Java.net
The Source for Java Technology Collaboration

☐ Forums ☐ Blogs ☐ Projects ☐ People   Search All

Home   **Projects**   **Forums**   **People**   **Java User Groups**   **JCP**

Get Involved

About Java.net

Adopt a JSR

Create a Project

Link an Offsite Project

Get Informed

Articles

Blogs

Events

Java Magazine

Oracle University

Ethan Nicholas

Leaking Evil | Main | JAXX user interface language:
marching towards 1.0

Understanding Weak References

Posted by **enicholas** on May 4, 2006 at 5:06 PM PDT

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:

Recent Entries

Introducing Java CSS

Java Secrets Revealed #1

Java Kernel Unmasked

Archives

```
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<Widget> weakWidget = new WeakReference<Widget>(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.

Related Topics >> Java Desktop

Blog Links >>   Login or **register** to post comments   **Printer-friendly version   enicholas's blog   325398 reads**

Comments

Thanks a lot for the detailed info on weak references. I had ...

by santhoshgutta - 2014-08-04 16:15

Thanks a lot for the detailed info on weak references. I had a bit overview on that, but your post actually gave a well defined knowledge. Thank You.

Login or register to post comments

Really good explanation. Thanks :)

by gauravjain.dtu - 2014-04-08 01:49

Really good explanation. Thanks :)

Login or register to post comments

Cheers Ethan, really useful, and yep I've been developing ...

by stream2stream - 2014-02-18 23:33

Cheers Ethan, really useful, and yep I've been developing Java apps since 1995 but kinda brushed over this topic quite a few times, a refresher was good and your explanation excellent...

Login or register to post comments

Good explanation! This was helpfull.

by napstr - 2013-09-22 23:40

Good explanation! This was helpfull.

Login or register to post comments

Very good, thanks!

by marlonpatrick10 - 2013-08-22 18:34

Very good, thanks!

Login or register to post comments

Thanks! This really cleared up a lot of questions I had ...

by bgroenks96 - 2013-01-21 17:31

Thanks! This really cleared up a lot of questions I had about using the reference package!

Login or register to post comments

Thanks a lot for such a nice contribution! It's a must read ...

by sumved.shami - 2013-01-04 04:06

Thanks a lot for such a nice contribution! It's a must read for all Java lovers. :-)

Login or register to post comments

Excellent written awork and very helpful. Thanks!

by wanziliuyy41 - 2012-09-12 02:18

Excellent written awork and very helpful.
Thanks!

Login or register to post comments

Thanks for the clear and useful explanation!

by sierk - 2011-11-17 02:11

Thanks for the clear and useful explanation!

Login or register to post comments

Thanks for the article, but I can't agree with one ...

by kosenkov - 2011-11-14 07:47

Thanks for the article, but I can't agree with one statement:

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.*

As documentation for the class tells us:

OutOfMemoryError is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and **no more memory could be made available** by the garbage collector.

What means, in my experience and understanding, that GC has already made every possible effort to de-allocate dead, softly-reachable and phantom-reachacble objects.
So, there's absolutely no need to wait for GC signal from Phantom Reference to try to allocate new big object.

Login or register to post comments

Understanding Weak

by deepnighttwo - 2010-11-24 02:36

The part about WeakRef vs PhantomRef is really clear and helpful! I learned the three reference classes in book Inside Java Virtual Machine. Finally know what is weak and what is phantom.

Examples here are very expressive too.
Many Thanks!

Login or register to post comments

Understanding Weak

by abhitabh - 2010-11-09 23:41

Very informative and easily understandable......
I had a basic idea but was not sure about the usage of the concept......

Login or register to post comments

Thanks for the article! It

by frodesto - 2010-10-18 04:07

Thanks for the article! It helped me understand why java.util.logging.Logger.getLogger(name) did not always return the same Logger instance. It was a great mystery until I discovered that the LogManager keeps its loggers in a Map of WeakReferences.

Login or register to post comments

Example source code

by jaydipbh1 - 2010-04-28 11:46

import java.lang.management.ManagementFactory; import java.lang.ref.WeakReference; import javax.management.MBeanServer; import com.sun.management.HotSpotDiagnosticMXBean; // had to do this to get this pacjage working // project.properties.BuildPath.Library , remove JRE Library // then Add Library-> JRE library import java.util.Map; import java.util.WeakHashMap; import java.util.HashMap; public class Heaptest { private static Map mapp; private static HotSpotDiagnosticMXBean hotspotMBean; static { mapp = new WeakHashMap(); } public static void main(String[] args) { byte [] b = new byte[1024 * 1024 *10]; String xx = new String("mykey"); mapp.put(xx, b); /* * WeakHashMap will now contain entries of the type * Entry. * Entry extends WeakReference implements Map.Entry - per source code * for WeakHashMap. * class Entry { * private K key; * private V value; * Entry() { * super(K , referenceQ); // WeakReference c'tor * key = k; * value = v; * ..... * } * } * So essentially WeakHashMap will contain list of WeakReference type objects */ /* * Observe entries in WeakHashMap before garbage collection . * Notice that object in WeakHashMap can be type casted successfully * to a weak reference. */ System.out.println("before garbage collection "); for(Map.Entry e : mapp.entrySet()) { WeakReference< String > we = (WeakReference< String>) e; if (we != null) { System.out.println(we.get()); } } System.out.println("Map size ="+ mapp.size()); dumpHeap("Y:/java.hprof1"); /* * Simulating garbage collection on xx. * When xx is garage collected, Weak Reference to xx is put in the ReferenceQ. * So, this will cause entry Entry< String, byte[] >:WeakReference("mykey") to be moved to * the referenceQ attached to WeakReference */ xx = null; System.gc(); /* contrary to conventional wisdom, this ACTUALY calls garbage collection */ try { Thread.sleep(10000); } catch(Exception e) {} /* * Most functions in WeakHashMap calls the routine for * cleaning up entries in referenceQ. * So mapp.entrySet will find entries Entryin the reference Q * and , it de-allocate the entry from the WeakHashMap, and byte[] will * garbage collected. */ System.out.println("After garbage collection "); for(Map.Entry e : mapp.entrySet()) { WeakReference< String > we = (WeakReference< String>) e; if (we != null) { System.out.println(we.get()); } } System.out.println("Map size ="+ mapp.size()); dumpHeap("Y:/java.hprof"); } static void dumpHeap(String filename) { if (hotspotMBean == null) { synchronized (Heaptest.class) { if (hotspotMBean == null) { try { MBeanServer server = ManagementFactory.getPlatformMBeanServer(); hotspotMBean = ManagementFactory.newPlatformMXBeanProxy(server, //"com.sun.management.HotSpotDiagnosticMXBean" "com.sun.management:type=HotSpotDiagnostic", HotSpotDiagnosticMXBean.class); } catch (RuntimeException re) { throw re; } catch (Exception exp) { throw new RuntimeException(exp); } } } } try {

