Reference types and Reachability – Strong, Soft, Weak, Phantom, Unreachable

<http://www.onkarjoshi.com/blog/186/garbage-collection-and-reachability-strong-soft-weak-phantom-unreachable/>

The Java Virtual Machine has 5 different levels of reachability for an object.

**Strongly reachable**

* An object is *strongly reachable* if it can be reached by some thread without traversing any reference objects. A newly-created object is strongly reachable by the thread that created it.

This is the reference you know about and use all the time. Any object that is reachable by a strong reference (from a live thread) will not be collected by the GC.

**Softly reachable**

* An object is *softly reachable* if it is not strongly reachable but can be reached by traversing a soft reference.

The garbage collector will collect softly reachable objects only if it actually requires more memory. Before it does a GC it is guaranteed to collect softly reachable objects.

So it is possible to write code on the lines of – “*Hey, I do want this data to be kept in memory. But in case the JVM starts running out of memory, go ahead and collect it and null out the referent. I’ll handle it in my code.*” The exact algorithm that is used to determine whether a SoftReference should be cleared out is JVM vendor dependent. It tends to be a function of how often and how long ago the reference was last used to reach the referent.

The semantics of a SoftReference make it possible to make a memory sensitive cache. The gotcha is that the exact behaviour will be JVM dependent. And as with anything else related to garbage collector behaviour, guarantees are few and are vendor dependent. For higher reliability (and many more features) most people tend to use something like Ehcache rather than roll out their own caching logic using SoftReference unless they have a very simple use case. Used in the right situation, they certainly can result in some elegant, simple code.

**Weakly reachable**

* An object is *weakly reachable* if it is neither strongly nor softly reachable but can be reached by traversing a weak reference. When the weak references to a weakly-reachable object are cleared, the object becomes eligible for finalization.

In a nut-shell, a weak reference will not prevent its referent from being collected by the garbage collector. The most common use case of a WeakReference is via a WeakHashMap. It is an easy way to associate data with an object bound to the lifetime of the object. The bound data will stay in the map only while the key is not weakly reachable due to the key being strongly reachable because of reachability via a reference elsewhere in the application. Once all other references in the application are gone, the key becomes weakly reachable and is gone from the map without the user needing to explicitly remove the entry. An elegant way to prevent memory leaks in some cases!

**Phantom reachable**

* An object is *phantom reachable* if it is neither strongly, softly, nor weakly reachable, it has been finalized, and some phantom reference refers to it.

The referent of a PhantomReference can never be retrieved. The get() method on this reference always returns null! So what is the use?

All Reference types allow specifying a ReferenceQueue in the constructor. The semantics of how and when a PhantomReference is enqueued makes it possible to do finalization in a better, less fragile way. More on that in a future blog post perhaps.

**Unreachable**

* An object is *unreachable*, and therefore eligible for reclamation, when it is not reachable in any of the above ways.
* Finalization and Phantom References

<https://dzone.com/articles/finalization-and-phantom>

Memory management is done automatically in Java. The programmer doesn't need to worry about reference objects that have been released. One downside to this approach is that the programmer cannot know when a particular object will be collected. Moreover, the programmer has no control over memory management. However, the [java.lang.ref](http://java.sun.com/javase/6/docs/api/java/lang/ref/package-summary.html) package defines classes that provide a limited degree of interaction with the garbage collector. The concrete classes [SoftReference](http://java.sun.com/javase/6/docs/api/java/lang/ref/SoftReference.html), [WeakReference](http://java.sun.com/javase/6/docs/api/java/lang/ref/WeakReference.html)and [PhantomReference](http://java.sun.com/javase/6/docs/api/java/lang/ref/PhantomReference.html) are subclasses of Reference that interact with the garbage collector in different ways. In this article we will discuss the functionality and behavior of the PhantomReferenceclasses and see how it can be used.

## **Problem with Finalization**

To perform some postmortem cleanup on objects that garbage collector consider as unreachable, one can use finalization. This feature can be utilized to reclaim native resources associated with an object. However, finalizershave many problems associated.

Firstly, we can’t foresee the call of finalize(). Since the Garbage Collection is unpredictable, the calling of finalize() cannot be predicted. There is no guarantee that the object will be garbage collected. The object might never become eligible for GC because it could be reachable through the entire lifetime of the JVM. It is also possible that no garbage collection actually runs from the time the object became eligible and before JVM stops.

Secondly, Finalizationcan slowdown an application. Managing objects with a finalize()method takes more resources from the JVM than normal objects.

As per doc,

*You should also use finalization only when it is absolutely necessary. Finalization is a nondeterministic -- and sometimes unpredictable -- process. The less you rely on it, the smaller the impact it will have on the JVM and your application*

 In Effective Java, 2nd ed., Joshua Bloch says,

*there is a severe performance penalty for using finalizers... So what should you do instead of writing a finalizer for a class whose objects encapsulate resources that require termination, such as files or threads? Just provide an explicit termination method, and require clients of the class to invoke this method on each instance when it is no longer needed.*

In short, Finalize() isn't used often, and also there is no much reason to use it. If we have a class with methods like close() or cleanup() and that should be called once user done with the object then placing these methods call in finalize() can be used as a safety measure, but not necessary.

## **Phantom Reference**

### **phantom reachable, phantomly reachable**

An object is phantom reachable if it is neither strongly nor softly nor weakly reachable and has been finalized and there is a path from the roots to it that contains at least one phantom reference.

The [PhantomReference](http://java.sun.com/javase/6/docs/api/java/lang/ref/PhantomReference.html) constructor accepts two arguments:

referent - the object the new phantom reference will refer to  
q - the reference is registered with the given queue.

