Types of References in Java

Software Engineering | Java

Reading time: 25 minutes | Coding time: 10 minutes

One of the most popular facilities in the Java Virtual Machine is the Garbage Collector. It is responsible for scanning the memory in runtime, locating and deleting the objects that will no longer be used by the Java Virtual Machine (JVM). Hence, it manages the memory resources this way and takes away this obligation from the developer.

Although, the Garbage Collector behaves slightly different depending on the types of reference that is attached to an object. This allowing the programmer to control memory resources if required by the application.

There are 4 types of reference in Java Language:

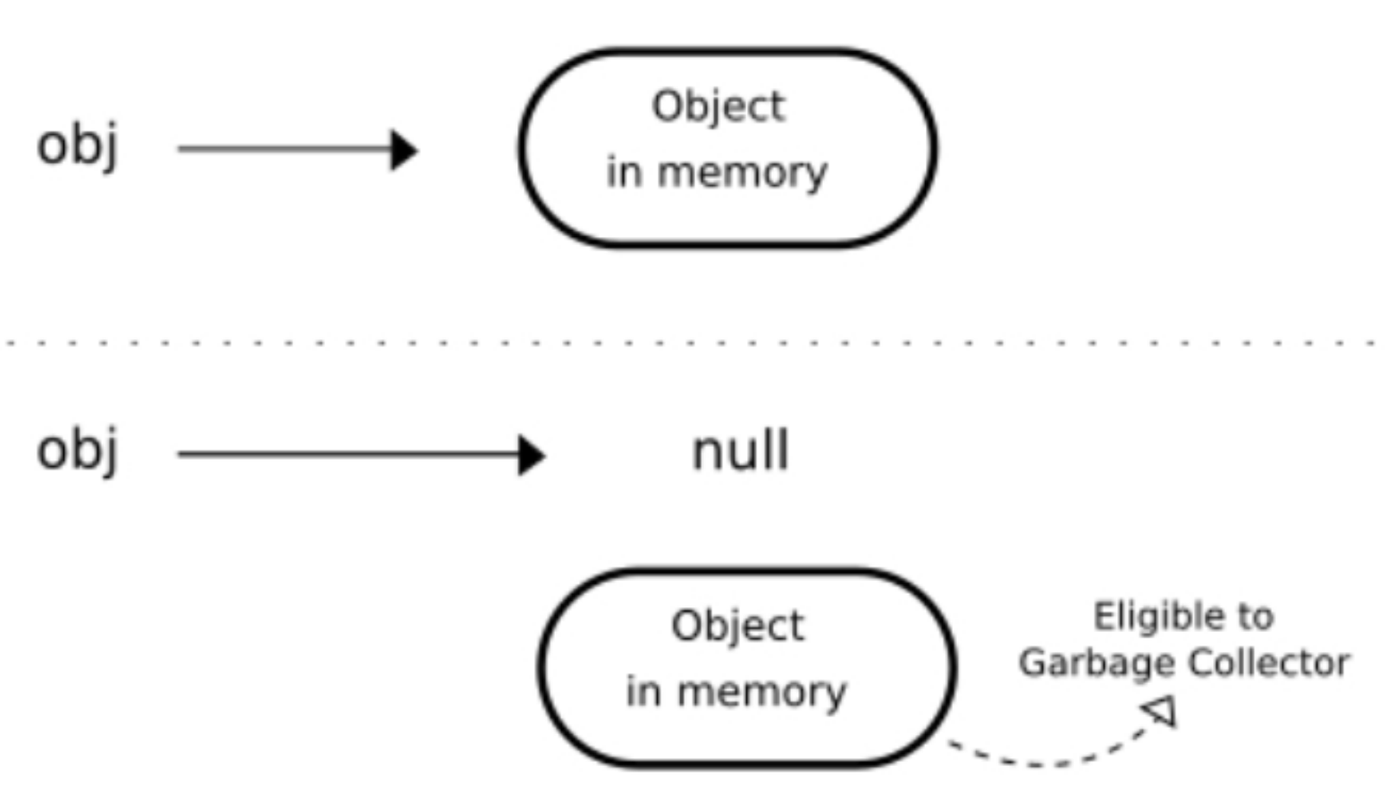
- Strong Reference
- Weak Reference
- Soft Reference
- Phantom Reference

Strong Reference

Strong References are the most common type of reference in Java. Every time we directly point a variable to an object, we are indeed strongly linking a memory section to our application.

```
Object obj = new Object(); // Creates a strong reference
```

With just one line of code, the Java Virtual Machine separates space in memory, instantiating a new object from a class, and give to the developer a way to interact with it through the `obj` variable. This communication only happens while the program has a memory reference, determined by `obj`. At the moment that the variable value changes to another object - or null - we will be lost the interaction with the first object, and the Garbage Collector can go into action.



