Joshua Bloch

Updated for Java 9

Effective Java

Third Edition

Best practices for

...the Java Platform

Effective Java

Third Edition

```
        // 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 7). 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 or 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.