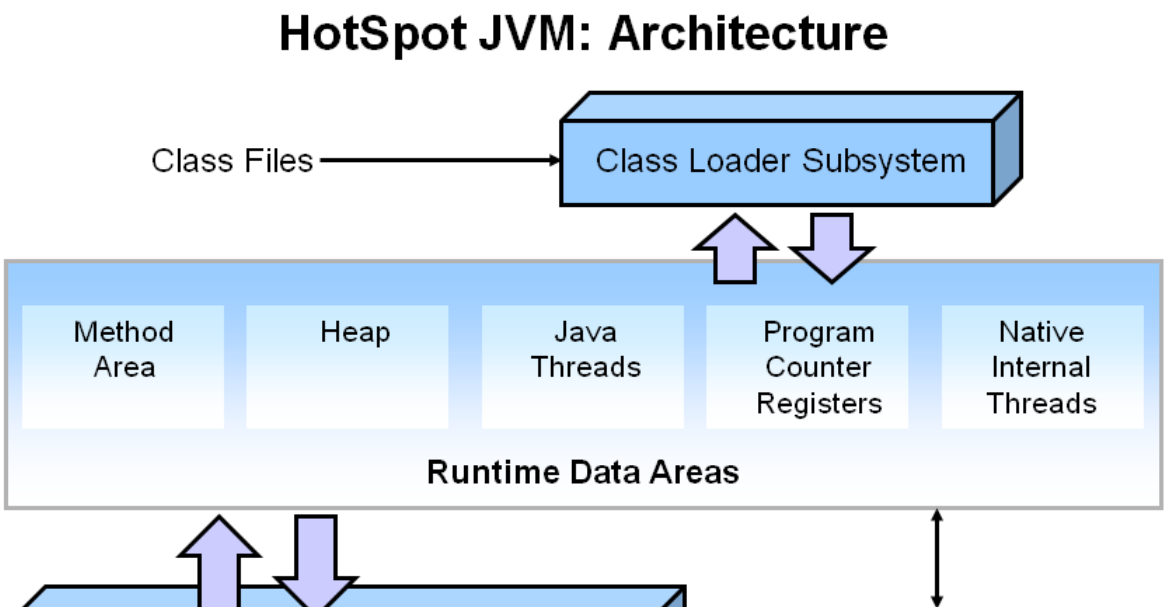
JAVA memory management and Garbage Collection...

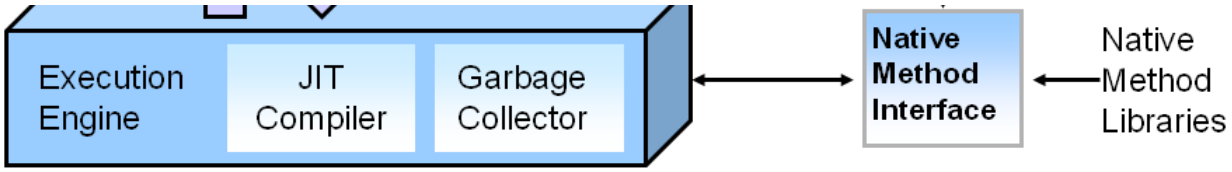
Kiran Chowdhary
Dec 30, 2018 · 6 min read

Java Memory Management, with its built-in garbage collection, is one of the language's finest achievements. It allows developers to create new objects without worrying explicitly about memory allocation and deallocation, because the garbage collector automatically reclaims memory for reuse. This enables faster development with less boilerplate code, while eliminating memory leaks and other memory-related problems. At least in theory.

Ironically, Java garbage collection seems to work too well, creating and removing too many objects. Most memory-management issues are solved, but often at the cost of creating serious performance problems. Making garbage collection adaptable to all kinds of situations has led to a complex and hard-to-optimize system. In order to wrap your head around garbage collection, you need first to understand how memory management works in a Java Virtual Machine (JVM).