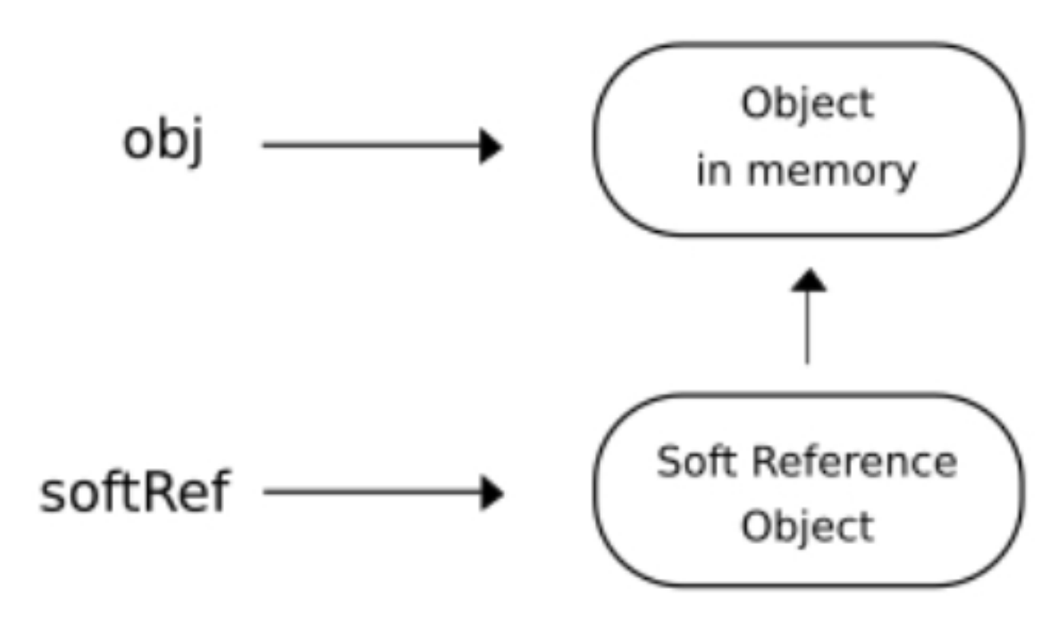
The primary rule from the Garbage Collector resumes to: **reclaim any memory space with no strong references pointing to it**.

The image above can illustrate this condition when the null value is attached to the variable and abandon the object in memory. However, despite this behavior, there are other types of references who change the GC response and can create useful situations for a Java application.

Soft Reference

Soft References are a special kind of reference, which slightly changes the default behavior of the Garbage Collector, allowing the GC to maintain objects without strong references until the Java Virtual Machine runs out of memory. In this last case, the Garbage Collector will try to save the application scanning the memory and deleting all the soft references.

