References in JAVA

Introduction

There are 4 types of references in JAVA:
– **Strong References**
– **Soft References**
– **Weak References**
– **Phantom References**

Those references differ only by the way the **garbage collector** manages them. If you've never heard of them, it means that you were only using the strong ones. Knowing the difference can help you, especially if you need to store temporary objects and can't use a real caching library like eHcache or Guava.

Since those types are stongly related to the JVM garbage collector, I'll briefly recall some information about garbage collection in JAVA and then I'll present the different types.

The garbage collector

The main difference between Java and C++ is the **memory management**. In Java, the developer doesn't need to know how memory works (but he should know!) because the JVM takes care of this part with its garbage collector.

When you create an object, it's allocated by the JVM in its **heap**. The heap is a limited amount of space in memory. Therefore, the JVM often needs to delete objects in order to release space. To destroy an object, the JVM needs to know if this object is active or inactive. An object is still in use if it is referenced (transitively) by a "***garbage collection root***".

For example:

- If an object C is referenced by an object B and B is referenced by an

object A and A is referenced by a *garbage collection root*, then C, B and A are considered active (Case 1).
- But, if B is not referenced anymore by A then C and B are not active anymore and can be destroyed (Case 2).

Since this post is not about the garbage collector, I won't go deeper in the explanation but FYI there are 4 types of garbage collection roots:

1. **Local variables**
2. **Active Java threads**
3. **Static variables**
4. **JNI References** which are Java objects containing native code and not memory managed by the jvm

Oracle doesn't specify how to manage memory so each JVM implementation has its own set of algorithms. But the idea is always the same:
– the JVM uses a recurent algorithm that looks for inactive objects and marks them
– marked objects are finalized (call to finalize() method) then destroyed
– the JVM sometimes moves a part of the remaining objects in order to rebuild large areas of free contiguous space in the heap

Problem

If the JVM manages the memory, why do you need to care? Because it doesn't mean that you can't have **memory leaks**!

Most of the time you are using garbage collection roots without realizing it. For example, lets say you need to store some objets during the lifetime of your program (because their intialization is costly). You'll likely use a static colllection (List, Map, …) to store and retrieve those objects anywhere in your code:

But, by doing so, you prevent the JVM from destroying the objects within the

collection. By mistake you might and up with an **OutOfMemoryError**. For example:

The output is:

> *Exception in thread "main" java.lang.OutOfMemoryError: Java heap space*

Java provides different types of references to avoid OutOfMemoryError.

Some types allows the JVM to release objects even if they are still needed by the program. It's the developer's responsability to handle these cases.

Strong Reference

The strong reference is the standard reference. When you create on object obj like that:

you are creating a strong reference called "obj" to the newly created instance of MyClass. When the garbage collector looks for inactive objects, it only checks if the objets are **strongly reachable** which means transitively linked to a garbage collection root by strong references.

Using this type of references forces the JVM to keep the objects in the heap until the objects are not used as described in the part "The garbage collector".

Soft Reference

According to the java API soft reference are:

> *"Soft reference objects, which are cleared at the discretion of the garbage collector in response to memory demand"*

Which means that the behavior of soft references might change if you run your program on different JVMs (Oracle's Hotspot, Oracle's JRockit, IBM's J9, …).

Lets have a look at Oracle's JVM Hotspot (the standard and most used JVM) to see how it manages soft references. According to Oracle documentation:

> *"The default value is 1000 ms per megabyte, which means that a soft reference will survive (after the last strong reference to the object has been collected) for 1 second for each megabyte of free space in the heap"*

Here is a concrete example: Lets assume the heap is 512 Mbytes and there are 400Mbytes free.

We create an object **A**, softly referenced to an object **cache** and strongly referenced **A** to an object **B**. Since **A** is strongly referenced to **B**, it's strongly reachable and won't be deleted by the garbage collector (case 1).