Hotspot JVM



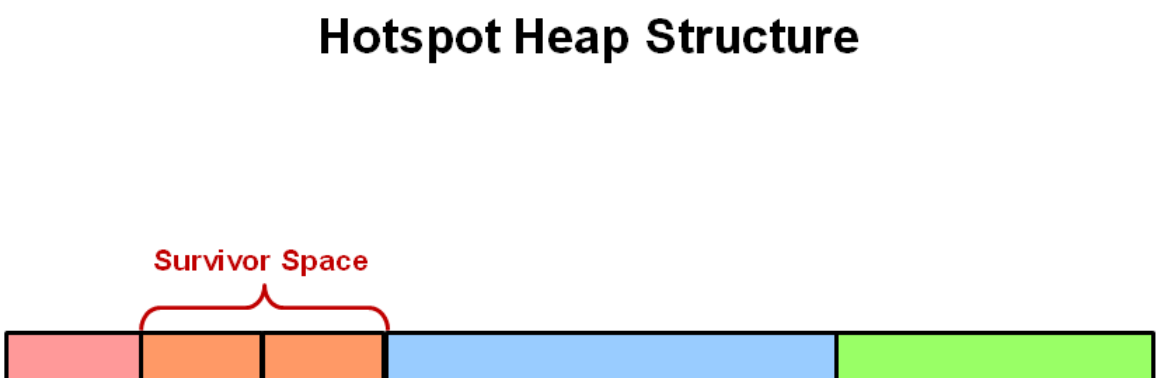


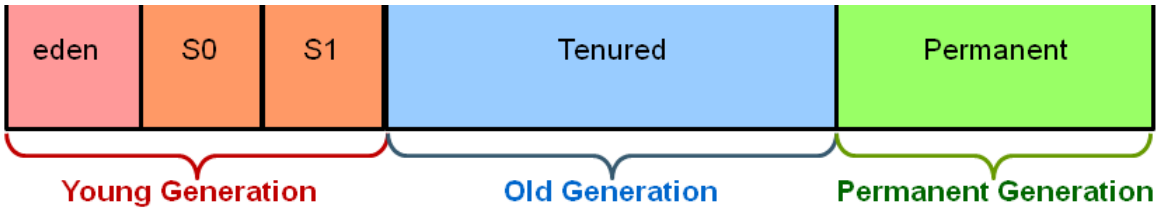
What is Automatic Garbage Collection?

> Automatic garbage collection is the process of looking at heap memory, identifying objects are in use and deleting the unreferenced objects. An unused/unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.
>
> In Java, process of deallocating memory is handled automatically by the garbage collector.

JVM Generations

The information learned from the object allocation behavior can be used to enhance the performance of the JVM. Therefore, the heap is broken up into smaller parts or generations. The heap parts are: Young Generation, Old or Tenured Generation, and Permanent Generation.





The **Young Generation** is where all new objects are allocated and aged. When the young generation fills up, this causes a ***minor garbage collection***. Minor collections can be optimized assuming a high object mortality rate. A young generation full of dead objects is collected very quickly. Some surviving objects are aged and eventually move to the old generation.

Stop the World Event — All minor garbage collections are "Stop the World" events. This means that all application threads are stopped until the operation completes. Minor garbage collections are *always* Stop the World events.

The **Old Generation** is used to store long surviving objects. Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation. Eventually the old generation needs to be collected. This event is called a ***major garbage collection***.

Major garbage collection are also **Stop the World events.** Often a major collection is much slower because it involves all live objects. So for Responsive applications, major garbage collections should be minimized. Also note, that the length of the Stop the World event for a major garbage collection is affected by the kind of garbage collector that is used for the old generation space.

The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

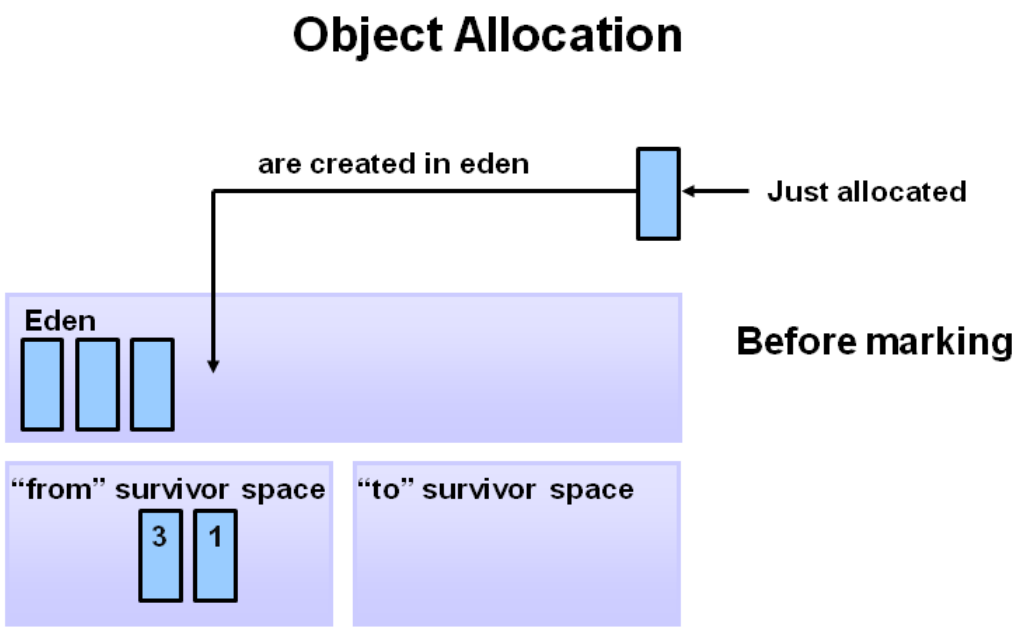
Classes may get collected (unloaded) if the JVM finds they are no longer needed and space may be needed for other classes. The permanent generation is included in a full garbage collection.