The argument q represents the instance of the [ReferenceQueue](http://java.sun.com/javase/6/docs/api/java/lang/ref/ReferenceQueue.html) class. If the garbage collector determines that the referent of a phantom reference is phantom reachable, then the PhantomReference will be added to this ReferenceQueue. You can then retrieve the PhantomReference by using the remove() methods of the ReferenceQueue class.

Consider the following example,

ReferenceQueue q = new ReferenceQueue();PhantomReference pr = new PhantomReference(object, referenceQueue);// Later on another pointReference r = q.remove();// Now, clear up any thing you want

## **PhantomReference, when to use?**

Phantom Reference can be used in situations, where sometime using finalize() is not  sensible thing to do.This reference type differs from the other types defined in [java.lang.ref](http://java.sun.com/javase/6/docs/api/java/lang/ref/package-summary.html) Package because it isn't meant to be used to access the object, but as a signal that the object has already been finalized, and the garbage collector is ready to reclaim its memory.

As per API doc,

*Phantom reference objects, which are enqueued after the collector determines that their referents may otherwise be reclaimed. Phantom references are most often used for scheduling pre-mortem cleanup actions in a more flexible way than is possible with the Java finalization mechanism.*

People usually attempt to use finalize() method to perform postmortem cleanup on objects which usually not advisable. As mentioned earlier, Finalizers have an impact on the performance of the garbage collector since Objects with finalizers are slow to garbage collect.

Phantom references are safe way to know an object has been removed from memory. For instance, consider an application that deals with large images. Suppose that we want to load a big image in to memory when large image is already in memory which is ready for garbage collected. In such case, we want to wait until the old image is collected before loading a new one. Here, the phantom reference is flexible and safely option to choose. The reference of the old image will be enqueued in the ReferenceQueue once the old image object is finalized. After receiving that reference, we can load the new image in to memory.

Similarly we can use Phantom References to implement a Connection Pool. We can easily gain control over the number of open connections, and can block until one becomes available.

## **Reference Objects and Garbage Collection**

Soft Reference can be garbage collected after there are no strong references to the referent. However, it typically retained until memory is low. All softly reachable objects will be reclaimed before an OutOfMemoryException is thrown. Therefore, it can be used to implement caches of objects that can be recreated if needed.

Weak Reference can be garbage collected when there are no strong or soft references to the referent. However, unlike Soft Reference, they are garbage collected on a gc even when memory is abundant. They often can be used for “canonical mappings” where each object has a unique identifier (one-to-one), and in collections of “listeners”

On the other hand, Phantom Reference, can be garbage collected once there are no strong, soft or weak references to the referent. When object is phantomly reachable, it means the object is already finalized but not yet reclaimed, so the GC enqueues it in a [ReferenceQueue](http://java.sun.com/javase/6/docs/api/java/lang/ref/ReferenceQueue.html) for post-finalization processing.

As per Java Doc,

*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.*

A PhantomReference is not automatically cleared when it is enqueued, so when we remove a PhantomReference from a [ReferenceQueue](http://java.sun.com/javase/6/docs/api/java/lang/ref/ReferenceQueue.html), we must call its clear() method or allow the PhantomReference object itself to be garbage-collected.

## **Summary**

In short, we should avoid finalize() as much as possible. There is no guarantee if the finalize()method will be called promptly following garbage collection, or even it will be called. If the finalize method runs for a long time, it can delay execution of finalize methods of other objects. Instead of relying on finalize(), we can use reference types define in [java.lang.ref](http://java.sun.com/javase/6/docs/api/java/lang/ref/package-summary.html) package.

Beside [java.lang.ref](http://java.sun.com/javase/6/docs/api/java/lang/ref/package-summary.html) package, Google collection library also provide some alternatives. For example, [FinalizablePhantomReference](http://google-collections.googlecode.com/svn/trunk/javadoc/com/google/common/base/FinalizablePhantomReference.html) extends [java.lang.ref.PhantomReference](http://java.sun.com/javase/6/docs/api/java/lang/ref/PhantomReference.html), deals with processing the [ReferenceQueue](http://java.sun.com/javase/6/docs/api/java/lang/ref/ReferenceQueue.html) and call back a convenient method finalizeReferent(). So if we want to do some cleanup operation when an object is claimed by the garbage collector (GC) then we just need to override the finalizeReferent() method.

# Understanding Weak References

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 eachWidget'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<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.

# [Strong, Soft, Weak and Phantom References (Java)](http://neverfear.org/blog/view/150/Strong_Soft_Weak_and_Phantom_References_Java)

<http://neverfear.org/blog/view/150/Strong_Soft_Weak_and_Phantom_References_Java>

There are four distinct forms of references in the JVM, and indeed many of these apply to other garbage collected languages.

* Strong references
* Soft references
* Weak references
* Phantom references

It's important to know the differences, what affect they have on the collector and when you should be using them.

# Strong references

Strong references never get collected

**package** org.neverfear.leaks;

*/\**

*\* URL: http://neverfear.org/blog/view/150/Java\_References*

*\* Author: doug@neverfear.org*

*\*/*

**public** **class** ClassStrong {

**public** **static** **class** Referred {

**protected** **void** finalize() {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Good bye cruel world");

}

}

**public** **static** **void** collect() **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Suggesting collection");

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).gc();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Sleeping");

[Thread](http://www.google.com/search?hl=en&q=allinurl%3Athread+java.sun.com&btnI=I%27m%20Feeling%20Lucky).sleep(5000);

}

**public** **static** **void** main([String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) args[]) **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Creating strong references");

*// This is now a strong reference.*

*// The object will only be collected if all references to it disappear.*

Referred strong = **new** Referred();

*// Attempt to claim a suggested reference.*

ClassStrong.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Removing reference");

*// The object may now be collected.*

strong = **null**;

ClassStrong.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Done");

}

}

# Soft references

[Soft references](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ref/SoftReference.html) only get collected if the JVM absolutely needs the memory. This makes them excellent for implementing object cache's.

