Joshua Bloch

Updated for Java 9

Effective Java

Third Edition

Best practices for

...the Java Platform

Item 8: Avoid finalizers and cleaners

Finalizers are unpredictable, often dangerous, and generally unnecessary. Their use can cause erratic behavior, poor performance, and portability problems. Finalizers have a few valid uses, which we'll cover later in this item, but as a rule, you should avoid them. As of Java 9, finalizers have been deprecated, but they are still being used by the Java libraries. The Java 9 replacement for finalizers is *cleaners*. **Cleaners are less dangerous than finalizers, but still unpredictable, slow, and generally unnecessary.**

C++ programmers are cautioned not to think of finalizers or cleaners as Java's analogue of C++ destructors. In C++, destructors are the normal way to reclaim the resources associated with an object, a necessary counterpart to constructors. In Java, the garbage collector reclaims the storage associated with an object when it becomes unreachable, requiring no special effort on the part of the programmer. C++ destructors are also used to reclaim other nonmemory resources. In Java, a `try`-with-resources or `try-finally` block is used for this purpose (Item 9).

One shortcoming of finalizers and cleaners is that there is no guarantee they'll be executed promptly [JLS, 12.6]. It can take arbitrarily long between the time that an object becomes unreachable and the time its finalizer or cleaner runs. This means that you should **never do anything time-critical in a finalizer or cleaner.** For example, it is a grave error to depend on a finalizer or cleaner to close files because open file descriptors are a limited resource. If many files are left open as a result of the system's tardiness in running finalizers or cleaners, a program may fail because it can no longer open files.

The promptness with which finalizers and cleaners are executed is primarily a function of the garbage collection algorithm, which varies widely across implementations. The behavior of a program that depends on the promptness of finalizer or cleaner execution may likewise vary. It is entirely possible that such a program will run perfectly on the JVM on which you test it and then fail miserably on the one favored by your most important customer.

Tardy finalization is not just a theoretical problem. Providing a finalizer for a class can arbitrarily delay reclamation of its instances. A colleague debugged a long-running GUI application that was mysteriously dying with an `OutOfMemoryError`. Analysis revealed that at the time of its death, the application had thousands of graphics objects on its finalizer queue just waiting to be finalized and reclaimed. Unfortunately, the finalizer thread was running at a lower priority than another application thread, so objects weren't getting finalized at the rate they became eligible for finalization. The language specification makes no guar-

antees as to which thread will execute finalizers, so there is no portable way to prevent this sort of problem other than to refrain from using finalizers. Cleaners are a bit better than finalizers in this regard because class authors have control over their own cleaner threads, but cleaners still run in the background, under the control of the garbage collector, so there can be no guarantee of prompt cleaning.

Not only does the specification provide no guarantee that finalizers or cleaners will run promptly; it provides no guarantee that they'll run at all. It is entirely possible, even likely, that a program terminates without running them on some objects that are no longer reachable. As a consequence, you should **never depend on a finalizer or cleaner to update persistent state.** For example, depending on a finalizer or cleaner to release a persistent lock on a shared resource such as a database is a good way to bring your entire distributed system to a grinding halt.

Don't be seduced by the methods `System.gc` and `System.runFinalization`. They may increase the odds of finalizers or cleaners getting executed, but they don't guarantee it. Two methods once claimed to make this guarantee: `System.runFinalizersOnExit` and its evil twin, `Runtime.runFinalizersOnExit`. These methods are fatally flawed and have been deprecated for decades [ThreadStop].

Another problem with finalizers is that an uncaught exception thrown during finalization is ignored, and finalization of that object terminates [JLS, 12.6]. Uncaught exceptions can leave other objects in a corrupt state. If another thread attempts to use such a corrupted object, arbitrary nondeterministic behavior may result. Normally, an uncaught exception will terminate the thread and print a stack trace, but not if it occurs in a finalizer—it won't even print a warning. Cleaners do not have this problem because a library using a cleaner has control over its thread.

There is a *severe* performance penalty for using finalizers and cleaners. On my machine, the time to create a simple `AutoCloseable` object, to close it using `try`-with-resources, and to have the garbage collector reclaim it is about 12 ns. Using a finalizer instead increases the time to 550 ns. In other words, it is about 50 times slower to create and destroy objects with finalizers. This is primarily because finalizers inhibit efficient garbage collection. Cleaners are comparable in speed to finalizers if you use them to clean all instances of the class (about 500 ns per instance on my machine), but cleaners are much faster if you use them only as a safety net, as discussed below. Under these circumstances, creating, cleaning, and destroying an object takes about 66 ns on my machine, which means you pay a factor of five (not fifty) for the insurance of a safety net *if* you don't use it.

Finalizers have a serious security problem: they open your class up to *finalizer attacks*. The idea behind a finalizer attack is simple: If an exception is

thrown from a constructor or its serialization equivalents—the `readObject` and `readResolve` methods (Chapter 12)—the finalizer of a malicious subclass can run on the partially constructed object that should have "died on the vine." This finalizer can record a reference to the object in a static field, preventing it from being garbage collected. Once the malformed object has been recorded, it is a simple matter to invoke arbitrary methods on this object that should never have been allowed to exist in the first place. **Throwing an exception from a constructor should be sufficient to prevent an object from coming into existence; in the presence of finalizers, it is not.** Such attacks can have dire consequences. Final classes are immune to finalizer attacks because no one can write a malicious subclass of a final class. **To protect nonfinal classes from finalizer attacks, write a final `finalize` method that does nothing.**

So what should you do instead of writing a finalizer or cleaner for a class whose objects encapsulate resources that require termination, such as files or threads? Just **have your class implement `AutoCloseable`,** and require its clients to invoke the `close` method on each instance when it is no longer needed, typically using `try`-with-resources to ensure termination even in the face of exceptions (Item 9). One detail worth mentioning is that the instance must keep track of whether it has been closed: the `close` method must record in a field that the object is no longer valid, and other methods must check this field and throw an `IllegalStateException` if they are called after the object has been closed.

