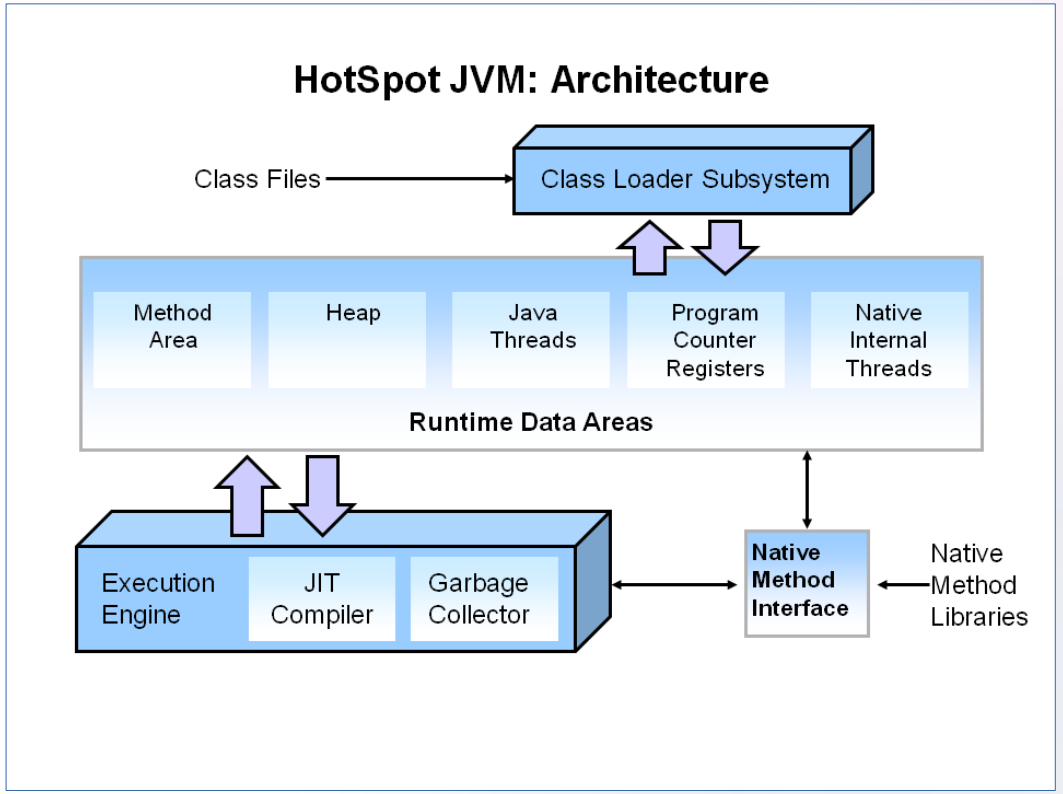
Session 6 : Garbage Collection

Java Programming Language

- Platform Independence
- Object-Oriented
- **Automatic Garbage Collection** - Java automatically allocates and deallocates memory so programs are not burdened with that task.
- Rich Standard Library

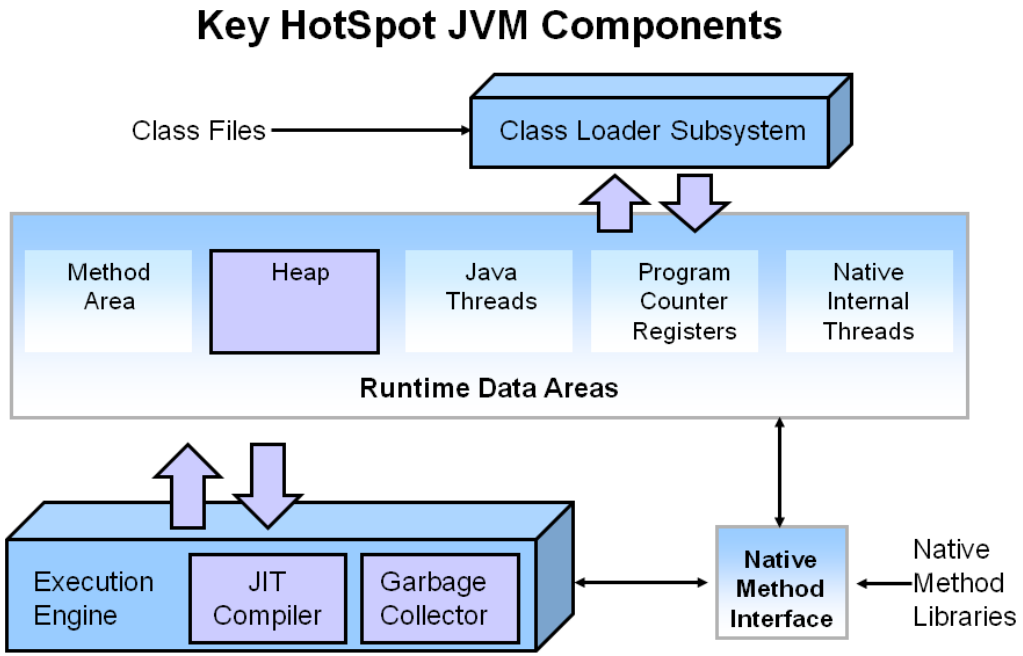
JVM Architecture

- The main components of the JVM include **the classloader, the runtime data areas, and the execution engine.**



Performance Components

- The key components of the JVM that relate to performance are highlighted in the following image.



- The heap is where your object data is stored. It is managed by the garbage collector selected at startup. Most tuning options relate to sizing the heap and choosing the most appropriate garbage collector for your situation. The JIT compiler also has a big impact on performance but rarely requires tuning with the newer versions of the JVM.