The Generational Garbage Collection Process

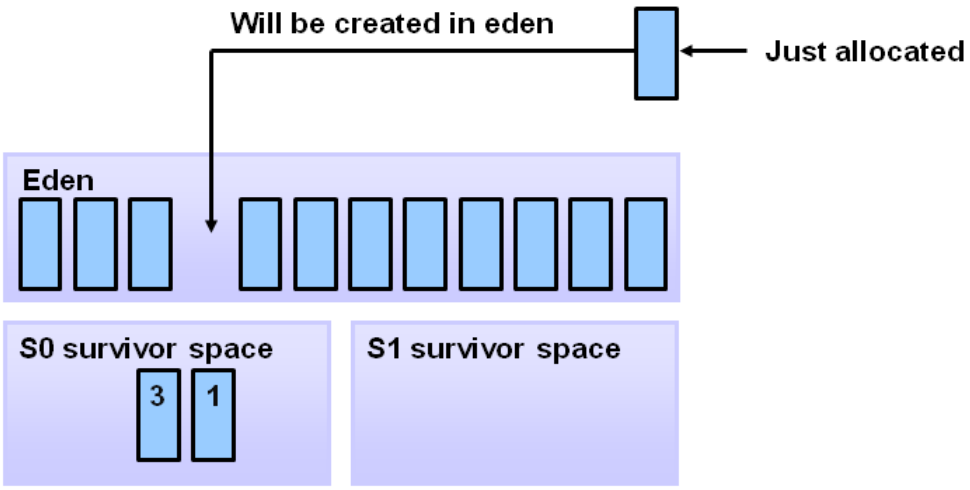
Minor GC

Now that you understand why the heap is separated into different generations, it is time to look at how exactly these spaces interact. The pictures that follow walks through the object allocation and aging process in the JVM.

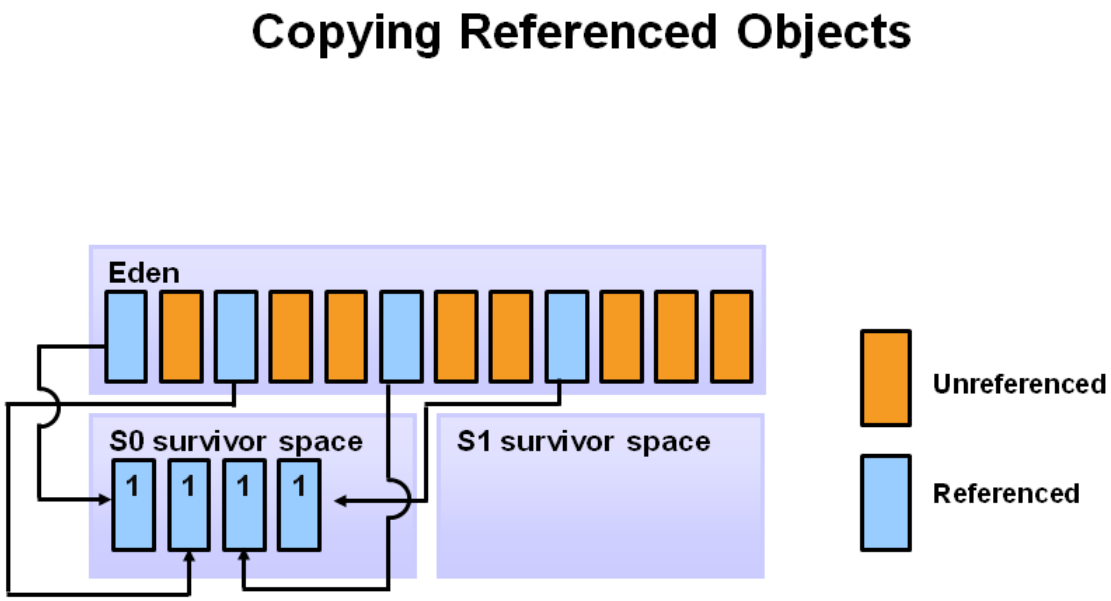
Step 1 : First, any new objects are allocated to the eden space. Both survivor spaces start out empty.



