop the primitive interface methods. Extending a skeletal implementation takes most of the work out of implementing an interface. This is the *Template Method* pattern [Gamma95].

By convention, skeletal implementation classes are called `Abstract`*Interface*, where *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: `AbstractCollection`, `AbstractSet`, `AbstractList`, and `AbstractMap`. Arguably it would have made sense to call them `SkeletalCollection`, `SkeletalSet`, `SkeletalList`, and `SkeletalMap`, but the `Abstract` convention is now firmly established. When properly designed, skeletal implementations (whether a separate abstract class, or consisting solely of default methods on an interface) can make it *very* easy for programmers to provide their own implementations of an interface. For example, here's a static factory method containing a complete, fully functional `List` implementation atop `AbstractList`:

```
// Concrete implementation built atop skeletal implementation
static List<Integer> intArrayAsList(int[] a) {
    Objects.requireNonNull(a);

    // The diamond operator is only legal here in Java 9 and later
    // If you're using an earlier release, specify <Integer>
    return new AbstractList<>() {
        @Override public Integer get(int i) {
            return a[i];  // Autoboxing (Item 6)
        }

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

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

When you consider all that a `List` implementation does for you, this example is an impressive demonstration of the power of skeletal implementations. Incidentally, this example is an *Adapter* [Gamma95] 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), its performance is not terribly good. Note that the implementation takes the form of an *anonymous class* (Item 24).

The beauty of skeletal implementation classes is that they provide all of 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 with a skeletal implementation class, extending this class is the obvious choice, but it is strictly optional. If a class cannot be made to extend the skeletal implementation, the class can always implement the interface directly. The class still benefits from any default methods present on the interface itself. 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 implementation. This technique, known as *simulated multiple inheritance*, is closely related to the wrapper class idiom discussed in Item 18. It provides many of the benefits of multiple inheritance, while avoiding the pitfalls.

Writing a skeletal implementation is a relatively simple, if somewhat tedious, process. First, 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. Next, provide default methods in the interface for all of the methods that can be implemented directly atop the primitives, but recall that you may not provide default methods for `Object` methods such as `equals` and `hashCode`. If the primitives and default methods cover the interface, you're done, and have no need for a skeletal implementation class. Otherwise, write a class declared to implement the interface, with implementations of all of the remaining interface methods. The class may contain any nonpublic fields ands methods appropriate to the task.

As a simple example, consider the `Map.Entry` interface. The obvious primitives are `getKey`, `getValue`, and (optionally) `setValue`. The interface specifies the behavior of `equals` and `hashCode`, and there is an obvious implementation of `toString` in terms of the primitives. Since you are not allowed to provide default implementations for the `Object` methods, all implementations are placed in the skeletal implementation class:

```
// Skeletal implementation class
public abstract class AbstractMapEntry<K,V>
        implements Map.Entry<K,V> {
    // Entries in a modifiable map must override this method
    @Override public V setValue(V value) {
        throw new UnsupportedOperationException();
    }
```

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

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

        @Override public String toString() {
            return getKey() + "=" + getValue();
        }
    }
```

Note that this skeletal implementation could not be implemented in the `Map.Entry` interface or as a subinterface because default methods are not permitted to override `Object` methods such as `equals`, `hashCode`, and `toString`.

Because skeletal implementations are designed for inheritance, you should follow all of the design and documentation guidelines in Item 19. For brevity's sake, the documentation comments were omitted from the previous example, but **good documentation is absolutely essential in a skeletal implementation,** whether it consists of default methods on an interface or a separate abstract class.

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 inheritance, but it 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.

To summarize, an interface is generally the best way to define a type that permits multiple implementations. If you export a nontrivial interface, you should strongly consider providing a skeletal implementation to go with it. To the extent possible, you should provide the skeletal implementation via default methods on the interface so that all implementors of the interface can make use of it. That said, restrictions on interfaces typically mandate that a skeletal implementation take the form of an abstract class.

Item 21: Design interfaces for posterity

Prior to Java 8, it was impossible to add methods to interfaces without breaking existing implementations. If you added a new method to an interface, existing implementations would, in general, lack the method, resulting in a compile-time error. In Java 8, the *default method* construct was added [JLS 9.4], with the intent of allowing the addition of methods to existing interfaces. But adding new methods to existing interfaces is fraught with risk.

The declaration for a default method includes a *default implementation* that is used by all classes that implement the interface but do not implement the default method. While the addition of default methods to Java makes it possible to add methods to an existing interface, there is no guarantee that these methods will work in all preexisting implementations. Default methods are "injected" into existing implementations without the knowledge or consent of their implementors. Before Java 8, these implementations were written with the tacit understanding that their interfaces would *never* acquire any new methods.