**package** org.neverfear.leaks;

**import** java.lang.ref.SoftReference;

**import** java.util.ArrayList;

**import** java.util.List;

*/\**

*\* A sample for Detecting and locating memory leaks in Java*

*\* URL: http://neverfear.org/blog/view/150/Java\_References*

*\* Author: doug@neverfear.org*

*\*/*

**public** **class** ClassSoft {

**public** **static** **class** Referred {

**protected** **void** finalize() {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Good bye cruel world");

}

}

**public** **static** **void** collect() **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Suggesting collection");

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).gc();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Sleeping");

[Thread](http://www.google.com/search?hl=en&q=allinurl%3Athread+java.sun.com&btnI=I%27m%20Feeling%20Lucky).sleep(5000);

}

**public** **static** **void** main([String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) args[]) **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Creating soft references");

*// This is now a soft reference.*

*// The object will be collected only if no strong references exist and the JVM really needs the memory.*

Referred strong = **new** Referred();

SoftReference<Referred> soft = **new** SoftReference<Referred>(strong);

*// Attempt to claim a suggested reference.*

ClassSoft.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Removing reference");

*// The object may but highly likely wont be collected.*

strong = **null**;

ClassSoft.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Consuming heap");

**try**

{

*// Create lots of objects on the heap*

List<ClassSoft> heap = **new** ArrayList<ClassSoft>(100000);

**while**(**true**) {

heap.add(**new** ClassSoft());

}

}

**catch** ([OutOfMemoryError](http://www.google.com/search?hl=en&q=allinurl%3Aoutofmemoryerror+java.sun.com&btnI=I%27m%20Feeling%20Lucky) e) {

*// The soft object should have been collected before this*

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Out of memory error raised");

}

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Done");

}

}

# Weak references

[Weak references](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ref/WeakReference.html) only get collected if no other object references it except the weak references. This makes them perfect for keeping meta data about a particular object for the life time of the object.

**package** org.neverfear.leaks;

**import** java.lang.ref.WeakReference;

**import** java.util.ArrayList;

**import** java.util.List;

*/\**

*\* A sample for Detecting and locating memory leaks in Java*

*\* URL: http://neverfear.org/blog/view/150/Java\_References*

*\* Author: doug@neverfear.org*

*\*/*

**public** **class** ClassWeak {

**public** **static** **class** Referred {

**protected** **void** finalize() {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Good bye cruel world");

}

}

**public** **static** **void** collect() **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Suggesting collection");

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).gc();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Sleeping");

[Thread](http://www.google.com/search?hl=en&q=allinurl%3Athread+java.sun.com&btnI=I%27m%20Feeling%20Lucky).sleep(5000);

}

**public** **static** **void** main([String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) args[]) **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Creating weak references");

*// This is now a weak reference.*

*// The object will be collected only if no strong references.*

Referred strong = **new** Referred();

WeakReference<Referred> weak = **new** WeakReference<Referred>(strong);

*// Attempt to claim a suggested reference.*

ClassWeak.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Removing reference");

*// The object may be collected.*

strong = **null**;

ClassWeak.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Done");

}

}

# Phantom references

[Phantom references](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/ref/PhantomReference.html) are objects that can be collected whenever the collector likes. The object reference is appended to a ReferenceQueue and you can use this to clean up after a collection. This is an alternative to the finalize() method and is slightly safer because the finalize() method may ressurect the object by creating new strong references. The PhantomReference however cleans up the object and enqueues the reference object to a ReferenceQueue that a class can use for clean up.

**package** org.neverfear.leaks;

**import** java.lang.ref.PhantomReference;

**import** java.lang.ref.Reference;

**import** java.lang.ref.ReferenceQueue;

**import** java.util.HashMap;

**import** java.util.Map;

*/\**

*\* A sample for Detecting and locating memory leaks in Java*

*\* URL: http://neverfear.org/blog/view/150/Java\_References*

*\* Author: doug@neverfear.org*

*\*/*

**public** **class** ClassPhantom {

**public** **static** **class** Referred {

*// Note that if there is a finalize() method PhantomReference's don't get appended to a ReferenceQueue*

}

**public** **static** **void** collect() **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Suggesting collection");

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).gc();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Sleeping");

[Thread](http://www.google.com/search?hl=en&q=allinurl%3Athread+java.sun.com&btnI=I%27m%20Feeling%20Lucky).sleep(5000);

}

**public** **static** **void** main([String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) args[]) **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Creating phantom references");

*// The reference itself will be appended to the dead queue for clean up.*

[ReferenceQueue](http://www.google.com/search?hl=en&q=allinurl%3Areferencequeue+java.sun.com&btnI=I%27m%20Feeling%20Lucky) dead = **new** [ReferenceQueue](http://www.google.com/search?hl=en&q=allinurl%3Areferencequeue+java.sun.com&btnI=I%27m%20Feeling%20Lucky)();

*// This map is just a sample we might use to locate resources we need to clean up.*

Map<Reference,String> cleanUpMap = **new** HashMap<Reference,String>();

*// This is now a phantom reference.*

*// The object will be collected only if no strong references.*

Referred strong = **new** Referred();

PhantomReference<Referred> phantom = **new** [PhantomReference](http://www.google.com/search?hl=en&q=allinurl%3Aphantomreference+java.sun.com&btnI=I%27m%20Feeling%20Lucky)(strong, dead);

cleanUpMap.put(phantom, "You need to clean up some resources, such as me!");

strong = **null**;

*// The object may now be collected*

ClassPhantom.collect();

*// Check for*

[Reference](http://www.google.com/search?hl=en&q=allinurl%3Areference+java.sun.com&btnI=I%27m%20Feeling%20Lucky) reference = dead.poll();

**if** (reference != **null**) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println(cleanUpMap.remove(reference));

}

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Done");

}

}

# ReferenceQueue

You saw me use the reference queue class in the previous example. A [ReferenceQueue](http://download.oracle.com/javase/1.5.0/docs/api/java/lang/ref/ReferenceQueue.html) instance can be supplied as an argument to SoftReference, WeakReference or PhantomReference. When an object is collected the reference instance itself will be enqueued to the supplied ReferenceQueue. This allows you to perform clean up operations on the object. This is useful if you are implementing any container classes that you want to contain a Soft, Weak or Phantom reference and some associated data because you can get notified via the ReferenceQueue which Reference was just collected.

