Understanding Weak References
Ethan Nicholas

Some time ago I was interviewing candidates for a Senior Java Engineer position. Among the many questions I asked was "What can you tell me about weak references?" I wasn't expecting a detailed technical treatise on the subject. I would probably have been satisfied with "Umm... don't they have something to do with garbage collection?" I was instead surprised to find that out of twenty-odd engineers, all of whom had at least five years of Java experience and good qualifications, only *two* of them even knew that weak references existed, and only one of those two had actual useful knowledge about them. I even explained a bit about them, to see if I got an "Oh yeah" from anybody -- nope. I'm not sure why this knowledge is (evidently) uncommon, as weak references are a massively useful feature which have been around since Java 1.2 was released, over seven years ago.

Now, I'm not suggesting you need to be a weak reference expert to qualify as a decent Java engineer. But I humbly submit that you should at least *know what they are* -- otherwise how will you know when you should be using them? Since they seem to be a little-known feature, here is a brief overview of what weak references are, how to use them, and when to use them.

Strong references

First I need to start with a refresher on *strong references*. A strong reference is an ordinary Java reference, the kind you use every day. For example, the code:

```
StringBuffer buffer = new StringBuffer();
```

creates a new `StringBuffer()` and stores a strong reference to it in the variable `buffer`. Yes, yes, this is kiddie stuff, but bear with me. The important part about strong references -- the part that makes them "strong" -- is how they interact with the garbage collector. Specifically, if an object is reachable via a chain of strong references (strongly reachable), it is not eligible for garbage collection. As you don't want the garbage collector destroying objects you're working on, this is normally exactly what you want.

When strong references are too strong

It's not uncommon for an application to use classes that it can't reasonably extend. The class might simply be marked final, or it could be something more complicated, such as an interface returned by a factory method backed by an unknown (and possibly even unknowable) number of concrete implementations. Suppose you have to use a class `Widget` and, for whatever reason, it isn't possible or practical to extend `Widget` to add new functionality.

What happens when you need to keep track of extra information about the object? In this case, suppose we find ourselves needing to keep track of each Widget's serial number, but the Widget class doesn't actually have a serial number property -- and because Widget isn't extensible, we can't add one. No problem at all, that's what HashMaps are for:

```
serialNumberMap.put(widget, widgetSerialNumber);
```

This might look okay on the surface, but the strong reference to widget will almost certainly cause problems. We have to know (with 100% certainty) when a particular Widget's serial number is no longer needed, so we can remove its entry from the map. Otherwise we're going to have a memory leak (if we don't remove Widgets when we should) or we're going to inexplicably find ourselves missing serial numbers (if we remove Widgets that we're still using). If these problems sound familiar, they should: they are exactly the problems that users of non-garbage-collected languages face when trying to manage memory, and we're not supposed to have to worry about this in a more civilized language like Java.

Another common problem with strong references is caching, particular with very large structures like images. Suppose you have an application which has to work with user-supplied images, like the web site design tool I work on. Naturally you want to cache these images, because loading them from disk is very expensive and you want to avoid the possibility of having two copies of the (potentially gigantic) image in memory at once.

Because an image cache is supposed to prevent us from reloading images when we don't absolutely need to, you will quickly realize that the cache should always contain a reference to any image which is already in memory. With ordinary strong references, though, that reference itself will force the image to remain in memory, which requires you (just as above) to somehow determine when the image is no longer needed in memory and remove it from the cache, so that it becomes eligible for garbage collection. Once again you are forced to duplicate the behavior of the garbage collector and manually determine whether or not an object should be in memory.

Weak references

A *weak reference*, simply put, is a reference that isn't strong enough to force an object to remain in memory. Weak references allow you to leverage the garbage collector's ability to determine reachability for you, so you don't have to do it yourself. You create a weak reference like this:

```
WeakReference weakWidget = new WeakReference(widget);
```

and then elsewhere in the code you can use weakWidget.get() to get the actual Widget object. Of course the weak reference isn't strong enough to prevent garbage collection,

so you may find (if there are no strong references to the widget) that weakWidget.get() suddenly starts returning null.

To solve the "widget serial number" problem above, the easiest thing to do is use the built-in WeakHashMap class. WeakHashMap works exactly like HashMap, except that the keys (*not* the values!) are referred to using weak references. If a WeakHashMap key becomes garbage, its entry is removed automatically. This avoids the pitfalls I described and requires no changes other than the switch from HashMap to a WeakHashMap. If you're following the standard convention of referring to your maps via the Map interface, no other code needs to even be aware of the change.