Many new default methods were added to the core collection interfaces in Java 8, primarily to facilitate the use of lambdas (Chapter 6). The Java libraries' default methods are high-quality general-purpose implementations, and in most cases, they work fine. But **it is not always possible to write a default method that maintains all invariants of every conceivable implementation.**

For example, consider the `removeIf` method, which was added to the `Collection` interface in Java 8. This method removes all elements for which a given `boolean` function (or *predicate*) returns `true`. The default implementation is specified to traverse the collection using its iterator, invoking the predicate on each element, and using the iterator's `remove` method to remove the elements for which the predicate returns `true`. Presumably the declaration looks something like this:

```
// Default method added to the Collection interface in Java 8
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean result = false;
    for (Iterator<E> it = iterator(); it.hasNext(); ) {
        if (filter.test(it.next())) {
            it.remove();
            result = true;
        }
    }
    return result;
}
```

This is the best general-purpose implementation one could possibly write for the `removeIf` method, but sadly, it fails on some real-world `Collection` implementations. For example, consider `org.apache.commons.collections4.collection.SynchronizedCollection`. This class, from the Apache Commons library, is similar to the one returned by the static factory `Collections.synchronizedCollection` in `java.util`. The Apache version additionally provides the ability to use a client-supplied object for locking, in place of the collection. In other words, it is a wrapper class (Item 18), all of whose methods synchronize on a locking object before delegating to the wrapped collection.

The Apache `SynchronizedCollection` class is still being actively maintained, but as of this writing, it does not override the `removeIf` method. If this class is used in conjunction with Java 8, it will therefore inherit the default implementation of `removeIf`, which does not, indeed *cannot*, maintain the class's fundamental promise: to automatically synchronize around each method invocation. The default implementation knows nothing about synchronization and has no access to the field that contains the locking object. If a client calls the `removeIf` method on a `SynchronizedCollection` instance in the presence of concurrent modification of the collection by another thread, a `ConcurrentModificationException` or other unspecified behavior may result.

In order to prevent this from happening in similar Java platform libraries implementations, such as the package-private class returned by `Collections.synchronizedCollection`, the JDK maintainers had to override the default `removeIf` implementation and other methods like it to perform the necessary synchronization before invoking the default implementation. Preexisting collection implementations that were not part of the Java platform did not have the opportunity to make analogous changes in lockstep with the interface change, and some have yet to do so.

In the presence of default methods, existing implementations of an interface may compile without error or warning but fail at runtime. While not terribly common, this problem is not an isolated incident either. A handful of the methods added to the collections interfaces in Java 8 are known to be susceptible, and a handful of existing implementations are known to be affected.

Using default methods to add new methods to existing interfaces should be avoided unless the need is critical, in which case you should think long and hard about whether an existing interface implementation might be broken by your default method implementation. Default methods are, however, extremely useful for providing standard method implementations when an interface is created, to ease the task of implementing the interface (Item 20).

It is also worth noting that default methods were not designed to support removing methods from interfaces or changing the signatures of existing methods. Neither of these interface changes is possible without breaking existing clients.

The moral is clear. Even though default methods are now a part of the Java platform, **it is still of the utmost importance to design interfaces with great care.** While default methods make it *possible* to add methods to existing interfaces, there is great risk in doing so. If an interface contains a minor flaw, it may irritate its users forever; if an interface is severely deficient, it may doom the API that contains it.

Therefore, it is critically important to test each new interface before you release it. Multiple programmers should implement each interface in different ways. At a minimum, you should aim for three diverse implementations. Equally important is to write multiple client programs that use instances of each new interface to perform various tasks. This will go a long way toward ensuring that each interface satisfies all of its intended uses. These steps will allow you to discover flaws in interfaces before they are released, when you can still correct them easily. **While it may be possible to correct some interface flaws after an interface is released, you cannot count on it**.

Item 22: 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:

```
// Constant interface antipattern - do not use!
public interface PhysicalConstants {
    // Avogadro's number (1/mol)
    static final double AVOGADROS_NUMBER   = 6.022_140_857e23;

    // Boltzmann constant (J/K)
    static final double BOLTZMANN_CONSTANT = 1.380_648_52e-23;

    // Mass of the electron (kg)
    static final double ELECTRON_MASS      = 9.109_383_56e-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 boxed numerical primitive classes, such as `Integer` and `Double`, export `MIN_VALUE` and `MAX_VALUE` constants. If the constants are best viewed as members of an enumerated type, you should export

them with an *enum type* (Item 34). Otherwise, you should export the constants with a noninstantiable *utility class* (Item 4). Here is a utility class version of the `PhysicalConstants` example shown earlier:

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

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

  public static final double AVOGADROS_NUMBER = 6.022_140_857e23;
  public static final double BOLTZMANN_CONST  = 1.380_648_52e-23;
  public static final double ELECTRON_MASS    = 9.109_383_56e-31;
}
```

Incidentally, note the use of the underscore character (_) in the numeric literals. Underscores, which have been legal since Java 7, have no effect on the values of numeric literals, but can make them much easier to read if used with discretion. Consider adding underscores to numeric literals, whether fixed of floating point, if they contain five or more consecutive digits. For base ten literals, whether integral or floating point, you should use underscores to separate literals into groups of three digits indicating positive and negative powers of one thousand.