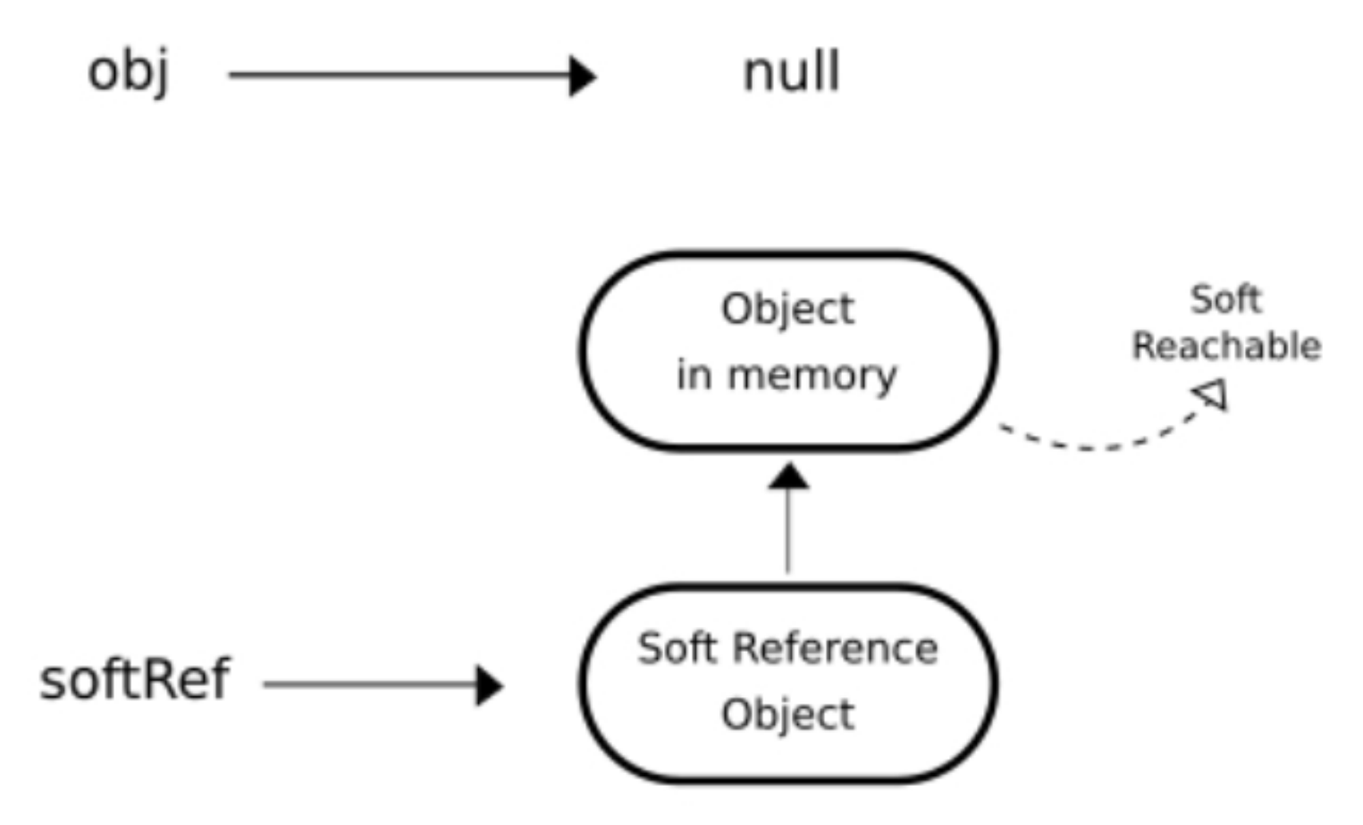
So, even if a variable does not refer directly to an object, this one can be ignored by the GC while the JVM doesn't need more resources. A representation of this indirectly reference it's described in the image below.



The following code shows how to implement a soft reference using the `SoftReference` class. To use this and the other types of reference resources it simply has to import the `java.lang.ref` library.

```
import java.lang.ref.*

// ...

Object obj = new Object();

// New Soft Reference to obj object
SofReference<Object> softRef = new SoftReference<Object>(obj);
```

Notice in the code above we have a soft and two strong references, this happens because all the variables and objects directly accessible by programming are strong references. If we change the `obj` value to another memory address and maintain the `softRef` value, it becomes soft reachable. The GC will reclaim this memory space only in the lack of resources.



This feature can be useful for objects with dynamic use, which are frequently changed but the cost to create a new object are too expensive, just like in object pools. Which tries to increase performance always having one of these classes instantiated in memory, but much of the time the objects may even be ignored by the JVM and can be deleted.