Performance Basics

- when tuning a Java application, the focus is on one of two main goals: **responsiveness or throughput.**

Performance Basics: Responsiveness

Responsiveness refers to how quickly an application or system responds with a requested piece of data. For examples:

- How quickly a desktop UI responds to an event
- How fast a website returns a page
- How fast a database query is returned

For applications that focus on responsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

Performance Basics: Throughput

Throughput focuses on maximizing the amount of work by an application in a specific period of time. For examples:

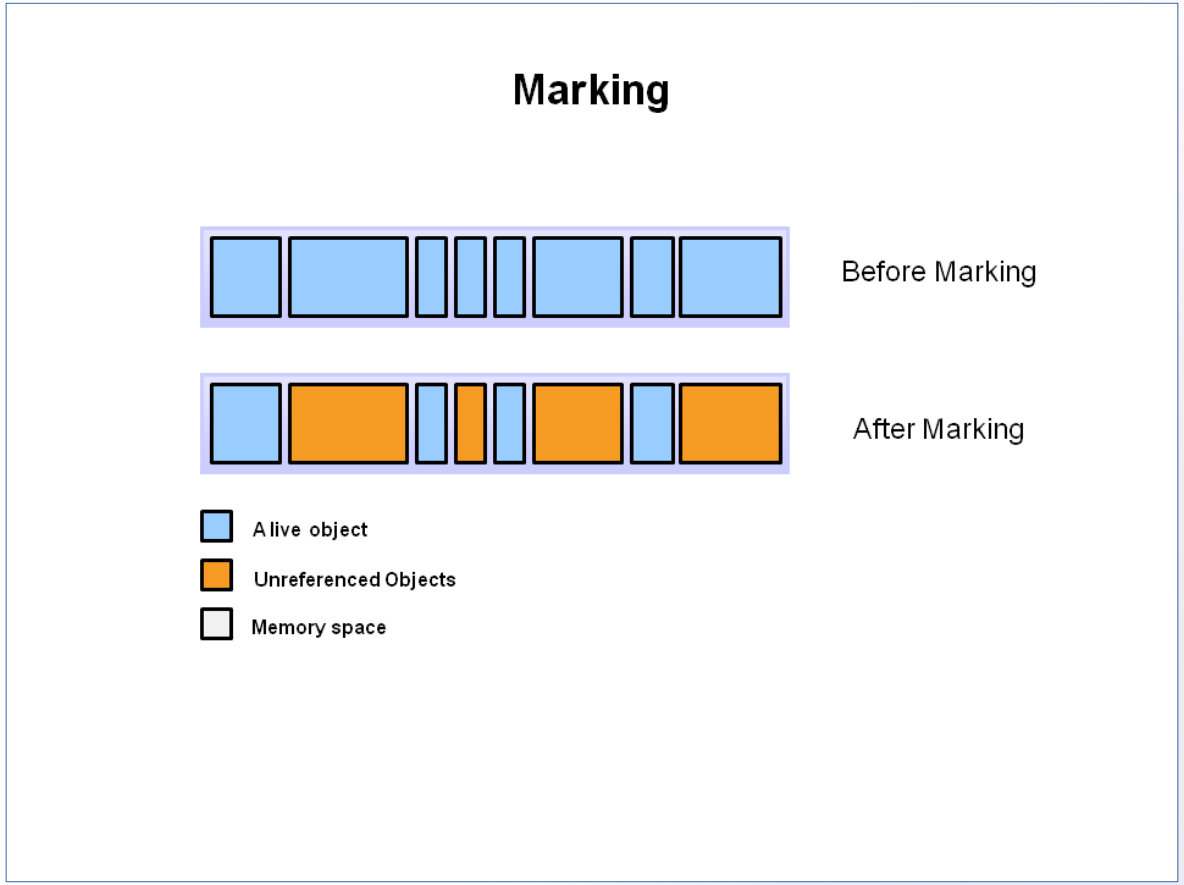
- The number of transactions completed in a given time.
- The number of jobs that a batch program can complete in an hour.
- The number of database queries that can be completed in an hour.

High pause times are acceptable for applications that focus on throughput. Since high throughput applications focus on benchmarks over longer periods of time, quick response time is not a consideration.