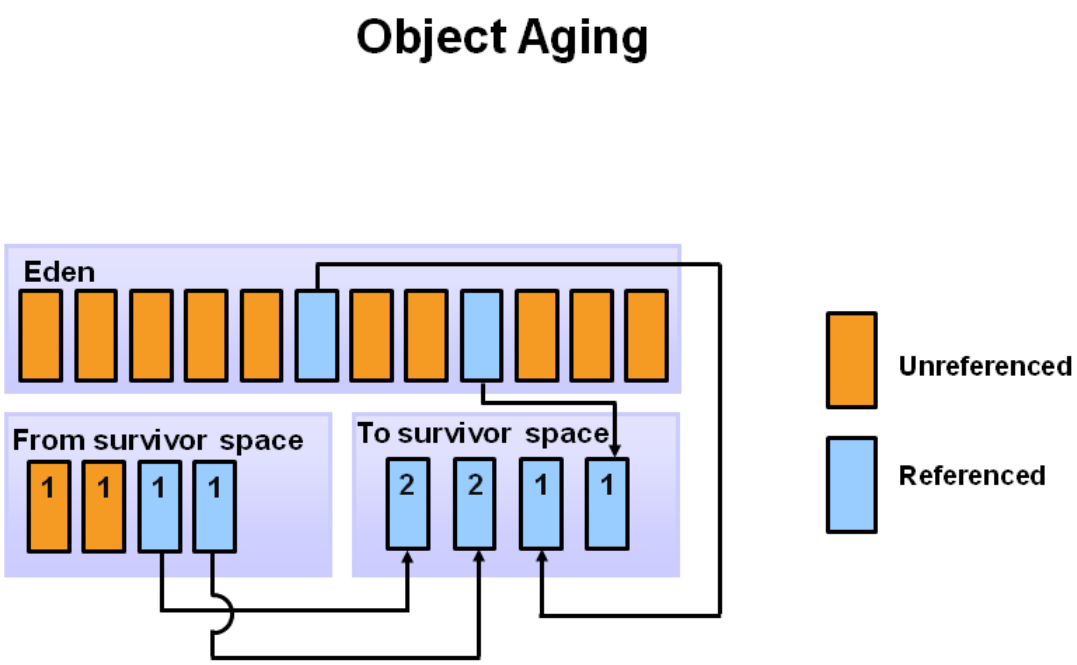
Step 2 : When the eden space fills up, a minor garbage collection is triggered.



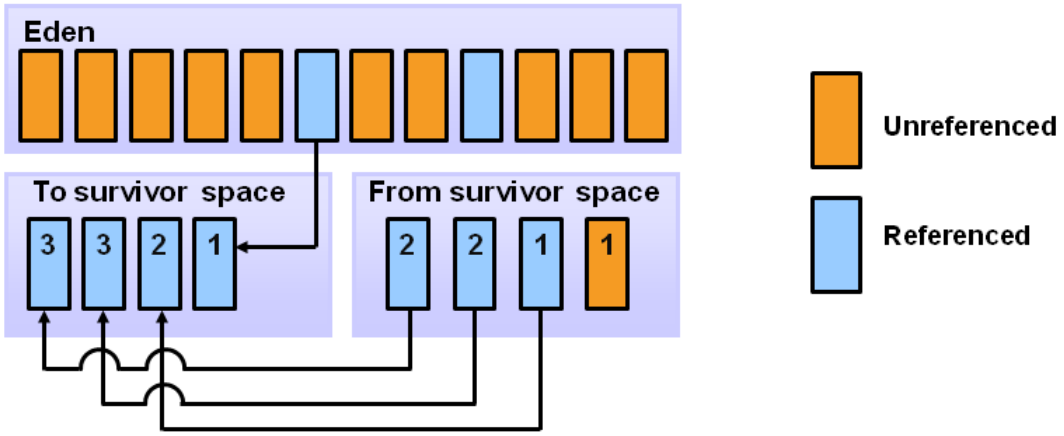
Step 3 : Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.



