**Minimize the accessibility of classes and members**

A well designed component hides all its implementation details, cleanly separating its API from its implementation. this concept know as **information hiding** or **encapsulation** is a fundamental tenet of software design.

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 packageprivate.

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.

## Why is it important ?

* Decouples the components, so better isolation
* Eases the burden of maintenance
* Better profiling for performance problems
* Decreases the risk of building huge systems

## Rules

* Make each class or member as inaccessible as possible.
* Make all the members private
* If a method overrides a superclass method, it cannot have more restrictive access level in the subclass than in the superclass.
* It is accessible to make a member package-private in order to test it
* Instance fields of public classes should rarely be public
* You can expose constants via public static final fields. Note that’s wrong to have non zero length array as a constant, because it’s always mutable. Prefer then using Collections.unmodifiableList(Arrays.asList(PRIVATE\_VALUES)).

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.

**In public classes, use accessor methods, not public fields**

If a class is accessible outside the package, provide accessor methods to preserve the flexibility of the class’s internal representation. However, if a class is package-private or is nested class, there is nothing inherently wrong with exposing its data fields.

While it is never a good idea for a public class to expose fields directly, it is less harmful if the fields are immutable.

public class Person {

public final String name;

public final int age ;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

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.

**Minimize mutability**

// 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

}

To make a class immutable:

* Don’t provide methods that modify the object state
* Make all fields private final
* Ensure that classes can not be extended.

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.

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.

**Favor composition over inheritance**

**Inheritance** is a powerful way to achieve code reuse. It is safe to use inheritance within the same package, but, it is dangerous to use inheritance cross packages. It is suitable when the relation between the 2 classes is is-a relationship.

Unlike method invocation, inheritance violates encapsulation; The subclass depends on the superclass implementation.

**Composition** is giving the class a private field that references an instance of the existing class. **Forwarding** : Each instance method in the new class invokes the corresponding method on the contained instance :

public class ForwardingSet<T> implements Set<T> {

private final Set<T> s ;

public ForwardingSet(Set<T> s) {

this.s = s;

}

@Override

public int size() {

return s.size();

}

@Override

public boolean isEmpty() {

return s.isEmpty();

}

// Other methods ...

}

**Wrapper** : The InstrumentedSet is know as wrapper because each InstrumentedSet instance contains another Set instance. This is also know as **Decorator** pattern, because it decorates a set by adding instrumentation.

public class InstrumentedSet<T> extends ForwardingSet<T> {

private int count = 0 ;

public InstrumentedSet(Set<T> s) {

super(s);

}

@Override

public boolean add(T t) {

count ++ ;

return super.add(t);

}

@Override

public boolean addAll(Collection<? extends T> c) {

count += c.size() ;

return super.addAll(c);

}

public int getCount() {

return count;

}

}

Sometimes, the combination of Forwarding and Composition is loosely referred to **Delegation**. Technically, it’s not delegation unless the wrapper object passes itself to the wrapped object.

For example, Guava provides forwarding classes for all the collection interface.

If you use inheritance where composition is appropriate, you needlessly expose implementation details, and the resulting API ties you to the original implementation.

**Design and document for inheritance or else prohibit it**

The class must document its self-use of overridable methods. The documentation must indicate which overridable method the method invokes, in what sequence, and how the result of each invocation affects subsequent processing.The @implSpec tag was added in java 8.

## Advices

* A class may have to provide hooks into its internal working in the form of protected method.
* The only way to test a class designed for inheritance is to write subclass, and you should do it before you release it.
* The constructor must not invoke overridable methods, directly or indirectly.
* If you decide to implement Cloneable or Serializable :
  + neither clone nor readObject may invoke overridable methods, directly or indirectly.
  + if the class has readResolve or writeReplace method, make this method protected, otherwise, it will be silently ignored by the subclass.
* Prohibit subclassing in classes that are not designed and documented to be the safety subclassed. Otherwise, each time the class changes, there’s a chance that subclasses extending the class will break.
  + Declare final class or make all constructor private or package-private and add public static factories.

**Prefer Interface over Abstract class.**

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 here 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.

Default methods are not permitted to override Object methods such as equals, hashCode, and toString

**Design Interface for prosperity.**

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.

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).