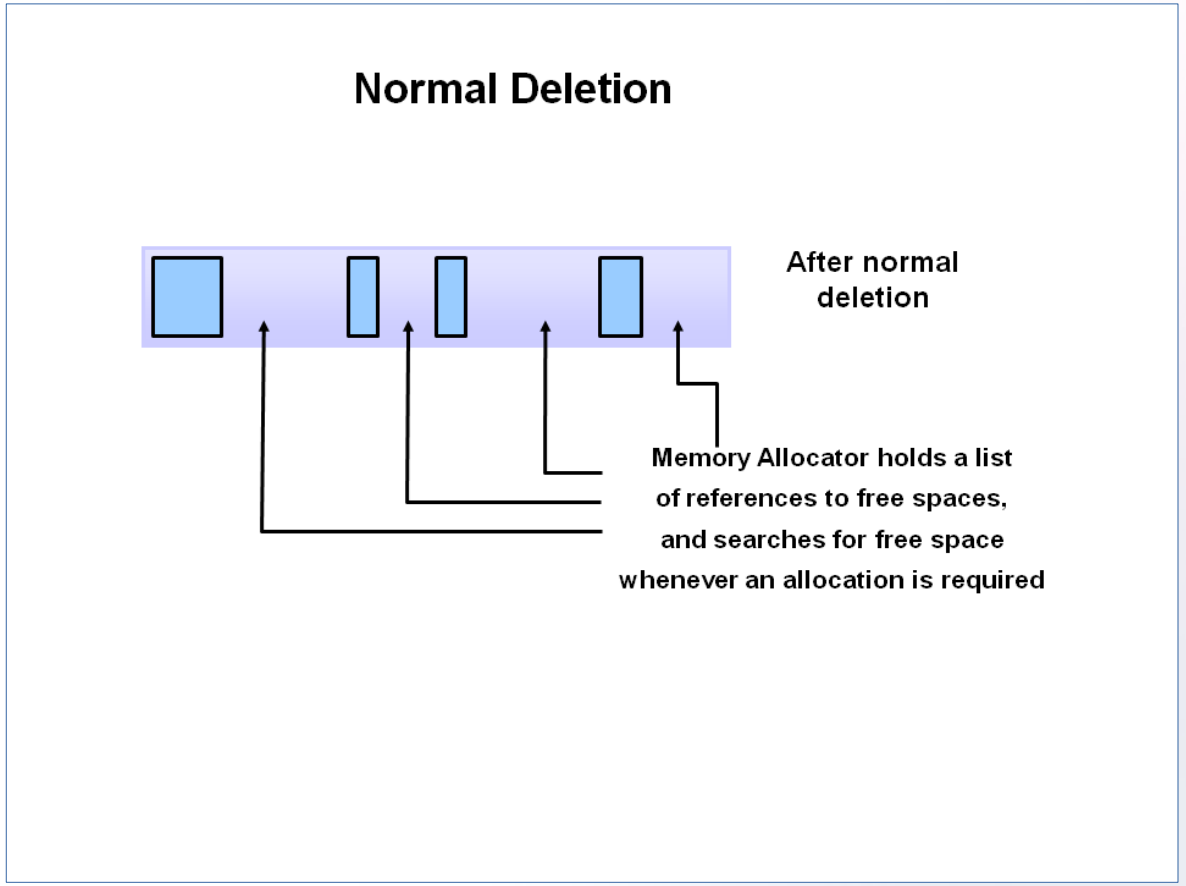
What is Automatic Garbage Collection?

- Automatic garbage collection is the process of looking at **heap memory**, identifying which **objects are in use** and **which are not**, and deleting the unused objects.
- An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