# WeakHashMap class

There is also a convience [WeakHashMap](http://download.oracle.com/javase/1.4.2/docs/api/java/util/WeakHashMap.html) that wraps all keys by a weak reference. Allowing you to easily store meta data against an object and have the map entry including the meta data removed and collected when the original object itself is unreachable.

**package** org.neverfear.leaks;

**import** java.util.Map;

**import** java.util.WeakHashMap;

*/\**

*\* A sample for Detecting and locating memory leaks in Java*

*\* URL: http://neverfear.org/blog/view/150/Java\_References*

*\* Author: doug@neverfear.org*

*\*/*

**public** **class** ClassWeakHashMap {

**public** **static** **class** Referred {

**protected** **void** finalize() {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Good bye cruel world");

}

}

**public** **static** **void** collect() **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Suggesting collection");

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).gc();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Sleeping");

[Thread](http://www.google.com/search?hl=en&q=allinurl%3Athread+java.sun.com&btnI=I%27m%20Feeling%20Lucky).sleep(5000);

}

**public** **static** **void** main([String](http://www.google.com/search?hl=en&q=allinurl%3Astring+java.sun.com&btnI=I%27m%20Feeling%20Lucky) args[]) **throws** [InterruptedException](http://www.google.com/search?hl=en&q=allinurl%3Ainterruptedexception+java.sun.com&btnI=I%27m%20Feeling%20Lucky) {

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Creating weak references");

*// This is now a weak reference.*

*// The object will be collected only if no strong references.*

Referred strong = **new** Referred();

Map<Referred,String> metadata = **new** WeakHashMap<Referred,String>();

metadata.put(strong, "WeakHashMap's make my world go around");

*// Attempt to claim a suggested reference.*

ClassWeakHashMap.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Still has metadata entry? " + (metadata.size() == 1));

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Removing reference");

*// The object may be collected.*

strong = **null**;

ClassWeakHashMap.collect();

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Still has metadata entry? " + (metadata.size() == 1));

[System](http://www.google.com/search?hl=en&q=allinurl%3Asystem+java.sun.com&btnI=I%27m%20Feeling%20Lucky).out.println("Done");

}

}

# Know the JVM Series -3- When Weaker is Better: Understanding Soft, Weak and Phantom References

# <http://blog.yohanliyanage.com/2010/10/ktjs-3-soft-weak-phantom-references/>

How many times have we created various object instances, and assign those to reference variables? We all know very well that Java has automatic garbage collection; so we just play around the reference variables, and once those variables are assigned null or falls out of scope, JVM takes care of it. No need to worry about ‘free’ as in C / C++. It’s a headache-less approach, which minimizes the risk of introducing memory leaks to our programs, and it works out great day in day out in billions of Java applications running out there 24×7. Kudos to [John McCarthy](http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) for inventing GC for Lisp, and to  all the folks who implemented the concept in Java.

But there are times, where we a little bit of more control over the process of garbage collection. I’m not talking about the dark art of tuning the garbage collector (which I might cover in a later article). This is about programmatic situations where we expect some object instances to be eligible for garbage collection, to release some unwanted memory that might get accumulated over the time. Well, the classic solution of explicitly assigning null could help us out; given that particular object is referred only through that particular ref variable. What if assigning null doesn’t work out for the problem at hand?

Consider a scenario where you are required to implement an object cache. You have some objects which are pretty expensive to build. We would like to keep the objects in the cache as long as we can (just in case if a component of the application needs to use it), but we want un-used objects from our cache to be released when we need memory for other important operations of the application. If we are to implement this using standard (strong) references, this would be quite difficult to implement. The moment we add the object to a collection, we maintain a (strong) reference to the instance, making it not-eligible for garbage collection. If the cache continues to grow, we could run out of memory, making it a memory leak point for the application. Obvious solution would be to limit the size of the cache, and to drop off older objects from the cache, making those instances eligible for GC. Well, that is the mechanism used by many cache implementations out there, and it works out fine. But the drawback is that our cache is limited by size and therefore, even though we have more free memory, we cannot make use of it. Also, since the cache will always have some references in it, it will continue to allocate a significant block of memory for the lifetime of the application (ex. if the cache is fixed to 1024 references, memory consumed by those 1024 references will not be released). Yes, there are ways to address each of these problems (ex. dynamically grow / shrink cache), but that requires some fair deal of coding to get it done. This was a practical issue that I came across in one of my projects in the past.

If only there was a better (and easier) way to get this done…

The solution has been part of the JDK for a long time, from the days of Java 2. Meet java.lang.ref package, where Soft, Weak and Phantom references can be used to resolve such problems. The references that we create using the assignment operator are known as strong references, because the instance is strongly referred by the application, making it ineligible for garbage collection.