Imagine now that **B** is deleted, so **A** is only softly referenced to the **cache** object. If object **A** is not strongly referenced during the next 400 secondes, it's going to be deleted after the timeout (case 2).

Here is how you can manipulate a soft reference:

But even if the soft referenced objects are automatically deleted by the garbage collector, the **soft references** (which are also objects) **are not deleted!** So, you still need to clear them. For example with a low heap size like 64 Mbytes (Xmx64m) the following code gives an OutOfMemoryException despite the use of soft references.

The output code is:

size of cache:1
size of cache:200001
size of cache:400001
size of cache:600001
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded

Oracle provides a **ReferenceQueue** that is filled with soft references when a referenced object is only softly reachable. Using this queue, you can clear the soft references and avoid an OutOfMemoryError.

Using a ReferenceQueue, the same code as above with the same heap size (64 Mbytes) but with more data to store (5 million vs 1 million) works:

The output is:

End, removed soft references=4976899

Soft references are useful when you need to store many objects that can be (costly) re-instanciate if they are deleted by the JVM.

Weak Reference

The weak reference is a concept even more volatile than soft references. According to the JAVA API:

"Suppose that the garbage collector determines at a certain point in time that an object is weakly reachable. At that time it will atomically clear all weak references to that object and all weak references to any other weakly-reachable objects from which that object is reachable through a chain of strong and soft references. At the same time it will declare all of the formerly weakly-reachable objects to be finalizable. At the same time or at some later time it will enqueue those newly-cleared weak references that are registered with reference queues."

Which means that when the garbage collector checks all the objects, if it detects an object with only weak references to a garbage collection root (i.e. no strong or soft references linked to the object), this object will be marked for removal and deleted ASAP. The way to use a WeakReference is exactly the same as using a SoftReference. So, look at the examples in part “soft references”.

Oracle provides a very interesting class based on weak references: the WeakHashMap. This map has the particularity to have weak referenced keys. The WeakHashMap can be used as a standard Map. The only difference is that it will **automatically clear itself** after the keys are destroyed from the heap:

For example, I’ve used a WeakHashMap for the following problem: Storing multiple information of transactions. I’ve used this structure: WeakHashMap<String,Map<K,V>> where they key of the WeakHashMap was a String containing the Id of the transaction and the “simple” Map was the information I needed to keep during the lifetime of the transaction. With this structure I was sure to get my information in the WeakHashMap because the String containing the transaction ID couldn’t be destroyed until the end of the transaction and I didn’t have to care for cleaning up the Map.

Oracle advises to use WeakHashMap as “canonicalized” mapping.

Phantom Reference

During the garbage collection process, objects without a strong/soft reference to a garbage collection root are deleted. Before beeing deleted, the method finalize() is called. When an object is finalized but not deleted (yet) it becomes “phantom reachable” which means there is only a phantom reference between the object and a garbage collection root.

Unlike soft and weak references, using an explicit phantom reference to an object prevents the object from being deleted. The programmer needs to explicitly or implicitly remove the phantom reference so that the finalized object can be destroyed. To explicitly clear a phantom reference the programmer needs to use a **ReferenceQueue** which is filled with a phantom reference when an object is

finalized.

A phantom reference cannot retrieve the referenced object: The get() method of the phantom reference always returns null so that a programmer cannot make a phantom reachable object strongly/softly/weakly reachable again. It makes sense because a phantom reachable object has been finalized so it migth not work anymore if for example the overrided finalize() function has cleared resources.

I don’t see how a phantom reference might be usefull since the referenced object cannot be accessed. A use case could be if you need to do an action after an object is finalized and you can’t (or don’t wan’t for performance reasons) do the specific action in the finalize() method of this object.

Conclusion

I hope you now have a better idea about these references. Most of the times, you won’t need to explicitly use them (and you shouldn’t). But, many frameworks are using them. And if you like to understand how stuff works it’s good to know this concept.

If you’re more a video guy, people from Webucator have turned this post into a video available on YouTube.