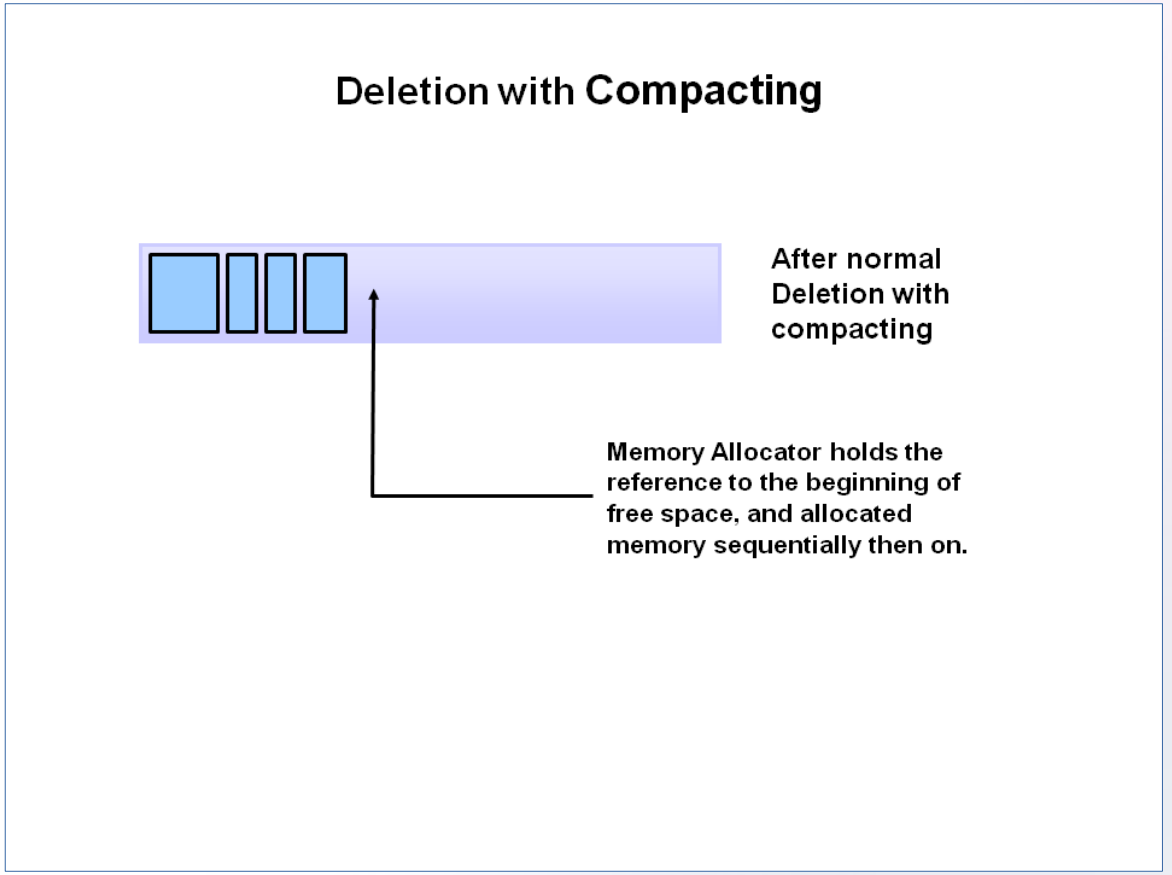
Basic process : Marking



Basic process : Normal Deletion



Basic process : Deletion with Compacting

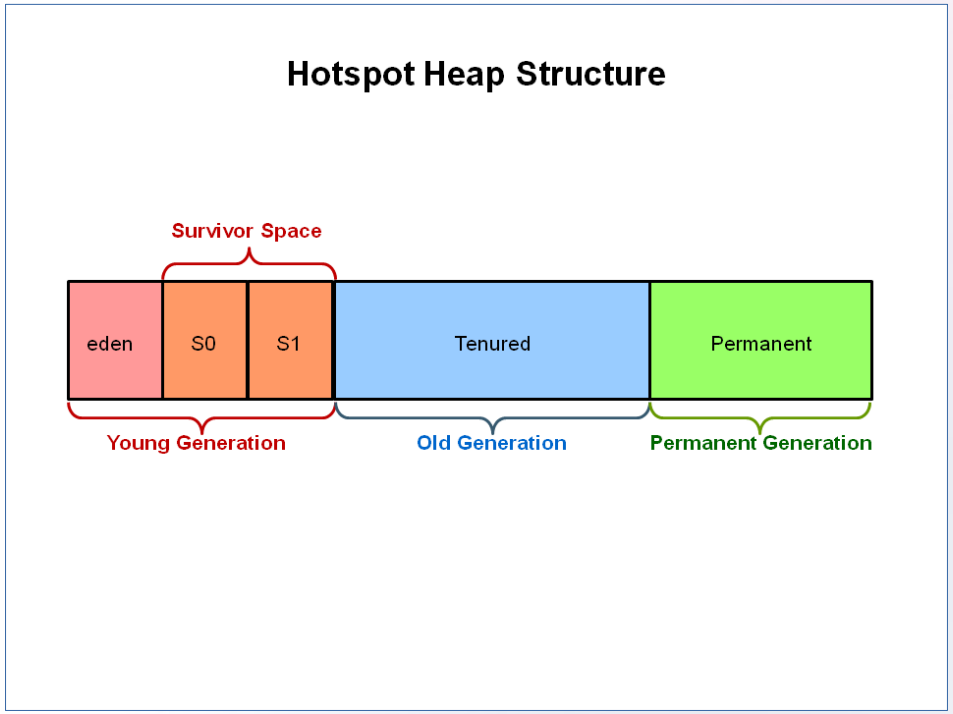


Generational Garbage Collection

- 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.
- Empirical analysis of applications has shown that most objects are short lived.

JVM Generations

- Heap is broken up into smaller parts or generations.
- The heap parts are: **Young Generation, Old or Tenured Generation, and Permanent Generation**



Young 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**.
- **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

Old Generation

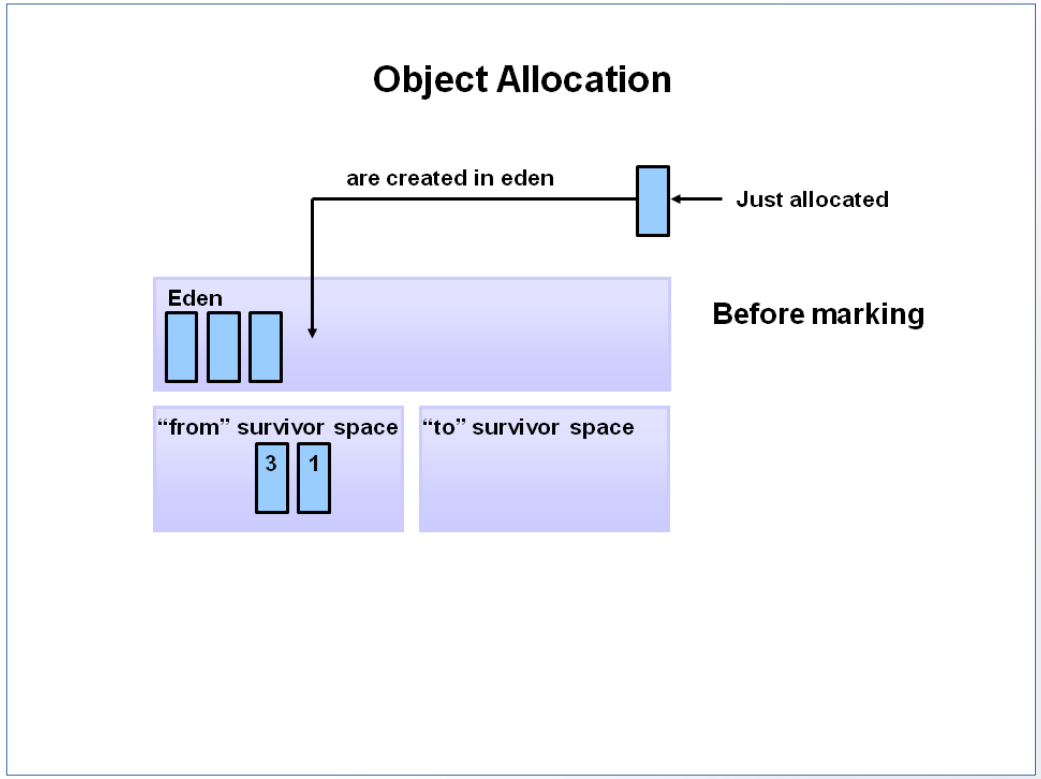
- The **Old Generation** is used to store long surviving objects.
- Typically, a **threshold** is set for young generation object and when that age is met, the object gets moved to the old generation.
- Eventually the old generation needs to be collected. This event is called a **major garbage collection**.

Permanent generation

- 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.