Object obj = new Object(); // Strong Ref

Soft, Weak and Phantom references are the weaker counterparts of referencing, where the garbage collection algorithm is allowed to mark an instance to be garbage collected, even though such a reference exists. What this means is that, even though you hold a weak reference to a particular instance, the JVM can sweep it out of the memory if it needs to. This works out great for the problem we discussed before, since instances in our cache will be automatically released if the JVM thinks it needs more memory for other parts.

A weak reference can be created to an instance as follows (all the reference types are available in java.lang.ref package).

WeakReference<Object> weakRef = new WeakReference<Object> (obj);

When we create a weak reference like this, the instance referred by the ref variable obj will be eligible for garbage collection if no strong reference exists. But, if some part of the application needs to use this particular object instance, we can get a strong reference back to it as follows (given that it was not gc’d during the time in-between).

Object strongRef = weakRef.get();

If the reference has been already garbage collected, calling the get method will return null.

Below is a fully working example of using weak references to demonstrate what we have covered so far.

public class TestRef {

public static void main(String[] args) {

// Initial Strong Ref

Object obj = new Object();

System.out.println("Instance : " + obj);

// Create a Weak Ref on obj

WeakReference<Object> weakRef

= new WeakReference<Object>(obj);

// Make obj eligible for GC !

obj = null;

// Get a strong reference again. Now its not eligible for GC

Object strongRef = weakRef.get();

System.out.println("Instance : " + strongRef);

// Make the instance eligible for GC again

strongRef = null;

// Keep your fingers crossed

System.gc();

// should be null if GC collected

System.out.println("Instance : " + weakRef.get());

}

}

And the output of the program would be:

Instance : java.lang.Object@a90653

Instance : java.lang.Object@a90653

Instance : null

Now that we have covered why we need weak references, and a practical example of using weak references, let’s cover some theory behind the five degrees of reachability. The following is based on the JDK [API Docs](http://download.oracle.com/javase/1.3/docs/api/java/lang/ref/package-summary.htm).

1. Strongly Reachable – If we have a strong reference to a particular instance, then it is said to be strongly reachable. Hence, it is not eligible for garbage collection.
2. Softly Reachable – If we do not have a strong reference to an instance, but we can access the object through a SoftReference (more on that later) to it, then the instance is said to be softly reachable.
3. Weakly Reachable – If we have neither a strong reference nor a soft reference, but the object can be accessed through a WeakReference, then the instance is said to be weakly reachable.
4. Phantomly Reachable – If we don’t have any of the strong, soft or weak references to a particular instance (which has not been finalized), but, if we do have a PhantomReference (explained in a while) to the instance, then the instance is said to be phantomly reachable.
5. Unreachable – If we do not have any of the above references to an instance, then it is unreachable from the program.

At this point, you must be wondering about the difference, and the need, to have three different levels of weaker referencing mechanisms. In the order of strength, the references can be arranged as,

**Strong References > Soft References > Weak References > Phantom References**

Each of these referencing mechanisms serves a specific purpose. We will look at each of these references, and some related constructs in the API next.

**1. Soft References**

According to the Java API Specification, the JVM implementations are encouraged not to clear out a soft reference if the JVM has enough memory. That is, if free heap space is available, chances are that a soft reference will not be freed during a garbage collection cycle (so it survives from GC).  However, before throwing an OutOfMemoryError, JVM will attempt to reclaim memory by releasing instances that are softly reachable.  This makes Soft References ideal for implementing memory sensitive caches (as in our example problem).

Consider the following example.

public class TestSoftRef {

public static void main(String[] args) {

// Initial Strong Ref

Object obj = new Object();

System.out.println("Instance : " + obj);

// Make a Soft Reference on obj

SoftReference<Object> softReference =

new SoftReference<Object>(obj);

// Make obj eligible for GC !

obj = null;

System.gc(); // Run GC

// should be null if GC collected

System.out.println("Instance : " + softReference.get());

}

}

And the output will be…

Instance : java.lang.Object@de6ced

Instance : java.lang.Object@de6ced

As we expected, since JVM had enough memory, it did not reclaim the memory consumed by our softly referenced instance.

**2. Weak References**

Unlike Soft References, Weak References can be reclaimed by the JVM during a GC cycle, even though there’s enough free memory available.  Our first example on weaker reference models was based on Weak References. As long as GC does not occur, we can retrieve a strong reference out of a weak reference by calling the ref.get() method.

**3. Phantom References**

Phantom references are the weakest form of referencing. Instances that are referred via a phantom reference cannot be accessed directly using a get() method (it always returns null), as in case of Soft / Weak references.

Instead, we need to rely on Reference Queues to make use of Phantom References. We will take a look at reference queues in a while. One use case of Phantom references is to keep track of active references with in an application, and to know when those instances will be garbage collected. If we use strong references, then the instance will not be eligible for GC due to the strong reference we maintain. Instead, we could rely on a phantom reference with the support of a reference queue to handle the situation. An example of Phantom References is provided under Reference Queues below.

**4. Reference Queues**

ReferenceQueue is the mechanism provided by the JVM to be notified when a referenced instance is about to be garbage collected. Reference Queues can be used with all of the reference types by passing it to the constructor. When creating a PhantomReference, it is a must to provide a Reference Queue.

The use of reference queue is as follows.

public class TestPhantomRefQueue {

public static void main(String[] args)

throws InterruptedException {

Object obj = new Object();

final ReferenceQueue queue = new ReferenceQueue();

PhantomReference pRef =

new PhantomReference(obj,queue);

obj = null;

new Thread(new Runnable() {

public void run() {

try {

System.out.println("Awaiting for GC");

// This will block till it is GCd

PhantomReference pRef =

(PhantomReference) queue.remove();

System.out.println("Referenced GC'd");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}).start();

// Wait for 2nd thread to start

Thread.sleep(2000);

System.out.println("Invoking GC");

System.gc();

}

}

The output would be

Awaiting for GC

Invoking GC

Referenced GC'd

**5. WeakHashMap**

java.util.WeakHashMap is a special version of the HashMap, which uses weak references as the key. Therefore, when a particular key is not in use anymore, and it is garbage collected, the corresponding entry in the WeakHashMap will magically disappear from the map. And the magic relies on ReferenceQueue mechanism explained before to identify when a particular weak reference is to be garbage collected. This is useful when you want to build a cache based on weak references. In more sophisticated requirements, it is better to write your own cache implementation.