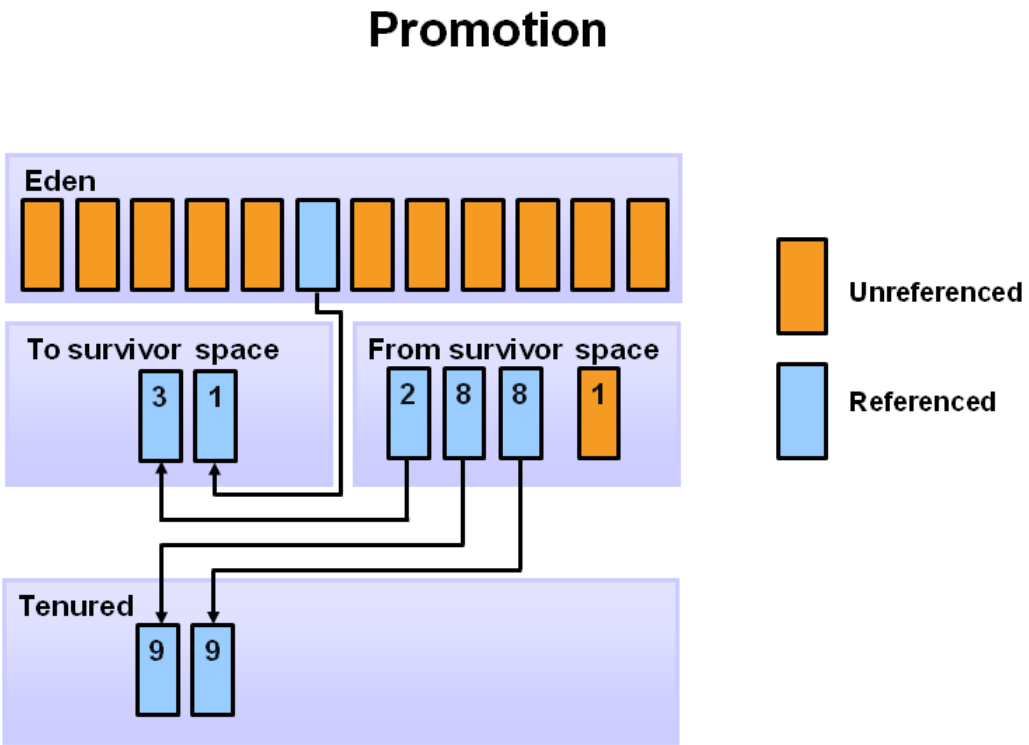
Step 4 : At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (S0) have their age incremented and get moved to S1. Once all surviving objects have been moved to S1, both S0 and eden are cleared. Notice we now have differently aged object in the survivor space.



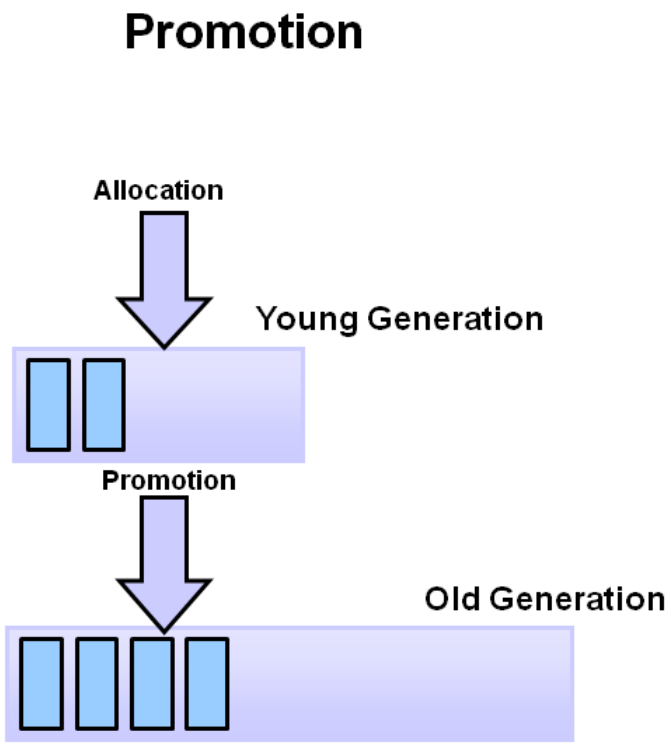
Step 5 : At the next minor GC, the same process repeats. However this time the survivor spaces switch. Referenced objects are moved to S0. Surviving objects are aged. Eden and S1 are cleared.



