Java Reference – 2022

This breakdown of weak, soft, and phantom references explains how they impact GC and memory allocation as well as some ideal use cases. It is a bit like in a restaurant. In the beginning, you can accommodate customers with empty tables, but when you don’t have empty tables anymore, you need to check if some of the already-allocated tables have got free in the meanwhile.

**SoftReference**: Soft reference objects are cleared at the discretion of the garbage collector in response to memory demand. Soft references are most often used to implement memory-sensitive caches. All soft references to softly reachable objects are guaranteed to have been cleared before the virtual machine throws an OutOfMemoryError.

**WeakReference**: Weak reference objects do not prevent their referents from being made finalizable, finalized, and then reclaimed. Weak references are most often used to implement canonicalizing mappings. (Here, Canonicalizing mappings means mapping only reachable object instances.)

**PhantomReference**: Phantom reference objects 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. 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.

**So in brief: Soft references try to keep the reference. Weak references don’t try to keep the reference. Phantom references don’t free the reference until cleared.**

To reuse (and stretch) our restaurant metaphor one last time: A SoftReference is like a customer that says, "I’ll leave my table only when there are no other tables available." A WeakReference is like someone ready to leave as soon as a new customer arrives. A PhantomReference is like someone ready to leave as soon as a new customer arrives, but actually not leaving until the manager gives them permission.

Reference<HeavyList> softRef = new SoftReference<>(curr, queue);

Reference<HeavyList> weakRef = new WeakReference<>(curr, queue);

Reference<HeavyList> phantomRef = new PhantomReference<>(curr, queue);

We can always reach the referenced object with the method **get()**. In the case of Weak and Soft, get will return the actual object if still active — that is, if it is reachable by other objects. In case the object has been collected, get() will return null.

This opens a possible problem if someone manages to “resurrect” the object using the reference get() during the finalization. For this reason, Phantom always returns null in the get() regardless of whether the object is still active. In this way, we can pass a PhantomReference to another object without risking that it will store a new, hard reference to it.

The other parameter in the constructor is the **ReferenceQueue**. To understand why is important, we have to consider how we know when the referenced object is finalized.

For Soft and Weak references, we can check the get() method, but it would be very time consuming if we have a big list of references. Moreover, for Phantom references, we cannot use it at all.

private static int removeRefs(ReferenceQueue queue, Set < Reference < HeavyList >> references) {

int removed = 0;

while (true) {

Reference r = queue.poll();

if (r == null) break;

references.remove(r);

removed++;

}

return removed;

}

If **queue.poll()** returns null, then the queue is empty. A less naive approach is to create a separate thread and call **queue.remove()**, which will block until there is something to remove.

Just remember that whilst Weak and Soft references are put in the queue after the object is finalized, Phantom references are put in the queue **before**. If for any reason you don’t poll the queue, the actual objects referenced by Phantom will not be finalized, and you can incur an OutOfMemory error.

**Possible Uses**

**SoftReferences** can be used **to implement a cache that can grow without risking an application crash**. To do this, you need to implement a Map interface in which values are stored, wrapped inside a SoftReference. SoftReferences will keep the objects alive until there is memory available on the heap, but it will discard them before an OutOfMemoryError.

**WeakReferences** can be used, for example, **to store some information related to an object until the object gets finalized**. To do this, you can implement a Map in which the keys are wrapped in a WeakReference. As soon as GC reclaims the key object, you can remove the value as well.

**PhantomReferences** can be used to **notify you when some object is out of scope to do some resource cleanup**. Remember that the object.finalize() method is not guaranteed to be called at the end of the life of an object, so if you need to close files or free resources, you can rely on Phantom. Since Phantom doesn't have a link to the actual object, a typical pattern is to derive your own Reference type from Phantom and add some info useful for the final freeing, for example filename.

Reference: [Weak, Soft, and Phantom References in Java (and Why They Matter) - DZone Java](https://dzone.com/articles/weak-soft-and-phantom-references-in-java-and-why-they-matter)

**What is a softly reachable object?**

Student strongReference = new Student();

WeakReference<Student> weakReference = new WeakReference<>(strongReference);

**Similarly**

Student strongRef = new Student();

SoftReference<Student> softReference = new SoftReference<>(strongRef);

During garbage collection if an object in heap has a strong reference to it then it survives, if it does not has strong reference but has WeakReference then it won't survive. It is used to avoid leak when objects are passed out side the life cycle manager context.

SoftReference are like weakreference but they survive garbage collection cycle till memory is available in plenty.