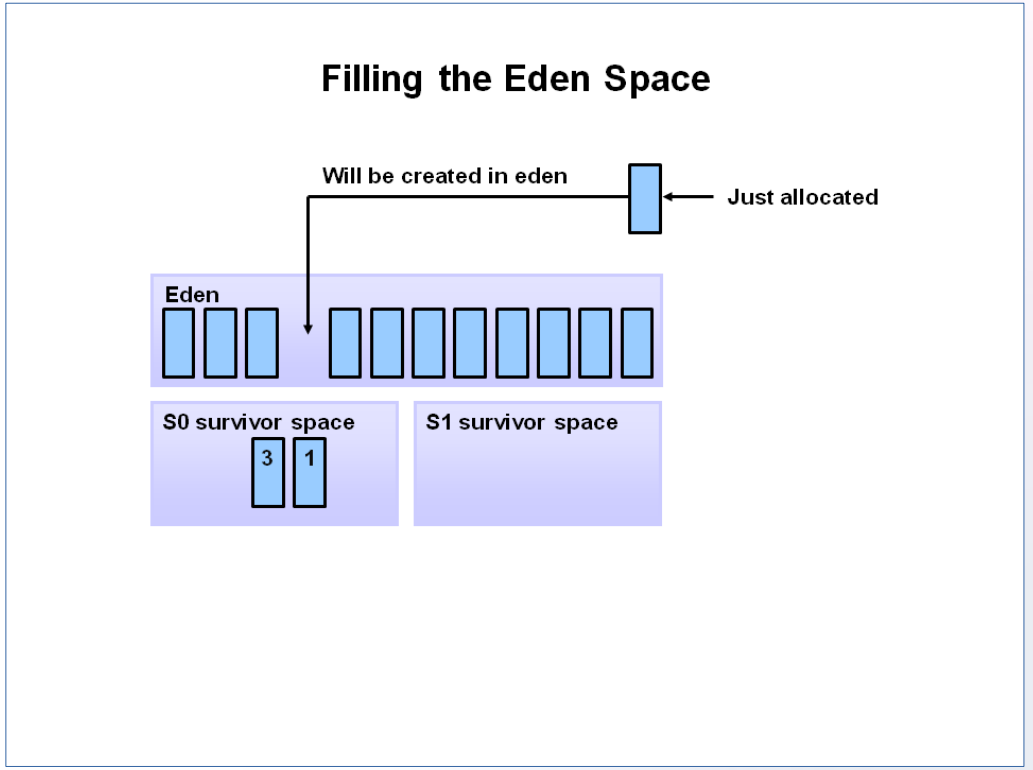
Generational Garbage Collection Process

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



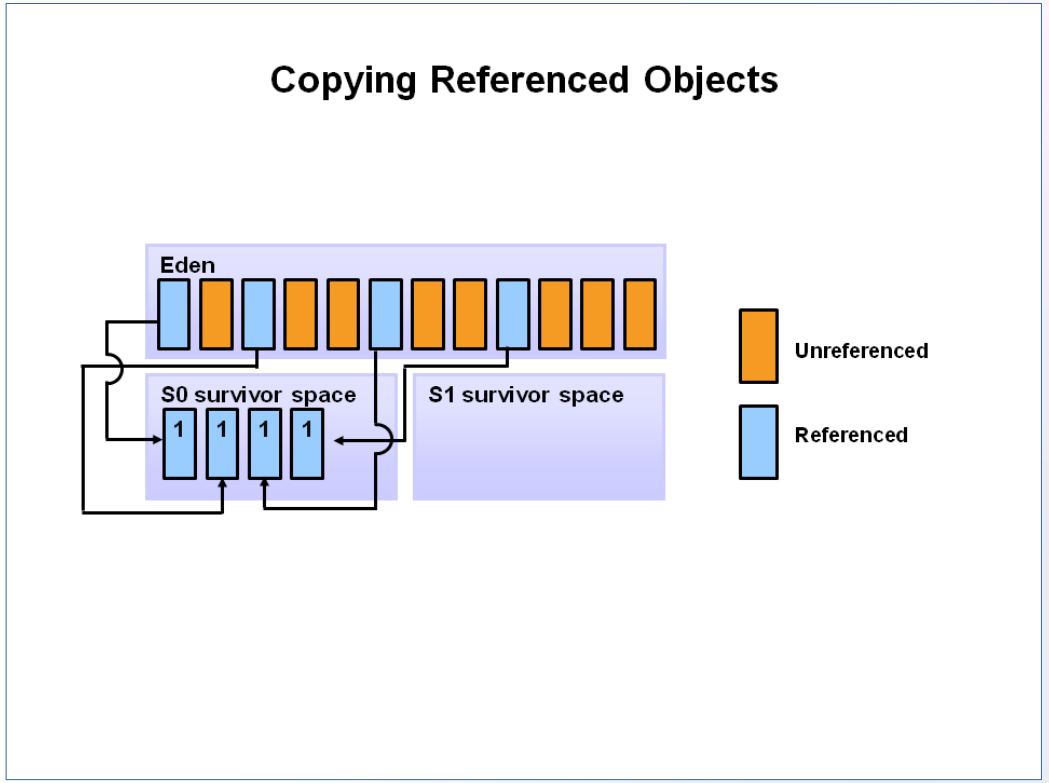
Generational Garbage Collection Process

(2) When the eden space fills up, a minor garbage collection is triggered.



