http://www.devx.com/Java/Article/30192

How to Handle Java Finalization's Memory-Retention Issues

By Tony Printezis
Dec 27, 2005

Finalization is a feature of the Java programming language that allows you to perform postmortem cleanup on objects that the garbage collector has found to be unreachable. It is typically used to reclaim native resources associated with an object. The following is a simple finalization example:

```
public class Image1 {
        // pointer to the native image data
private int nativeImg;
        private Point pos;
        private Dimension dim;

        // it disposes of the native image;
        // successive calls to it will be ignored
        private native void disposeNative();
        public void dispose() { disposeNative(); }
        protected void finalize() { dispose(); }

        static private Image1 randomImg;
}
```

Some time after an `Image1` instance has become unreachable, the Java virtual machine (JVM) will call its `finalize()` method to ensure that the native resource that holds the image data (pointed to by the integer `nativeImg` in the example) has been reclaimed. Notice, however, that the `finalize()` method, despite its special treatment by the JVM, is an arbitrary method that contains arbitrary code. In particular, it can access any field in the object (`pos` and `dim` in the example). Surprisingly, it can also make the object reachable again by, say, making it reachable from a static field (e.g., `randomImg = this;`). I really don't recommend the latter programming practice, but unfortunately the Java programming language allows it.

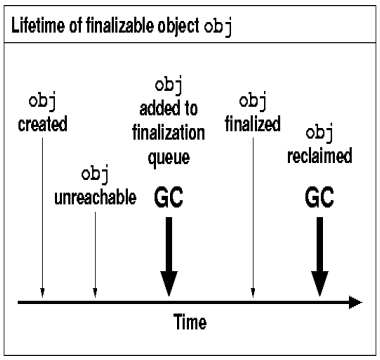
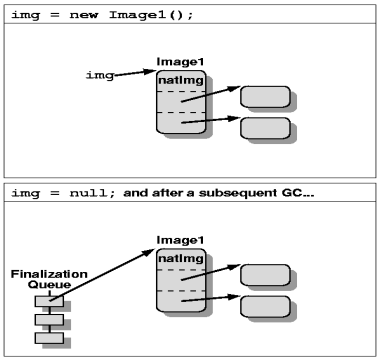


Figure 1. A Finalizable Object `obj`

The following steps describe the lifetime of a *finalizable object* `obj`—that is, an object whose class has a non-trivial finalizer (see Figure 1):

1. When `obj` is allocated, the JVM internally records that `obj` is finalizable (this typically slows down the otherwise fast allocation path that modern JVMs have).
2. When the garbage collector determines that `obj` is unreachable, it notices that `obj` is finalizable (as it had been recorded upon allocation) and adds it to the JVM's *finalization queue*. It also ensures that all objects reachable from `obj` are retained, even if they are otherwise unreachable, as they might be accessed by the finalizer. Figure 2 illustrates this for an instance of `Image1`.
3. At some point later, the JVM's *finalizer thread* will dequeue



`obj`, call its `finalize()` method, and record that `obj`'s finalizer has been called. At this point, `obj` is considered to be *finalized*.

Figure 2. Garbage Collector Determines That `obj` Is Unreachable

4. When the garbage collector rediscovers that `obj` is unreachable, it will reclaim its space along with everything reachable from it (provided that the latter is otherwise unreachable).

Notice that the garbage collector needs a minimum of two cycles (maybe more) to reclaim `obj` and needs to retain all other objects reachable from `obj` during this process. If a programmer is not careful, this can create temporary, subtle, and unpredictable resource-retention issues. Additionally, the JVM does not guarantee that it will call the finalizers of all the finalizable objects that have been allocated; it might exit before the garbage collector discovers some of them to be unreachable.

Avoid Memory-Retention Problems When Subclassing

Finalization can delay the reclamation of resources, even if you do not use it explicitly. Consider the following example:

```
public class RGBImage1 extends Image1 {
        private byte rgbData[];
}
```

`RGBImage1` extends `Image1` and introduces field `rgbData` (and maybe some methods the example doesn't show). Even though you did not explicitly define a finalizer on `RGBImage1`, the class will naturally inherit the `finalize()` method from `Image1`, and all `RGBImage1` instances will also be considered to be finalizable. When an `RGBImage1` instance becomes unreachable, the reclamation of the potentially very large `rgbData` array will be delayed until the instance is finalized (see Figure 3). It can be a difficult problem to find because the finalizer might be "hidden" in a deep class hierarchy.