In this rather long article, we have covered the basics of the Referencing API provided by Java Specification. The content that we have discussed here are the basics of the referencing API, and you might find it helpful to glance through the [Java Docs](http://download.oracle.com/javase/6/docs/api/java/lang/ref/package-summary.html) for the Referencing API.

# Difference between WeakReference vs SoftReference vs PhantomReference vs Strong reference in Java

# <http://www.javacodegeeks.com/2014/03/difference-between-weakreference-vs-softreference-vs-phantomreference-vs-strong-reference-in-java.html>

WeakReference and SoftReference were added into Java API from long time but not every Java programmer is familiar with it. Which means there is a gap between where and *how to use WeakReference and SoftReference in Java*. Reference classes are particularly important in context of [How Garbage collection works](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html). As we all know that Garbage Collector reclaims memory from objects which are eligible for garbage collection, but not many programmer knows that this eligibility is decided based upon which kind of references are pointing to that object. This is also main *difference between WeakReference and SoftReference in Java*. Garbage collector can collect an object if only weak references are pointing towards it and they are eagerly collected, on the other hand Objects with SoftReference are collected when JVM absolutely needs memory. These special behaviour of SoftReference and WeakReference makes them useful in certain cases e.g. SoftReference looks perfect for implementing caches, so when JVM needs memory it removes object which have only SoftReference pointing towards them. On the other hand WeakReference is great for storing meta data e.g. storing ClassLoader reference. If no class is loaded then no point in keeping reference of ClassLoader, a WeakReference makes [ClassLoader](http://javarevisited.blogspot.sg/2012/12/how-classloader-works-in-java.html" \o ")eligible for Garbage collection as soon as last strong reference removed. In this article we will explore some more about various reference in Java e.g. Strong reference and Phantom reference.

## **WeakReference vs SoftReference in Java**

For those who don’t know there are four kind of reference in Java :

1. Strong reference
2. Weak Reference
3. Soft Reference
4. Phantom Reference

Strong Reference is most simple as we use it in our day to day programming life e.g. in the code, String s = “abc” , reference variable **s** has strong reference to String object “abc”. Any object which has Strong reference attached to it is *not eligible for garbage collection*. Obviously these are objects which is needed by Java program. Weak Reference are represented using java.lang.ref.WeakReference class and you can create Weak Reference by using following code :

|  |  |
| --- | --- |
| 1 | Counter counter = new Counter(); // strong reference - line 1 |
| 2 | WeakReference<Counter> weakCounter = new WeakReference<Counter>(counter); //weak reference |

|  |  |
| --- | --- |
| 3 | counter = null; // now Counter object is eligible for garbage collection |

Now as soon as you make strong reference counter = null, counter object created on line 1 becomes eligible for garbage collection; because it doesn’t have any more Strong reference and Weak reference by reference variable weakCounter can not prevent Counter object from being garbage collected.  On the other hand, had this been Soft Reference, Counter object is not garbage collected until [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html)absolutely needs memory. Soft reference in Java is represented using java.lang.ref.SoftReference class. You can use following code to create a SoftReference in Java

|  |  |
| --- | --- |
| 1 | Counter prime = new Counter();  // prime holds a strong reference - line 2 |
| 2 | SoftReference<Counter> soft = new SoftReference<Counter>(prime) ; //soft reference variable has SoftReference to Counter Object created at line 2 |

|  |  |
| --- | --- |
| 3 |  |
| 4 | prime = null;  // now Counter object is eligible for garbage collection but only be collected when JVM absolutely needs memory |

After making strong reference null, Counter object created on line 2 only has one soft reference which can not prevent it from being garbage collected but it can delay collection, which is eager in case of WeakReference. Due to this major*difference between SoftReference and WeakReference*, SoftReference are more suitable for **caches**and WeakReference are more suitable for storing **meta data**. One convenient example of WeakReference is WeakHashMap, which is another implementation of Map interface like [HashMap](http://java67.blogspot.sg/2013/08/best-way-to-iterate-over-each-entry-in.html" \o ")or [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html" \o ")but with one unique feature. WeakHashMap wraps keys as WeakReference which means once strong reference to actual object removed, WeakReference present internally on WeakHashMap doesn’t prevent them from being Garbage collected.

Phantom reference is third kind of reference type available in java.lang.ref package. Phantom reference is represented by java.lang.ref.PhantomReference class. Object which only has Phantom reference pointing them can be collected whenever Garbage Collector likes it. Similar to WeakReference and SoftReference you can create PhantomReference by using following code :

|  |  |
| --- | --- |
| 1 | DigitalCounter digit = new DigitalCounter(); // digit reference variable has strong reference - line 3 |
| 2 | PhantomReference<DigitalCounter> phantom = new PhantomReference<DigitalCounter>(digit); // phantom reference to object created at line 3 |

|  |  |
| --- | --- |
| 3 |  |
| 4 | digit = null; |

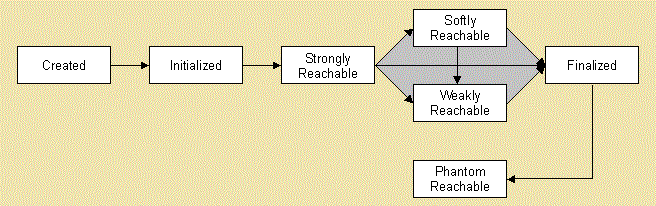
As soon as you remove Strong reference, DigitalCounter object created at line 3 can be garbage collected at any time as it only has one more PhantomReference pointing towards it, which can not prevent it from GC’d.