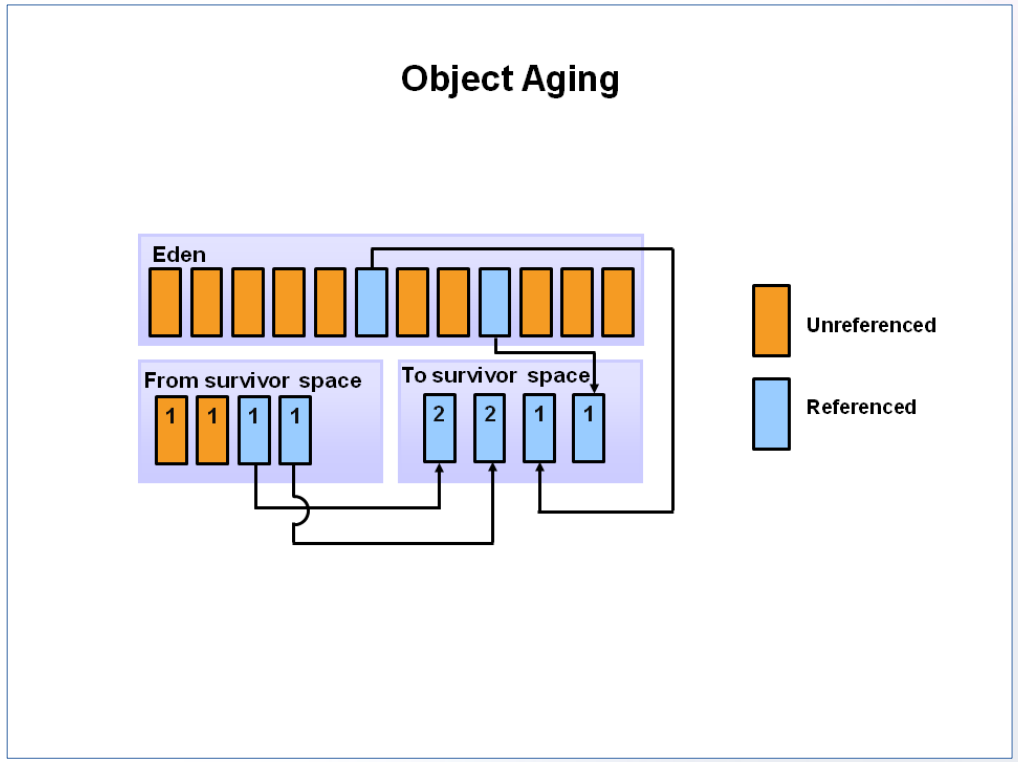
Generational Garbage Collection Process

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



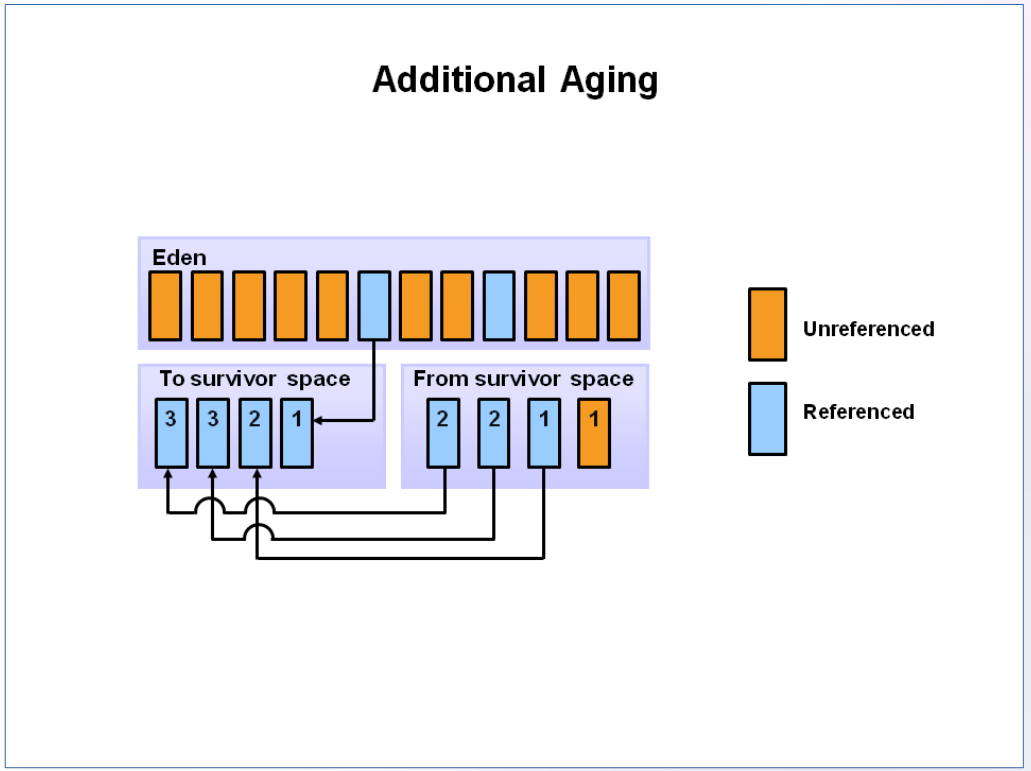
Generational Garbage Collection Process

(4) At the next minor GC, the same thing happens for the eden space. However, Unreferenced objects are moved to the second survivor space (S1). Objects 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.