Reference queues

Once a WeakReference starts returning null, the object it pointed to has become garbage and the WeakReference object is pretty much useless. This generally means that some sort of cleanup is required; WeakHashMap, for example, has to remove such defunct entries to avoid holding onto an ever-increasing number of dead WeakReferences.

The ReferenceQueue class makes it easy to keep track of dead references. If you pass a ReferenceQueue into a weak reference's constructor, the reference object will be automatically inserted into the reference queue when the object to which it pointed becomes garbage. You can then, at some regular interval, process the ReferenceQueue and perform whatever cleanup is needed for dead references.

Different degrees of weakness

Up to this point I've just been referring to "weak references", but there are actually four different degrees of reference strength: strong, soft, weak, and phantom, in order from strongest to weakest. We've already discussed strong and weak references, so let's take a look at the other two.

Soft references

A *soft reference* is exactly like a weak reference, except that it is less eager to throw away the object to which it refers. An object which is only weakly reachable (the strongest references to it are WeakReferences) will be discarded at the next garbage collection cycle, but an object which is softly reachable will generally stick around for a while.

SoftReferences aren't *required* to behave any differently than WeakReferences, but in practice softly reachable objects are generally retained as long as memory is in plentiful supply. This makes them an excellent foundation for a cache, such as the image cache described above, since you can let the garbage collector worry about both how

reachable the objects are (a strongly reachable object will *never* be removed from the cache) and how badly it needs the memory they are consuming.

Phantom references

A *phantom reference* is quite different than either SoftReference or WeakReference. Its grip on its object is so tenuous that you can't even retrieve the object -- its get() method always returns null. The only use for such a reference is keeping track of when it gets enqueued into a ReferenceQueue, as at that point you know the object to which it pointed is dead. How is that different from WeakReference, though?

The difference is in exactly when the enqueuing happens. WeakReferences are enqueued as soon as the object to which they point becomes weakly reachable. This is before finalization or garbage collection has actually happened; in theory the object could even be "resurrected" by an unorthodox finalize() method, but the WeakReference would remain dead. PhantomReferences are enqueued only when the object is physically removed from memory, and the get() method always returns null specifically to prevent you from being able to "resurrect" an almost-dead object.

What good are PhantomReferences? I'm only aware of two serious cases for them: first, they allow you to determine exactly when an object was removed from memory. They are in fact the *only* way to determine that. This isn't generally that useful, but might come in handy in certain very specific circumstances like manipulating large images: if you know for sure that an image should be garbage collected, you can wait until it actually is before attempting to load the next image, and therefore make the dreaded OutOfMemoryError less likely.

Second, PhantomReferences avoid a fundamental problem with finalization: finalize() methods can "resurrect" objects by creating new strong references to them. So what, you say? Well, the problem is that an object which overrides finalize() must now be determined to be garbage in at least two separate garbage collection cycles in order to be collected. When the first cycle determines that it is garbage, it becomes eligible for finalization. Because of the (slim, but unfortunately real) possibility that the object was "resurrected" during finalization, the garbage collector has to run again before the object can actually be removed. And because finalization might not have happened in a timely fashion, an arbitrary number of garbage collection cycles might have happened while the object was waiting for finalization. This can mean serious delays in actually cleaning up garbage objects, and is why you can get OutOfMemoryErrors even when most of the heap is garbage.

With PhantomReference, this situation is impossible -- when a PhantomReference is enqueued, there is absolutely no way to get a pointer to the now-dead object (which is good, because it isn't in memory any longer). Because PhantomReference cannot be used to resurrect an object, the object can be instantly cleaned up during the first

garbage collection cycle in which it is found to be phantomly reachable. You can then dispose whatever resources you need to at your convenience.

Arguably, the finalize() method should never have been provided in the first place. PhantomReferences are definitely safer and more efficient to use, and eliminating finalize() would have made parts of the VM considerably simpler. But, they're also more work to implement, so I confess to still using finalize() most of the time. The good news is that at least you have a choice.

Conclusion

I'm sure some of you are grumbling by now, as I'm talking about an API which is nearly a decade old and haven't said anything which hasn't been said before. While that's certainly true, in my experience many Java programmers really don't know very much (if anything) about weak references, and I felt that a refresher course was needed. Hopefully you at least learned a *little* something from this review.