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:

```
// 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 merely to export constants.

Item 23: 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 a rectangle:

```
// 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(shape);
        }
    }
}
```

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 runtime. 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 runtime. 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:

```
// 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; }

    @Override 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;
    }
    @Override 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 is 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 fields in the above hierarchy are accessed directly rather than by accessor methods. This was done for brevity and would be a poor design if the hierarchy were public (Item 16).

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 24: 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 helper class, useful only in conjunction with its outer class. For example, consider an enum describing the operations supported by a calculator (Item 34). 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 *must* be a static 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 member class instance is created and 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, though 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] 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:

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

    @Override 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, *always* put the `static` modifier in its declaration, making it a static rather than a nonstatic member class. If you omit this modifier, each instance will have a hidden extraneous reference to its enclosing instance. As previously mentioned, storing this reference takes time and space. More seriously, it can result in the enclosing instance being retained when it would otherwise be eligible for garbage collection (Item 7). The resulting memory leak can be catastrophic. It is often difficult to detect because the reference is invisible.

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 a 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 cannot be changed from a nonstatic to a static member class in a subsequent release without violating backward compatibility.

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 have enclosing instances if and only if they occur in a nonstatic context. But even if they occur in a static context, they cannot have any static members other than *constant variables*, which are final primitive or string fields initialized to constant expressions [JLS, 4.12.4].

There are many limitations on the applicability of anonymous classes. You can't instantiate them except at the point they're declared. You can't perform `instanceof` tests or do anything else that requires you to name the class. You can't declare an anonymous class to implement multiple interfaces or to extend a class and implement an interface at the same time. Clients of an anonymous class can't invoke any members except those it inherits from its supertype. Because anonymous classes occur in the midst of expressions, they must be kept short—about ten lines or fewer—or readability will suffer.

Before lambdas were added to Java (Chapter 6), anonymous classes were the preferred means of creating small *function objects* and *process objects* on the fly, but lambdas are now preferred (Item 42). Another common use of anonymous classes is in the implementation of static factory methods (see `intArrayAsList` in Item 20).

Local classes are the least frequently used of the four kinds of nested classes. A local class can be declared practically anywhere a local variable can 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 can be used repeatedly. Like anonymous classes, they have enclosing instances only if they are defined in a nonstatic context, and they cannot contain static members. And like anonymous classes, they should be kept 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 a 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.

Item 25: Limit source files to a single top-level class

While the Java compiler lets you define multiple top-level classes in a single source file, there are no benefits associated with doing so, and there are significant risks. The risks stem from the fact that defining multiple top-level classes in a source file makes it possible to provide multiple definitions for a class. Which definition gets used is affected by the order in which the source files are passed to the compiler.

To make this concrete, consider this source file, which contains only a `Main` class that refers to members of two other top-level classes (`Utensil` and `Dessert`):

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
}
```

Now suppose you define both `Utensil` and `Dessert` in a single source file named `Utensil.java`:

```
// Two classes defined in one file. Don't ever do this!
class Utensil {
    static final String NAME = "pan";
}

class Dessert {
    static final String NAME = "cake";
}
```

Of course the main program prints `pancake`.

Now suppose you accidentally make *another* source file named `Dessert.java` that defines the same two classes:

```
// Two classes defined in one file. Don't ever do this!
class Utensil {
    static final String NAME = "pot";
}

class Dessert {
    static final String NAME = "pie";
}
```

If you're lucky enough to compile the program with the command `javac Main.java Dessert.java`, the compilation will fail, and the compiler will

tell you that you've multiply defined the classes `Utensil` and `Dessert`. This is so because the compiler will first compile `Main.java`, and when it sees the reference to `Utensil` (which precedes the reference to `Dessert`), it will look in `Utensil.java` for this class and find both `Utensil` and `Dessert`. When the compiler encounters `Dessert.java` on the command line, it will pull in that file too, causing it to encounter both definitions of `Utensil` and `Dessert`.

If you compile the program with the command `javac Main.java` or `javac Main.java Utensil.java`, it will behave as it did before you wrote the `Dessert.java` file, printing `pancake`. But if you compile the program with the command `javac Dessert.java Main.java`, it will print `potpie`. The behavior of the program is thus affected by the order in which the source files are passed to the compiler, which is clearly unacceptable.

Fixing the problem is as simple as splitting the top-level classes (`Utensil` and `Dessert`, in the case of our example) into separate source files. If you are tempted to put multiple top-level classes into a single source file, consider using static member classes (Item 24) as an alternative to splitting the classes into separate source files. If the classes are subservient to another class, making them into static member classes is generally the better alternative because it enhances readability and makes it possible to reduce the accessibility of the classes by declaring them private (Item 15). Here is how our example looks with static member classes:

```
// Static member classes instead of multiple top-level classes
public class Test {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }

    private static class Utensil {
        static final String NAME = "pan";
    }

    private static class Dessert {
        static final String NAME = "cake";
    }
}
```

The lesson is clear: **Never put multiple top-level classes or interfaces in a single source file.** Following this rule guarantees that you can't have multiple definitions for a single class at compile time. This in turn guarantees that the class files generated by compilation, and the behavior of the resulting program, are independent of the order in which the source files are passed to the compiler.