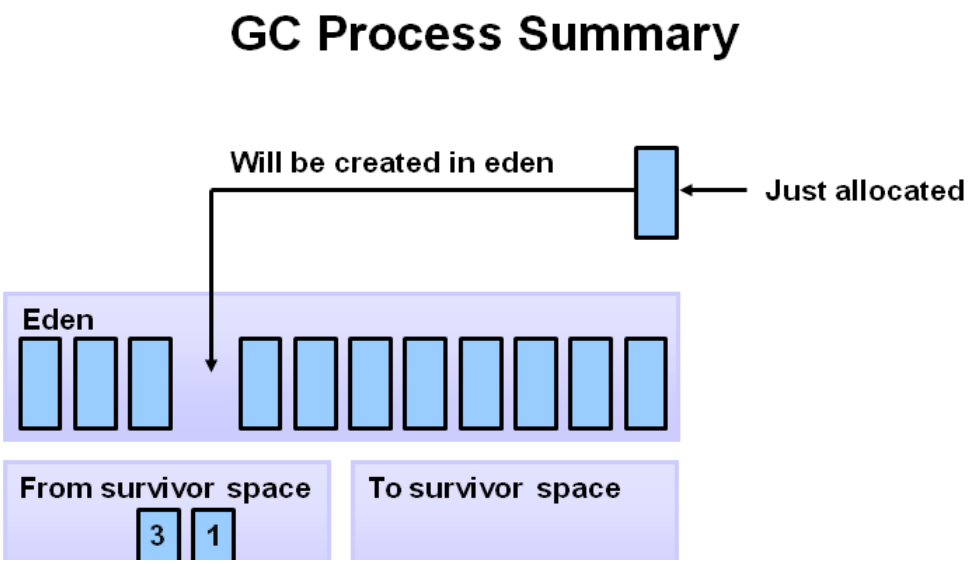
Step 6 : This slide demonstrates promotion. After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.

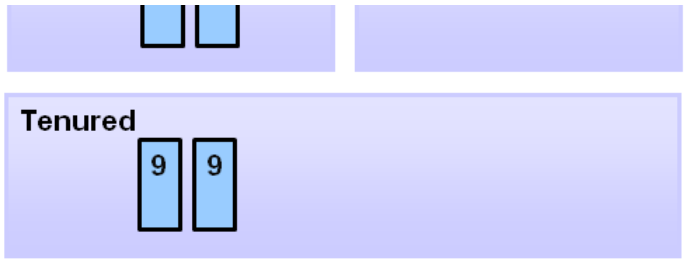


Step 7 : As minor GCs continue to occur objects will continue to be promoted to the old generation space.



Step 8 : So that pretty much covers the entire process with the young generation. Eventually, a major GC will be performed on the old generation which cleans up and compacts that space.