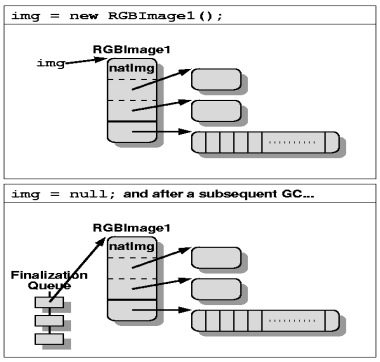


Figure 3. Reclamation of `rgbData` Array Will Be Delayed Until the Instance Is Finalized

One way to avoid this problem is to re-arrange the code so that it uses the "contains," instead of the "extends," pattern, as follows:

```
public class RGBImage2 {
        private Image1 img;
        private byte rgbData[];

        public void dispose() {
                img.dispose();
        }
}
```

Compared with `RGBImage1`, `RGBImage2` contains an instance of `Image1` instead of extending `Image1`. When an instance of `RGBImage2` becomes unreachable, the garbage collector will promptly reclaim it, along with the `rgbData` array (assuming the latter is not reachable from anywhere else), and will queue up only the `Image1` instance for finalization (see Figure 4). Since class `RGBImage2` does not subclass `Image1`, it will not inherit any methods from it. Therefore, you might have to add delegator methods to `RGBImage1` to access the required methods of `Image1` (the `dispose()` method is such an example).

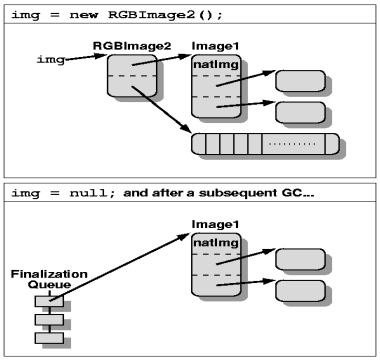


Figure 4. GC Will Queue Up Only the `Image1` Instance for Finalization

You cannot always re-arrange your code in the manner described above, however. In that case, as a user of the class, you will have to do a little more work to ensure that its instances do not hold on to more space than necessary when they are being finalized. The following code illustrates how:

```
public class RGBImage3 extends Image1 {
        private byte rgbData[];

        public void dispose() {
                super.dispose();
                rgbData = null;
        }
}
```

`RGBImage3` is identical to `RGBImage1` but with the addition of the `dispose()` method, which nulls the `rgbData` field. You are required to explicitly call `dispose()` after using an `RGBImage3` instance to ensure that the `rgbData` array is promptly reclaimed (see Figure 5). I recommend explicit nulling of fields on very few occasions; this is one of them.

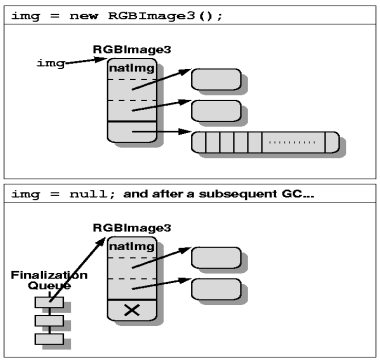


Figure 5. Call `dispose()` After Using an `RGBImage3` Instance

Shield Users from Memory-Retention Problems

The previous section described how to avoid memory-retention problems when working with third-party classes that use finalizers. This section describes how to write classes that require postmortem cleanup so that their users don't encounter the problems previously outlined. The best way to do so is to split such classes into two (one to hold the data that need postmortem cleanup, the other to hold everything else) and define a finalizer only on the former. The following code illustrates this technique:

```
final class NativeImage2 {
        // pointer to the native image data
        private int nativeImg;

        // it disposes of the native image;
        // successive calls to it will be ignored
        private native void disposeNative();
        void dispose() { disposeNative(); }
        protected void finalize() { dispose(); }
}

public class Image2 {
        private NativeImage2 nativeImg;
        private Point pos;
        private Dimension dim;

        public void dispose() { nativeImg.dispose(); }
}
```