**Use Interface 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;

}

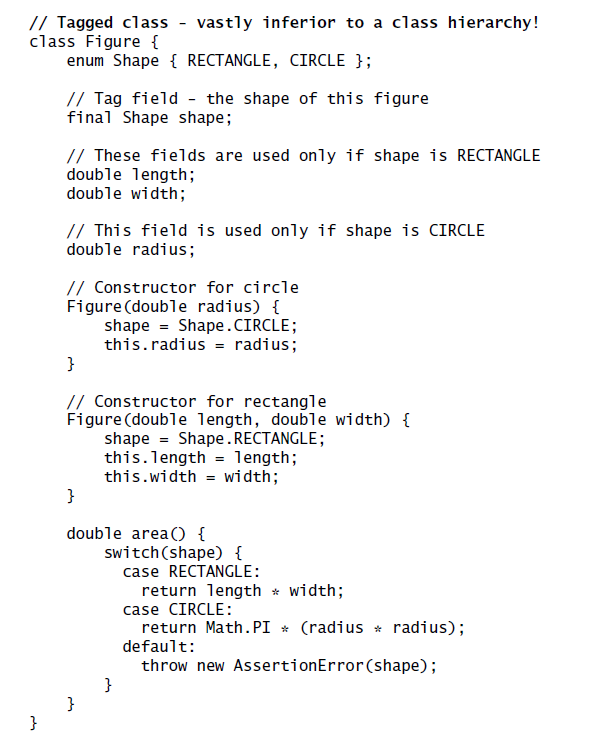
The constant interface pattern is a poor use of interfaces. That a class uses some constants internally is an implementation detail. Implementing a constant interface causes this implementation detail to leak into the class’s exported API. It is of no consequence to the users of a class that the class 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 anbomalies and should not be emulated.

In summary, interfaces should be used only to define types. They should not be used merely to export constants.

**Prefer class hierarchies to tagged classes**

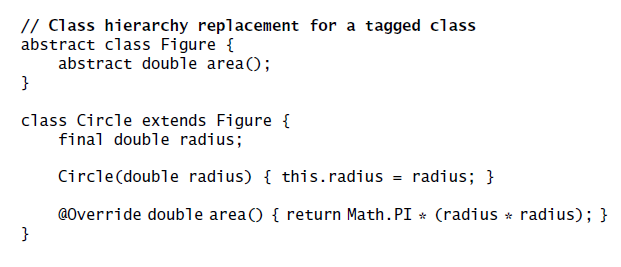
Occasionally you may run across a class whose instances come in two or more flavors and contain a tag field indicating the flavor of the instance. For example, consider this class, which is capable of representing a circle or a rectangle:

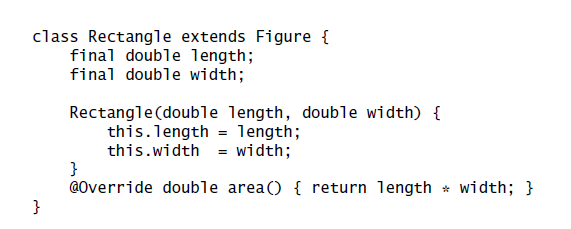


Such tagged classes have numerous shortcomings. They are cluttered with boilerplate, including enum declarations, tag fields, and switch statements. Readability is further harmed because multiple implementations are jumbled together in a single class. Memory footprint is increased because instances are 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 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.

**Favor static member classes over nonstatic**

A nested class is a class defined in another class and should exists only to serve its enclosing class. There are 4 types :

## Non static member class

Each instance of non static member class is implicitly associated with an enclosing instance of its containing class. It is impossible to create an instance of a non static member class without an enclosing instance.

One common use of non static class is to define Adapter.

A screenshot of a computer program

Description automatically generated with low confidence

## static class

One common use of static member class is public helper class. If you declare a member class that does not require access to an enclosing instance, always put the static modifier in its declaration.

A picture containing text, screenshot, font

Description automatically generated

## Anonymous classes

class with no name. It is not member of its enclosing class. It was very used before Lambdas

A screenshot of a computer program

Description automatically generated with medium confidence

## Local classes

Local classes can be declared anywhere a local variable is declared and obeys the same scoping rules. They have names and can be used repeatedly.

A picture containing text, screenshot, font

Description automatically generated

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.

**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):

A screenshot of a computer program

Description automatically generated with low confidence

Problem will arise when you create a new Dessert.java file and output will random based on the way it gets compiled. Hence prefer static inner class definition instead of defining multiple class in a same java file.

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.