An object is *softly reachable* if there are no Strong references and have SoftReferences. As an object having only weak reference is eligible for garbage collection and on the other hand an object having only soft reference is more egar to survive garbage collection (as compared to weak reference) hence

1. **An object which has No Strong Reference and has only Soft or Weak Reference is Softly Reachable**
2. An object having only WeakReference and no Strong or soft references is Weekly Reachable
3. An object with atleast one Strong reference with or without any soft or weak references is Strongly Reachable.

Both the below cases The object in heap is softly reachable.

Student stRef = new Student();

SoftReference <Student> sfRef = new SoftReference<>(stRef);

stRef = null;

**Or**

SoftReference <Student> sfRef = new SoftReference<>(new Student());

Student strongReference = new Student();

SoftReference<Student> softReference = new SoftReference<>(strongReference);

strongReference = null; // object in heap is softly reachable now

Student anotherStrongReference = softReference.get();

if(anotherStrongReference != null){

// you have a strong reference again

}

Reference: [java - What is a softly reachable object? - Stack Overflow](https://stackoverflow.com/questions/51891168/what-is-a-softly-reachable-object#:~:text=An%20object%20is%20softly%20reachable,compared%20to%20weak%20reference)%20hence)

**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.

Weak references are collected eagerly. If GC finds that an object is weakly reachable (reachable only through weak references), it'll clear the weak references to that object immediately. As such, they're good for keeping a reference to an object for which your program also keeps (strongly referenced) "associated information" somewere, like cached reflection information about a class, or a wrapper for an object, etc.

**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 suppl**y. 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.

SoftReferences on the other hand are good for caching external, recreatable resources as the GC typically delays clearing them. It is guaranteed though that all SoftReferences will get cleared before OutOfMemoryError is thrown, so they theoretically can't cause an OOME. Typical use case example is keeping a parsed form of a contents from a file. You'd implement a system where you'd load a file, parse it, and keep a SoftReference to the root object of the parsed representation. Next time you need the file, you'll try to retrieve it through the SoftReference. If you can retrieve it, you spared yourself another load/parse, and if the GC cleared it in the meantime, you reload it. That way, you utilize free memory for performance optimization, but don't risk an OOME.

So, the decision depends on usage - if you're caching information that is expensive to construct, but nonetheless reconstructible from other data, use soft references - if you're keeping a reference to a canonical instance of some data, or you want to have a reference to an object without "owning" it (thus preventing it from being GC'd), use a weak reference.

Reference: [reference - What's the difference between SoftReference and WeakReference in Java? - Stack Overflow](https://stackoverflow.com/questions/299659/whats-the-difference-between-softreference-and-weakreference-in-java)

A **Strong reference** is a normal reference that protects the referred object from collection by GC. i.e. Never garbage collects.

A **Soft reference** is eligible for collection by garbage collector, but probably won't be collected until its memory is needed. i.e. garbage collects before OutOfMemoryError.

A **Weak reference** is a reference that does not protect a referenced object from collection by GC. i.e. garbage collects when no Strong or Soft refs.

A **Phantom reference** is a reference to an object is phantomly referenced after it has been finalized, but before its allocated memory has been reclaimed.

**Analogy:** Assume a JVM is a kingdom, Object is a king of the kingdom, and GC is an attacker of the kingdom who tries to kill the king(object).

* When King is **Strong**, GC can not kill him.
* When King is **Soft**, GC attacks him but King rule the kingdom with protection until resource are available.
* When King is **Weak**, GC attacks him but rule the kingdom without protection.
* When king is **Phantom**, GC already killed him but king is available via his soul.

From: [Difference between WeakReference vs SoftReference vs PhantomReference in Java? Example (javarevisited.blogspot.com)](https://javarevisited.blogspot.com/2014/03/difference-between-weakreference-vs-softreference-phantom-strong-reference-java.html)

**Counter** counter **=** **new** **Counter**(); // strong reference - line 1

**WeakReference**<**Counter**> weakCounter **=** **new** **WeakReference**<**Counter**>(counter);

//weak reference

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 absolutely needs memory.

**Counter** prime **=** **new** **Counter**(); // prime holds a strong reference - line 2

**SoftReference**<**Counter**> soft **=** **new** **SoftReference**<**Counter**>(prime) ;

//soft reference variable has SoftReference to Counter Object created at line 2

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.

<https://plumbr.io/handbook/gc-tuning-in-practice/weak-soft-and-phantom-references#:~:text=Unlike%20soft%20and%20weak%20references,cleared%20or%20themselves%20become%20unreachable>.