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.