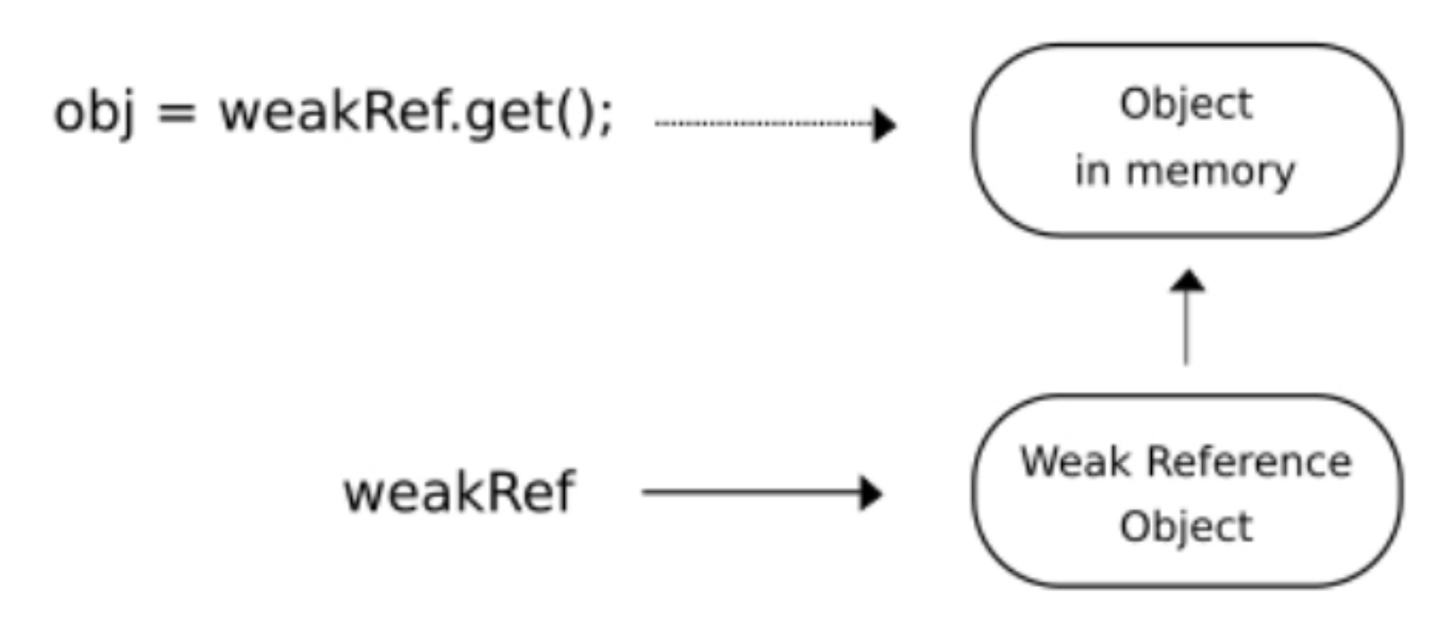
Weak Reference

Weak References have another interpretation from the Garbage Collector. While the types mentioned before are, for default, preserved in memory, if an object has only a weak reference attached to him when the GC is running, then it will be reclaimed even if the virtual machine doesn't need more space.

We can create a new weak reference using the WeakReference class, just like showed before with the `SoftReference`. We can also use the `get()` method for rescue the object from both soft and weak reference. But, if the GC has already reclaimed the memory space, then the `get()` method will return null.

```
Object obj = new Object();

// New Weak Reference to obj object
WeakReference<Object> softRef = new WeakReference<Object>(obj);

// Removing the Strong Reference
obj = null;

// Explicit call of the Garbage Collector
System.gc();

// Returns the object reference if it remains in memory,
// otherwise returns null
obj = softRef.get();
```

This can be useful to implement cache data and hash maps. Being able to be discarded at any time, or also too large objects which the developer wants to make sure they were deleted and don't have copies in memory.



Phantom Reference

The last type of reference it's even more special than the mentioned before. The Garbage Collector can delete phantom referenced objects, but before that, the JVM append the reference object in a queue named "Reference Queue" responsible for telling to the application when the GC removes the phantom referenced object from memory. For this case, the `get()` method always return null.

```
Object obj = new Object();

// Creating a reference queue
ReferenceQueue<obj> refQueue = new ReferenceQueue<obj>();

// New Phantom Reference to obj object in a queue
PhantomReference<Object> phantomRef = new PhantomReference<Object>(obj, refQueue);

// Removing the Strong Reference
obj = null;

// Explicit call of the Garbage Collector
System.gc();

// get() method always returns null
phantomRef.get();
```

When an object becomes phantom reachable - only have phantom references - the GC will append it in the `refQueue` at the final of its iteration. We can recover this information using the `poll()` method from the reference queue class.

```
// Removing the Strong Reference
obj  = null;
System.out.println("Queue: " + queue.poll()); // empty queue

// Explicit call of the Garbage Collector
System.gc();

System.out.println("Queue: " + queue.poll()); // show PhantomReference object
```

A developer can use this resource to have sure that an object was deleted or not. Also, it can be used to make cleanup actions more flexible without explicit override the `finalize()` method.

Aplications

With the types of references, a Java developer can have access to mechanisms to advanced memory management. These features allow to dynamically change the Garbage Collector behave, handing some of its powers to the programmer.

Gabriel Lechenco
Bachelor of Engineering (2016 to 2021) in Computer Engineering at The Federal University of Technology (UTFPR), Paraná, Brazil

Read More

OpenGenus Foundation

Tags

Software Engineering | Java

Start Discussion — 0 replies