So what, if anything, are cleaners and finalizers good for? They have perhaps two legitimate uses. One is to act as a safety net in case the owner of a resource neglects to call its `close` method. While there's no guarantee that the cleaner or finalizer will run promptly (or at all), it is better to free the resource late than never if the client fails to do so. If you're considering writing such a safety-net finalizer, think long and hard about whether the protection is worth the cost. Some Java library classes, such as `FileInputStream`, `FileOutputStream`, `ThreadPoolExecutor`, and `java.sql.Connection`, have finalizers that serve as safety nets.

A second legitimate use of cleaners concerns objects with *native peers*. A native peer is a native (non-Java) object to which a normal object delegates via native methods. Because a native peer is not a normal object, the garbage collector doesn't know about it and can't reclaim it when its Java peer is reclaimed. A cleaner or finalizer may be an appropriate vehicle for this task, assuming the performance is acceptable and the native peer holds no critical resources. If the performance is unacceptable or the native peer holds resources that must be reclaimed promptly, the class should have a `close` method, as described earlier.

Cleaners are a bit tricky to use. Below is a simple `Room` class demonstrating the facility. Let's assume that rooms must be cleaned before they are reclaimed. The `Room` class implements `AutoCloseable`; the fact that its automatic cleaning safety net uses a cleaner is merely an implementation detail. Unlike finalizers, cleaners do not pollute a class's public API:

```
// An autocloseable class using a cleaner as a safety net
public class Room implements AutoCloseable {
    private static final Cleaner cleaner = Cleaner.create();

    // Resource that requires cleaning. Must not refer to Room!
    private static class State implements Runnable {
        int numJunkPiles; // Number of junk piles in this room

        State(int numJunkPiles) {
            this.numJunkPiles = numJunkPiles;
        }

        // Invoked by close method or cleaner
        @Override public void run() {
            System.out.println("Cleaning room");
            numJunkPiles = 0;
        }
    }

    // The state of this room, shared with our cleanable
    private final State state;

    // Our cleanable. Cleans the room when it's eligible for gc
    private final Cleaner.Cleanable cleanable;

    public Room(int numJunkPiles) {
        state = new State(numJunkPiles);
        cleanable = cleaner.register(this, state);
    }

    @Override public void close() {
        cleanable.clean();
    }
}
```

The static nested `State` class holds the resources that are required by the cleaner to clean the room. In this case, it is simply the `numJunkPiles` field, which represents the amount of mess in the room. More realistically, it might be a final `long` that contains a pointer to a native peer. `State` implements `Runnable`, and its `run` method is called at most once, by the `Cleanable` that we get when we register our `State` instance with our cleaner in the `Room` constructor. The call to the `run` method will be triggered by one of two things: Usually it is triggered by a call to

`Room`'s `close` method calling `Cleanable`'s clean method. If the client fails to call the `close` method by the time a `Room` instance is eligible for garbage collection, the cleaner will (hopefully) call `State`'s `run` method.

It is critical that a `State` instance does not refer to its `Room` instance. If it did, it would create a circularity that would prevent the `Room` instance from becoming eligible for garbage collection (and from being automatically cleaned). Therefore, `State` must be a *static* nested class because nonstatic nested classes contain references to their enclosing instances (Item 24). It is similarly inadvisable to use a lambda because they can easily capture references to enclosing objects.

As we said earlier, `Room`'s cleaner is used only as a safety net. If clients surround all `Room` instantiations in `try`-with-resource blocks, automatic cleaning will never be required. This well-behaved client demonstrates that behavior:

```
public class Adult {
    public static void main(String[] args) {
        try (Room myRoom = new Room(7)) {
            System.out.println("Goodbye");
        }
    }
}
```

As you'd expect, running the `Adult` program prints `Goodbye`, followed by `Cleaning room`. But what about this ill-behaved program, which never cleans its room?

```
public class Teenager {
    public static void main(String[] args) {
        new Room(99);
        System.out.println("Peace out");
    }
}
```

You might expect it to print `Peace out`, followed by `Cleaning room`, but on my machine, it never prints `Cleaning room`; it just exits. This is the unpredictability we spoke of earlier. The `Cleaner` spec says, "The behavior of cleaners during `System.exit` is implementation specific. No guarantees are made relating to whether cleaning actions are invoked or not." While the spec does not say it, the same holds true for normal program exit. On my machine, adding the line `System.gc()` to `Teenager`'s `main` method is enough to make it print `Cleaning room` prior to exit, but there's no guarantee that you'll see the same behavior on your machine.

In summary, don't use cleaners, or in releases prior to Java 9, finalizers, except as a safety net or to terminate noncritical native resources. Even then, beware the indeterminacy and performance consequences.