



Java Magazine

Written by the Java community for Java and JVM developers

Coding, Design Patterns, JVM Internals

You should favor composition over inheritance in Java. Here's why.

July 14, 2022 | 12 minute read



Joshua Bloch

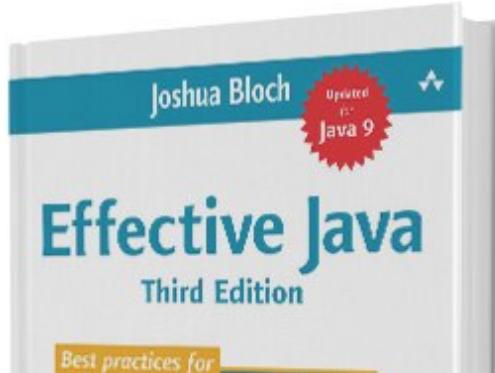


Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job.

[The following is adapted from *Effective Java, third edition* (Addison-Wesley, 2017) by Joshua Bloch, reprinted here with permission. —Ed.]

Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, inheritance 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.

Inheriting from ordinary concrete classes across

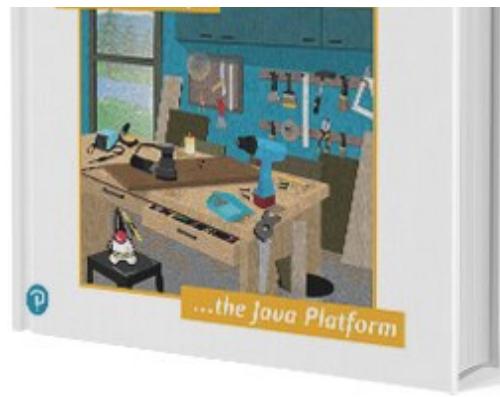


....., ~~but~~ ~~cross~~ ~~class~~ ~~boundaries~~

package boundaries, however, is dangerous.

By the way, this article uses the word *inheritance* to mean *implementation inheritance* (when one class extends another). The problems discussed in this article do not apply to *interface inheritance* (when a class implements an interface or when one interface extends another).

Unlike method invocation, inheritance violates encapsulation.



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 you have a program that uses a `HashSet`. To tune the performance of the program, you need to query the `HashSet` to determine 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, you 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 you override both methods, as follows:

```
// 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;
    }
}
```

[Copy code snippet](#)

This class looks reasonable, but it doesn't work. Suppose you create an instance and add three elements using the `addAll` method. (Incidentally, note that you 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"));
```

[Copy code snippet](#)

You would expect the `getAddCount` method to return 3 at this point, but it returns 6. What went wrong? Internally, the `HashSet` `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 the `HashSet` `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 counted twice.

You 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 the `HashSet` `addAll` method is implemented on top of its `add` method. This self-use is an implementation detail that is not guaranteed to hold in all implementations of the Java platform and is 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 the `HashSet` `addAll` method were implemented atop its `add` method because the `HashSet` `addAll` implementation would no longer be invoked. This technique, however, does not solve all the 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 that are 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 gave the subclass a method with the same signature and a different return type, your subclass will no longer compile (see the [Java Language Specification](#), section 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 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.

Solving the problem

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 and have 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, below is 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 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(); }
}

```

[Copy code snippet](#)

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 `InstrumentedSet` 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 with any preexisting constructor.

```

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

```

[Copy code snippet](#)

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

```
static void walk(Set<Door> doors) {
```

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

[Copy code snippet](#)

The `InstrumentedSet` class is known as a *wrapper* class because each `InstrumentedSet` instance contains (or “wraps”) another set instance. This is also known as the *Decorator pattern* 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.

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*. 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, [Google’s Guava core libraries for Java](#) provides forwarding classes for all of the collection interfaces.

Subtypes of superclasses

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 this 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. In other words, `A` is not an essential part of `B`; it’s merely a detail of its implementation.

(There are a number of 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 than `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 Java 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 (such as `load` and `store`). By the time this problem was discovered, it was too late to correct it because clients depended on the use of nonstring keys and values.

Conclusion

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, but they are also more powerful.

Dig deeper

- [The evolving nature of Java interfaces](#)
- [The Decorator pattern in depth](#)
- [Exploring Joshua Bloch's Builder design pattern in Java](#)
- [Quiz yourself: Final classes](#)



Joshua Bloch

[Joshua Bloch](#) is adjunct professor of the practice at Institute for Software Research at Carnegie Mellon University (CMU).

Bloch has decades of experience in software research and development. After completing his PhD, he joined Transarc, a CMU spinoff, as senior systems designer. He later became a distinguished engineer at Sun Microsystems where he wrote the Collections library for Java. He then left Sun to join Google as chief [Show more](#)

◀ Previous Post

Next Post ▶

Resources for	Why Oracle	Learn	What's New	Contact Us
About	Analyst Reports	What is Customer Service?	Try Oracle Cloud Free Tier	US Sales 1.800.633.0738
Careers	Best CRM	What is ERP?	Oracle Sustainability	How can we help?
Developers	Cloud Economics	What is Marketing Automation?	Oracle COVID-19 Response	Subscribe to Oracle Content
Investors	Corporate Responsibility	What is Procurement?	Oracle and SailGP	Try Oracle Cloud Free Tier
Partners	Diversity and Inclusion	What is Talent Management?	Oracle and Premier League	Events
Startups	Security Practices	What is VM?	Oracle and Red Bull Racing Honda	News