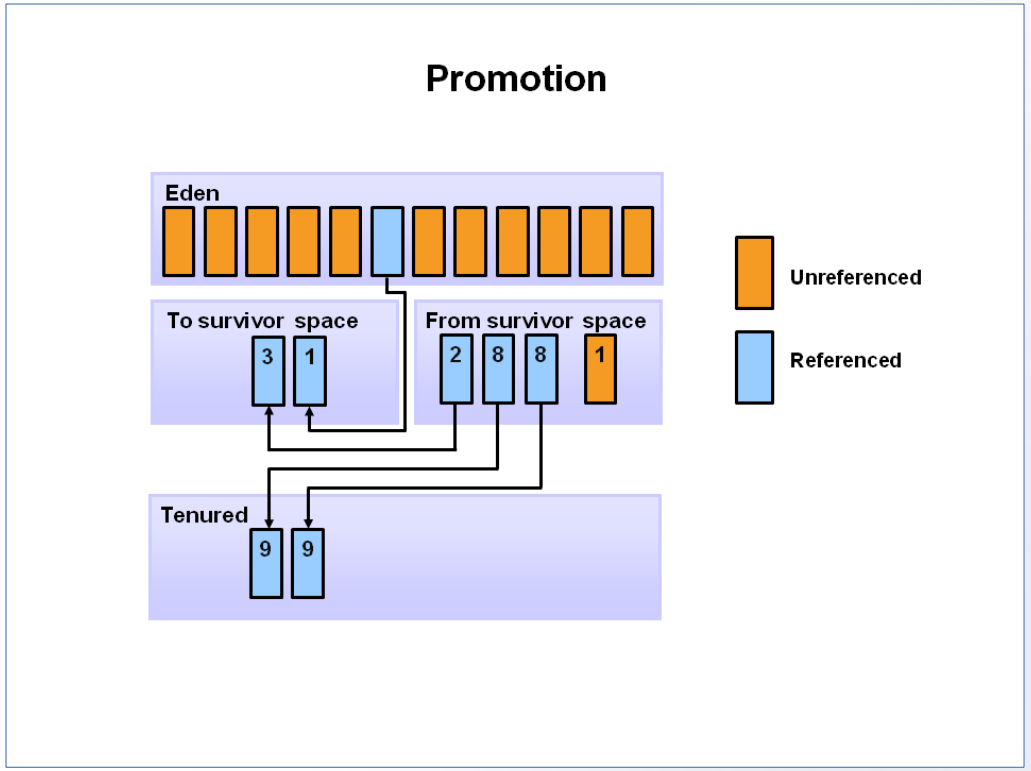
Generational Garbage Collection Process

(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.



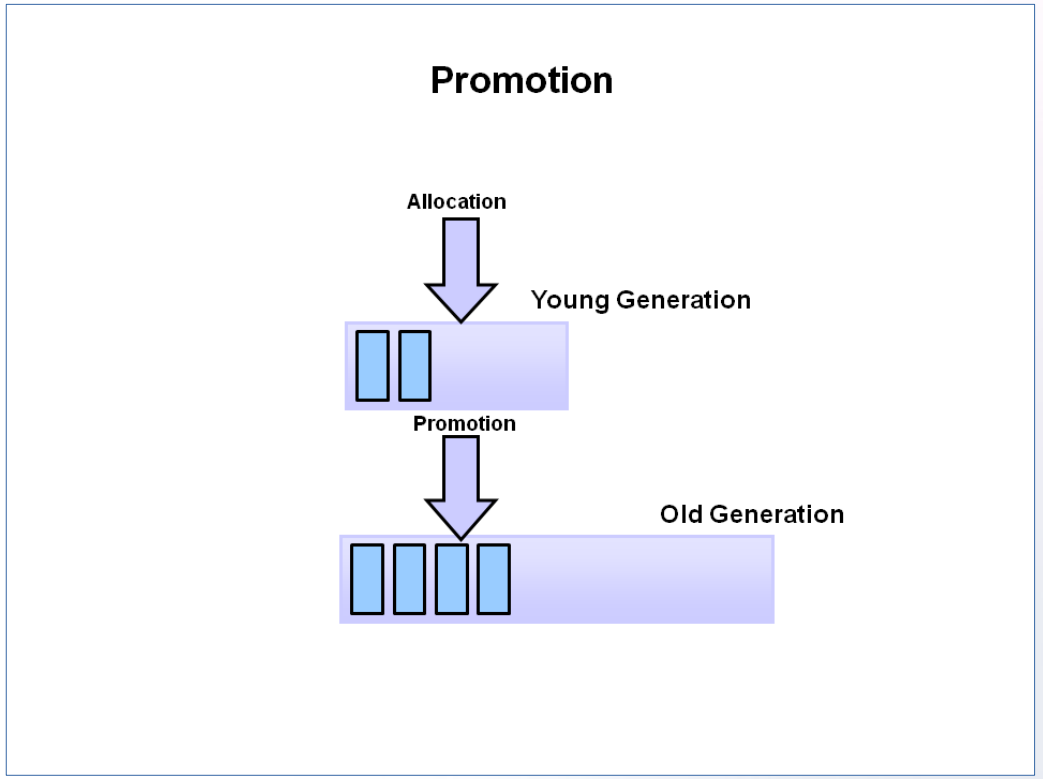
Generational Garbage Collection Process

(6) 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.