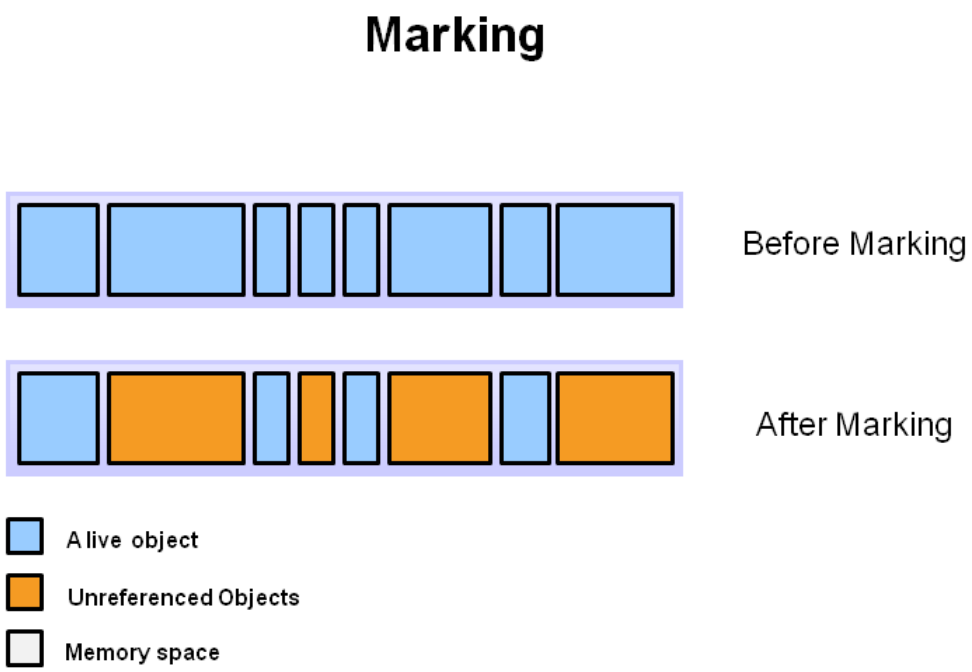




Marking Process

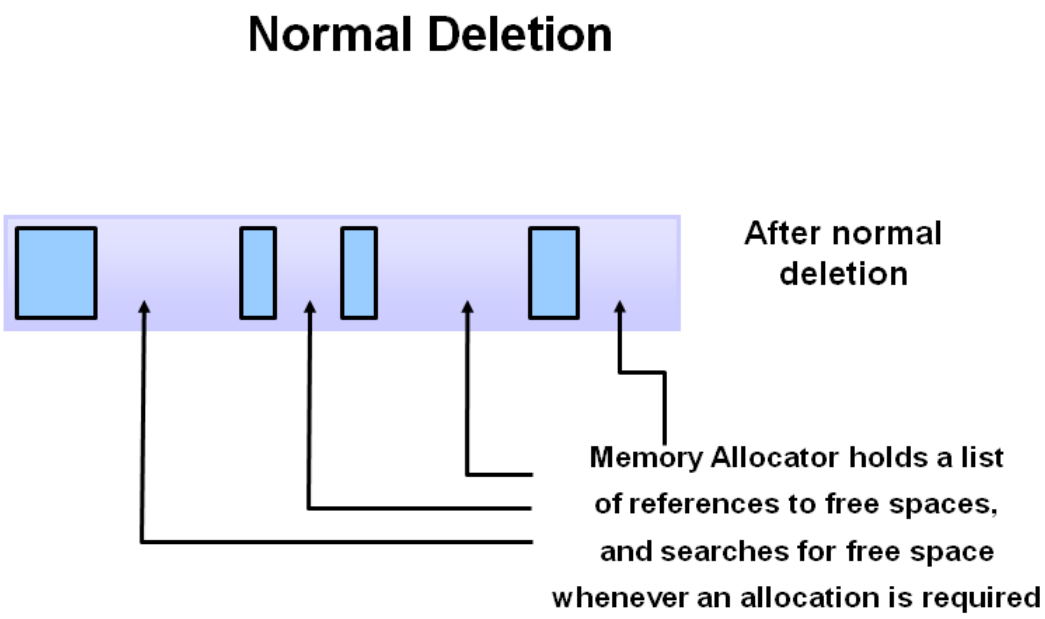
The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.

Referenced objects are shown in blue. Unreferenced objects are shown in gold. All objects are scanned in the marking phase to make this determination. This can be a very time consuming process if all objects in a system must be scanned.



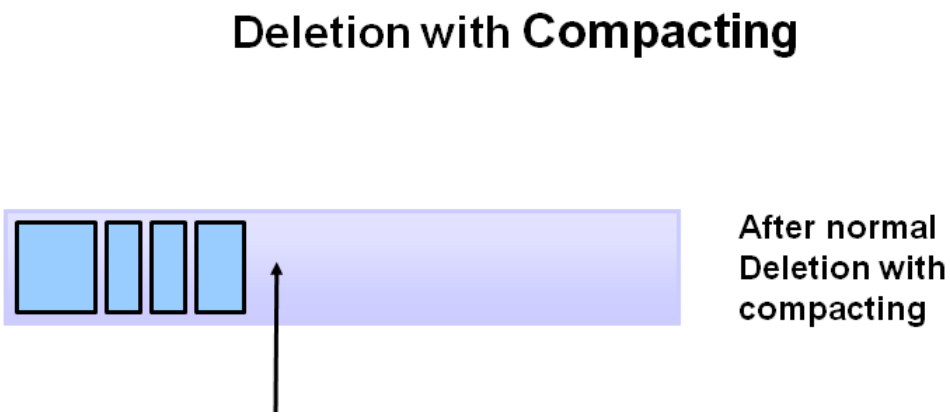
Deletion without Compacting(Minor GC on young gen)