Apart from knowing about WeakReference, SoftReference, PhantomReference and WeakHashMap there is one more class called ReferenceQueue which is worth knowing. You can supply a ReferenceQueue instance while creating any WeakReference, SoftReference or PhantomReference as shown in following code :

|  |  |
| --- | --- |
| 1 | ReferenceQueue refQueue = new ReferenceQueue(); //reference will be stored in this queue for cleanup |
| 2 |  |

|  |  |
| --- | --- |
| 3 | DigitalCounter digit = new DigitalCounter(); |
| 4 | PhantomReference<DigitalCounter> phantom = new PhantomReference<DigitalCounter>(digit, refQueue); |

Reference of instance will be appended to ReferenceQueue and you can use it to perform any clean-up by polling ReferenceQueue. An Object’s life-cycle is nicely summed up by this diagram.

[](http://a3ab771892fd198a96736e50.javacodegeeks.netdna-cdn.com/wp-content/uploads/2014/03/Weak-Strong-Soft-and-Phantom-Reference-in-Java.gif)

That’s all on **Difference between WeakReference and SoftReference in Java**. We also learned basics of reference classes e.g. Weak, soft and phantom reference in Java and WeakHashMap and ReferenceQueue. Careful use of reference can assist Garbage Collection and result in better memory management in Java.

# [Types Of References In Java : Strong, Soft, Weak And Phantom](http://javaconceptoftheday.com/types-of-references-in-java-strong-soft-weak-and-phantom/)

# <http://javaconceptoftheday.com/types-of-references-in-java-strong-soft-weak-and-phantom/>

# One of the beauty of the Java language is that it doesn’t put burden of memory management on the programmers. Java automatically manages the memory on the behalf of the programmers. Java programmers need not to worry about freeing the memory after the objects are no more required. **Garbage Collector Thread** does this for you. This thread is responsible for sweeping out unwanted objects from the memory. But, you have no control over garbage collector thread. You can’t make it to run whenever you want. It is up to JVM which decides when to run garbage collector thread. But, with the introduction of **java.lang.ref** classes, you can have little control over when your objects will be garbage collected.

Depending upon how objects are garbage collected, references to those objects in java are grouped into 4 types. They are,

1) Strong References

2) Soft References

3) Weak References

4) Phantom References

Let’s discuss these reference types in detail.

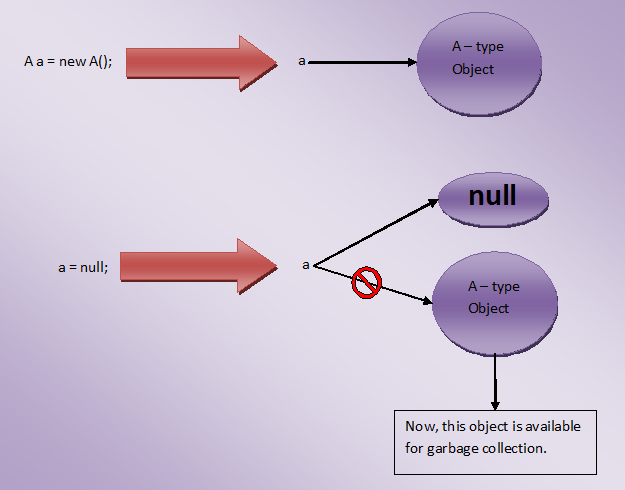
## 1) Strong References

These type of references we use daily while writing the code. Any object in the memory which has active **strong reference** is not eligible for garbage collection. For example, in the below program, reference variable **‘a’** is a strong reference which is pointing to class A-type object. At this point of time, this object can’t be garbage collected as it has strong reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class A  {      //Class A  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            a = null;    //Now, object to which 'a' is pointing earlier is eligible for garbage collection.      }  } |

If you make reference **‘a’** to point to null like in Line 12, then, object to which ‘a’ is pointing earlier will become eligible for garbage collection. Because, it will have no active references pointing to it. This object is most likely to be garbage collected when garbage collector decides to run.

Look at the below picture for more precise understanding.