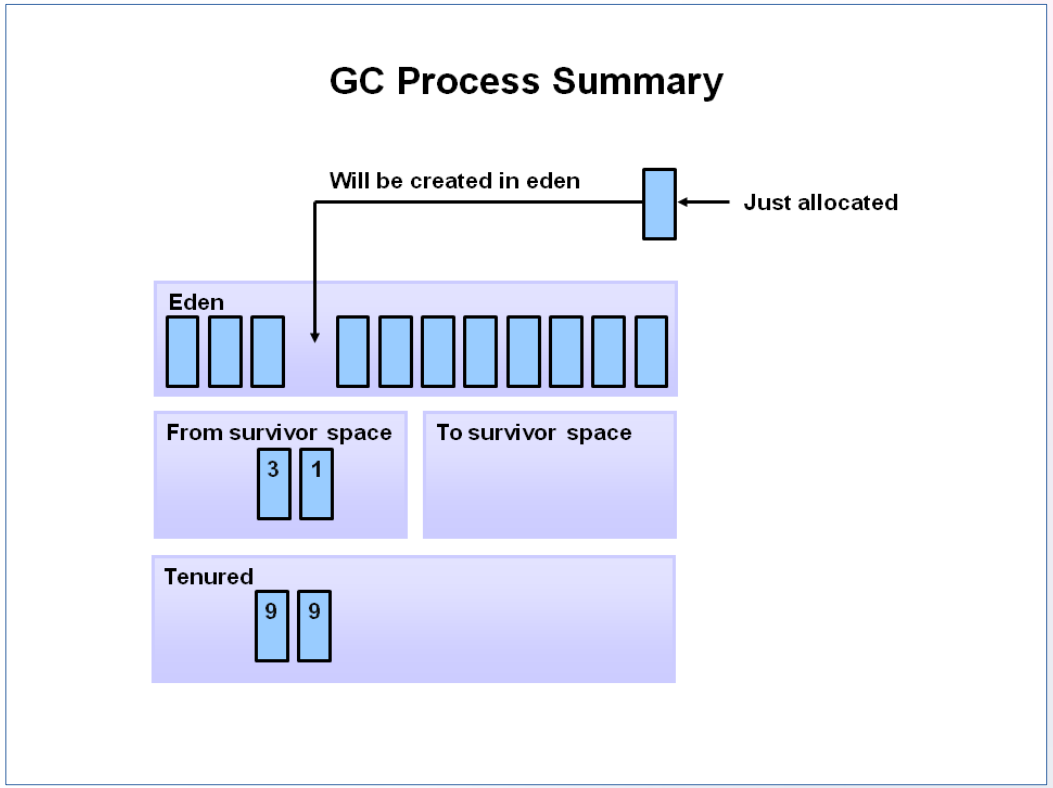
Generational Garbage Collection Process

(7) As minor GCs continue to occure objects will continue to be promoted to the old generation space.



Generational Garbage Collection Process

(8)



Java Garbage Collectors

- command line switches

Switch	Description
-Xms	Sets the initial heap size for when the JVM starts.
-Xmx	Sets the maximum heap size.
-Xmn	Sets the size of the Young Generation.
-XX:PermSize	Sets the starting size of the Permanent Generation.
-XX:MaxPermSize	Sets the maximum size of the Permanent Generation

Serial GC

- With the serial collector, both minor and major garbage collections are done serially (using a single virtual CPU). In addition, it uses a mark-compact collection method. This method moves older memory to the beginning of the heap so that new memory allocations are made into a single continuous chunk of memory at the end of the heap. This compacting of memory makes it faster to allocate new chunks of memory to the heap.

Serial GC : Usage Cases

- The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work (therefore, its name).
- Usuful in environments where a high number of JVMs are run on the same machine (in some cases, more JVMs than available processors!).
- Good for embedded hardware with minimal memory and few cores

Serial GC : command option

To enable the Serial Collector use:

-XX:+UseSerialGC

Parallel GC

- The parallel garbage collector uses multiple threads to perform the young genertion garbage collection. By default on a host with N CPUs, the parallel garbage collector uses N garbage collector threads in the collection.

-XX:ParallelGCThreads=<desired number> : limit threads

- On a host with a single CPU the default garbage collector is used even if the parallel garbage collector has been requested.

Parallel GC : Usage Cases

- The Parallel collector is also called a throughput collector
- This collector should be used when a lot of work need to be done and long pauses are acceptable. For example, batch processing like printing reports or bills or performing a large number of database queries.

Parallel GC : Command options

- **-XX:+UseParallelGC**

Multi-thread young generation collector with a single-threaded old generation collector.

Single-threaded compaction of old generation.

-XX:+UseParallelOldGC

Multithreaded young generation collector and multithreaded old generation collector.

Multithreaded compacting collector.

Concurrent Mark Sweep (CMS) Collector

- The Concurrent Mark Sweep (CMS) collector (also referred to as the concurrent low pause collector) attempts to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.
- Does not copy or compact the live objects

CMS Collector : Usage Cases

- The CMS collector should be used for applications that require low pause times and can share resources with the garbage collector.
- Examples include desktop UI application that respond to events, a webserver responding to a request or a database responding to queries.

CMS Collector : command option

To enable the CMS Collector use:

-XX:+UseConcMarkSweepGC

set the number of threads use:

-XX:ParallelCMSThreads=<n>

G1 Garbage Collector

- The Garbage First or G1 garbage collector is designed to be the long term replacement for the CMS collector.
- The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

G1 Garbage Collector : command option

Enable the G1 Collector:

-XX:+UseG1GC

OutofMemoryError

- An **OutofMemoryError** typically occurs when an application or program tries to create new objects, but the JVM is unable to allocate memory to accommodate them.
- When an **OutofMemoryError** occurs, the application will usually crash and terminate. This error is common in programs that deal with large amounts of metadata, such as an image or video processing applications, or programs that handle large databases.

Garbage collection events trigger

- **Heap space allocation**: When the JVM needs to allocate memory for a new object, and there is not enough space in the heap
- **System.gc()**: You can explicitly request garbage collection by calling the System.gc()
- **Old generation threshold**: Garbage collection can also be triggered when the heap size of the old generation heap space (which stores long-lived objects) reaches a certain threshold.
- Etc...

End

sourcemind