```
hotspotMBean.dumpHeap(filename, true ); } catch (RuntimeException re) {
throw re; } catch (Exception exp) { throw new RuntimeException(exp); }
System.out.println("Heap dump done"); } }
```

Login or register to post comments

Great explanation. Thanks.

by aconn7 - 2010-04-28 08:12

This made immediate sense to me coming from a C / C++ background.
Reminiscent of some classes I've used from the well known C++ BOOST
library. It's nice to see this kind of memory management available in Java as
well.

Login or register to post comments

Phantom references

by nikolayo - 2010-03-10 15:18

I wonder what the following quote from JDK 1.6 javadoc means: "Unlike soft and
weak references, phantom references are not automatically cleared by the
garbage collector as they are enqueued. An object that is reachable via
phantom references will remain so until all such references are cleared or
themselves become unreachable." It seems to imply that object will NOT be
reclaimed (since it is still reachable) before reference is retrieved from queue
and explicitly cleared or made unreachable. At the same time actual behavior of
phantom references in Sun JDK 6 is as described in the article. Is this wrong
reading of documentation or wrong documentation or (god forbid :) non
compliant implementation? BTW object referred by phantom reference can be
"resurrected" in finalize() but in such case it does not make it to the queue. I
also wander about guarantees of phantom references being queued within
certain time or at all. All that javadoc says is: "If the garbage collector
determines at a certain point in time that the referent of a phantom reference is
phantom reachable, then at that time or at some later time it will enqueue the
reference. " Here "at some later time" sounds way too vague to be relied on for
guaranteed deinitalization of related objects. This does not sound much better
than finalize().

Login or register to post comments

Re: Phantom references

by mzwaart - 2010-03-29 06:21

The behavior I observe on JDK 1.6.0_18 is as stated in the javadoc: the
referenced object is not reclaimed until after the phantom reference has
been collected. Even after the phantom reference has been enqueued,
and taken from the queue, as long as the reference is alive, the referenced
object won't be collected.

Login or register to post comments

thanks

by ustad1024 - 2010-01-27 18:14

This article was quite helpful - thanks. It would be even better if you could
provide a few coding examples of Weak Reference use.

Login or register to post comments

Great Post

by kramer2718 - 2010-01-26 12:37

Well explained, great topic, and (I am embarrassed to say) one I was not
previously familiar with.

Login or register to post comments

Thanks

by soorejkumar - 2010-01-18 09:28

Nice informative topic. Thanks for sharing.

Login or register to post comments

thanks for sharing

by kangjunwei - 2012-06-20 22:58

thanks for sharing

Login or register to post comments

Feedback | **FAQ** | **Terms of Use** | **Privacy** | **Trademarks**

ORACLE Project K≡nai cognisync

Powered by Oracle, Project Kenai and Cognisync