Deletion removes unreferenced objects leaving referenced objects and pointers to free space.The memory allocator holds references to blocks of free space where new object can be allocated.



Deletion with Compacting(Major GC on Old gen)

To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.

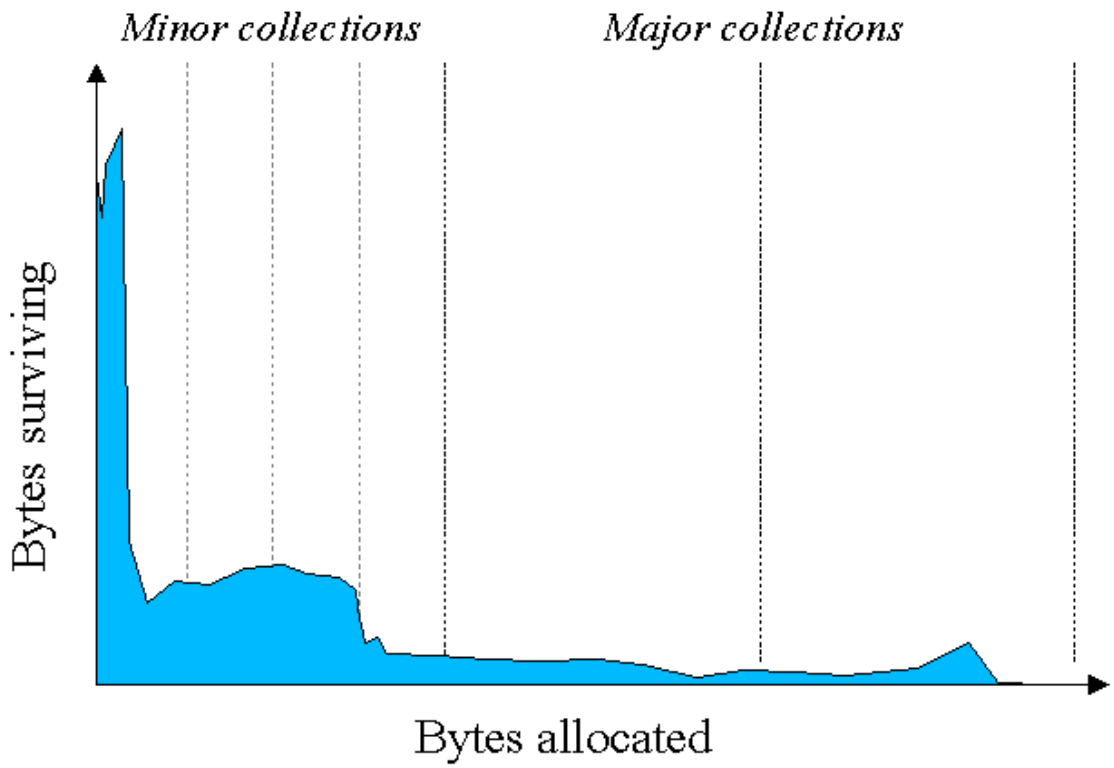


Memory Allocator holds the reference to the beginning of free space, and allocated memory sequentially then on.

Why Generational Garbage Collection?

As stated earlier, having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short lived.

Here is an example of such data. The Y axis shows the number of bytes allocated and the X access shows the number of bytes allocated over time.



As you can see, fewer and fewer objects remain allocated over time. In fact most objects have a very short life as shown by the higher values on the left side of the graph.

Summary

In this Article, you have been given an overview of garbage collection system on the Java JVM. First you learned how the Heap and the Garbage Collector are key parts of any Java JVM. Automatic garbage collection is accomplished using generational garbage collection approach.

In this tutorial, you have learned:

- The components of the Java JVM
- How automatic garbage collection works
- The generational garbage collection process