`Image2` is similar to `Image1`, but with the `nativeImg` field included in a separate class, `NativeImage2`. All accesses to `nativeImg` from the image class must go through one level of indirection. However, when an `Image2` instance becomes unreachable, only the `NativeImage2` instance will be queued up for finalization; anything else reachable from the `Image2` instance will be promptly reclaimed (see Figure 6). Class `NativeImage2` is declared to be `final` so that users cannot subclass it and re-introduce the memory-retention problems described in the previous section.

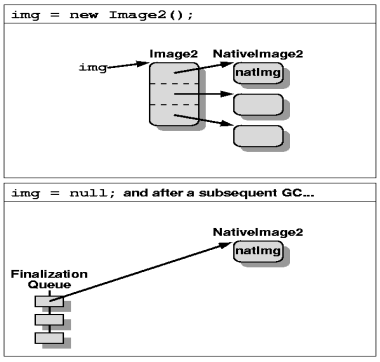


Figure 6. When `Image2` Instance Becomes Unreachable, Only the `NativeImage2` Instance Will Be Queued Up

A subtle point is that `NativeImage2` should *not* be an inner class of `Image2`. Instances of inner classes have an implicit reference to the instance of the outer class that created them. Therefore, if `NativeImage2` was an inner class of `Image2`, and a `NativeImage2` instance was queued up for finalization, it would have also retained the corresponding `Image2` instance, which is precisely what you are trying to avoid. Assume, however, that the `NativeImage2` class will be accessible only from the `Image2` class. This is the reason why it has no public methods (its `dispose()` method, as well as

the class itself, is package-private).

An Alternative to Finalization

The example in the previous section still has one source of non-determinism: the JVM does *not* guarantee the order in which it will call the finalizers of the objects in the finalization queue. And finalizers from all classes (application, libraries, etc.) are treated equally. So, an object that is holding on to a lot of memory or a scarce native resource can get stuck in the finalization queue behind objects whose finalizers are making slow progress (not necessarily maliciously; maybe due to sloppy programming).

To avoid this type of non-determinism, you can use weak references, instead of finalization, as the postmortem hook. This way, you have total control over how to prioritize the reclamation of native resources, instead of relying on the JVM to do so. The following example illustrates this technique:

```
final class NativeImage3 extends WeakReference<Image3> {
        // pointer to the native image data
        private int nativeImg;

        // it disposes of the native image;
        // successive calls to it will be ignored
        private native void disposeNative();
        void dispose() {
                disposeNative();
                refList.remove(this);
        }

        static private ReferenceQueue<Image3> refQueue;
        static private List<NativeImage3> refList;
        static ReferenceQueue<Image3> referenceQueue() {
                return refQueue;
        }

        NativeImage3(Image3 img) {
                super(img, refQueue);
                refList.add(this);
        }
}

public class Image3 {
        private NativeImage3 nativeImg;
        private Point pos;
        private Dimension dim;

        public void dispose() { nativeImg.dispose(); }
}
```

`Image3` is identical to `Image2`. `NativeImage3` is similar to `NativeImage2`, but its postmortem cleanup relies on weak references instead of finalization. `NativeImage3` extends `WeakReference`, whose referent is the associated `Image3` instance. Remember that when the referent of a reference object (in this case a `WeakReference`) becomes unreachable, the reference object is added to the reference queue associated with it. Embedding `nativeImg` into the reference object itself ensures that the JVM will enqueue exactly what is needed and nothing more (see Figure 7). Again, `NativeImage3` should *not* be an inner class of `Image3`, for the reasons previously outlined.

You can determine whether the referent of a reference object has been reclaimed by the garbage collector in two ways: explicitly, by calling the `get()` method on the reference object, or implicitly, by noticing that the reference object has been enqueued on the associated reference queue. This example uses only the latter.