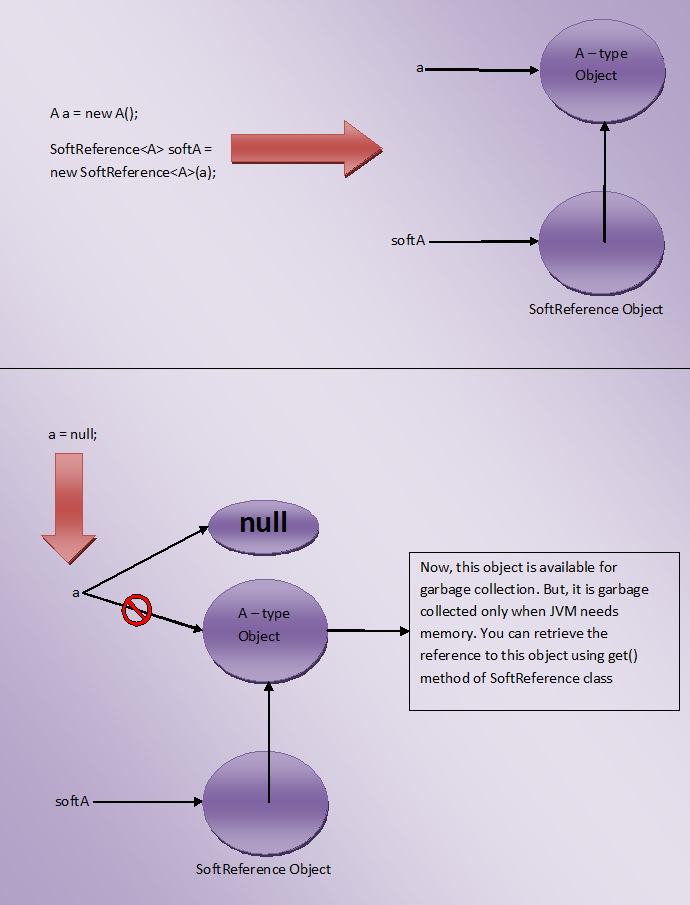
## 2) Soft References

The objects which are softly referenced will not be garbage collected (even though they are available for garbage collection) until JVM badly needs memory. These objects will be cleared from the memory only if JVM runs out of memory. You can create a soft reference to an existing object by using  **java.lang.ref.SoftReference** class. Below is the code example on how to create a soft reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Soft Reference to A-type object to which 'a' is also pointing            SoftReference<A> softA = new SoftReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is eligible for garbage collection. But, it will be garbage collected only when JVM needs memory.            a = softA.get();    //You can retrieve back the object which has been softly referenced      }  } |

In the above example, you create two strong references – ‘**a**‘ and ‘**softA**‘. ‘a’ is pointing to A-type object and ‘softA’ is pointing to SoftReference type object. This SoftReference type object is internally referring to A-type object to which ‘a’ is also pointing. When ‘a’ is made to point to null, object to which ‘a’ is pointing earlier becomes eligible for garbage collection. But, it will be garbage collected only when JVM needs memory. Because, it is softly referenced by ‘softA’ object.

Look at the below picture for more clarity.



One more use of SoftReference class is that you can retrieve back the object which has been softly referenced. It will be done by using **get()** method. This method returns reference to the object if object is not cleared from the memory. If object is cleared from the memory, it will return null.

## 3) Weak References

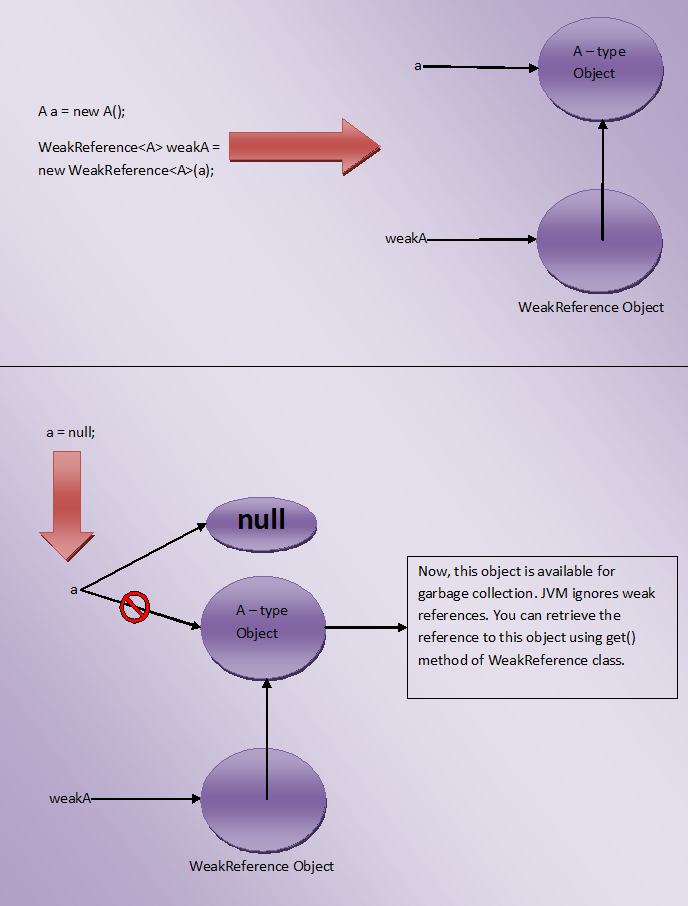
JVM ignores the **weak references**. That means objects which has only week references are eligible for garbage collection. They are likely to be garbage collected when JVM runs garbage collector thread. JVM doesn’t show any regard for weak references.

Below is the code which shows how to create weak references.

[?](http://javaconceptoftheday.com/types-of-references-in-java-strong-soft-weak-and-phantom/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating Weak Reference to A-type object to which 'a' is also pointing.            WeakReference<A> weakA = new WeakReference<A>(a);            a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection.            a = weakA.get();    //You can retrieve back the object which has been weakly referenced.      }  } |

Look at the below picture for more clear understanding.



You may think that what is the use of creating weak references if they are ignored by the JVM, Use of weak reference is that you can retrieve back the weakly referenced object if it is not yet removed from the memory. This is done using get() method of WeakReference class. It will return reference to the object if object is not yet removed from the memory.

## 4) Phantom References

The objects which are being referenced by **phantom references** are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called**‘reference queue’**. They are put in a reference queue after calling finalize() method on them. You can’t retrieve back the objects which are being phantom referenced. That means calling get() method on phantom reference always returns null.

Below example shows how to create Phantom References.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class A  {      //A Class  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();      //Strong Reference            //Creating ReferenceQueue            ReferenceQueue<A> refQueue = new ReferenceQueue<A>();            //Creating Phantom Reference to A-type object to which 'a' is also pointing            PhantomReference<A> phantomA = new PhantomReference<A>(a, refQueue);            a = null;    //Now, A-type object to which 'a' is pointing earlier is available for garbage collection. But, this object is kept in 'refQueue' before removing it from the memory.            a = phantomA.get();    //it always returns null      }  } |