Notice that reference objects are discovered only by the garbage collector and added to their associated references queues only if

they are reachable themselves. Otherwise, they are simply reclaimed like any other unreachable object. This is why you add all `NativeImage3` instances to the static list (actually, any data structure will suffice): to ensure that they remain reachable and processed when their referents become unreachable. Naturally, you also have to make sure that you remove them from the list when you dispose of them (this is done in the `dispose()` method).

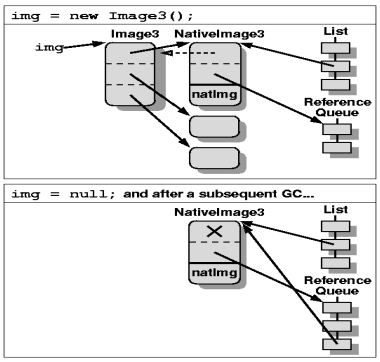


Figure 7. Embedding `nativeImg` into the Reference Object Itself

When the `dispose()` method is explicitly called on an `Image3` instance, no postmortem cleanup will subsequently take place on that instance; correctly so, too, as none is necessary. The `dispose()` method removes the `NativeImage3` instance from the static list so that it is not reachable when its corresponding `Image3` instance becomes unreachable. And, as previously stated, unreachable reference objects are not added to their corresponding reference queues. In contrast, in all the previous examples that use finalization, the finalizable objects will always be considered for finalization when they become unreachable, whether you have explicitly disposed of their associated native resources or not.

The JVM will ensure that when an `Image3` instance is found to be unreachable by the garbage collector it adds its corresponding `NativeImage3` instance to its associated reference queue. It is then up to you to dequeue it and dispose of its native resource. This can be done with a loop like the following, executed, say, on a "clean up" thread:

```
ReferenceQueue<Image3> refQueue =
        NativeImage3.referenceQueue();
while (true) {
        NativeImage3 nativeImg =
                (NativeImage3) refQueue.remove();
        nativeImg.dispose();
}
```

This is a simplistic example. Sophisticated developers can also ensure that different reference objects are associated with different reference queues, according to how they need to prioritize their disposal. And a single "clean up" thread can poll all the available reference queues and dequeue objects according to the required priorities. Additionally, you can choose to spread out the reclamation of resources so that it is less disruptive to the application.

Even though cleaning up resources in this way is clearly a more involved process than using finalization, it is also more powerful and more flexible, and it minimizes a lot of the non-determinism associated with the use of finalization. It is also very similar to the way finalization is actually implemented within the JVM. I recommend this approach for projects that explicitly use a lot of native resources and need more control when cleaning them up. Using finalization with care would suffice for most other projects.

[**Note:** This article covered only two types of issues that arise when using finalization, namely memory- and resource-retention issues. The use of finalization and the `Reference` classes can also cause very subtle synchronization problems. Read Hans-J. Boehm's *Finalization, Threads, and the Java Technology-Based Memory Model* for a good overview of these.]

Use Finalization Only When You Must

This article briefly described how finalization is implemented in a JVM. It then gave examples of how memory can be unnecessarily retained by finalizable objects and outlined solutions to such problems. Finally, it described a method that uses weak references instead of finalization, which allows you to perform postmortem cleanup in a more flexible and predictable manner.

However, total reliance on the garbage collector to identify unreachable objects so that their associated native—and potentially scarce—resources can be reclaimed has a serious flaw: memory is typically plentiful, and guarding a potentially scarce resource with a plentiful one is not a good strategy. So, when you use an object that you know has native resources associated with it (e.g., a GUI component, file, socket), by all means call its `dispose()` or equivalent method when you are done with it. This will ensure the immediate reclamation of the native resources

and decrease the probability of those resources running out. This way, you will use the approaches discussed in this article for postmortem cleanup only as last resorts and not as main cleanup mechanisms.

You should also try to limit your use of finalization to only when it is absolutely necessary. Finalization is a non-deterministic—and sometimes unpredictable—process. The less you rely on it, the smaller the impact it will have on the JVM and your application.

Acknowledgments

The author is grateful to Peter Kessler for his many constructive comments on this article.