**Garbage Collection**

**Q 1) 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.**

In a programming language like C, allocating and deallocating memory is a manual process. In Java, process of deallocating memory is handled automatically by the garbage collector.

## Q 2) How can an object be unreferenced?

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By annonymous object etc.

1) By nulling a reference:

Employee e=**new** Employee();

e=**null**;

2) By assigning a reference to another:

Employee e1=**new** Employee();

Employee e2=**new** Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection

3) By annonymous object:

**new** Employee();

## 4)Island of isolation?

## Q 3) finalize() method ?

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

**protected** **void** finalize(){}

## Q 4) gc() method ?

## We cannot force Garbage collection.

## gc() method gives request to garbage collector to perform cleanup processing

The gc() is found in System and Runtime classes.

**public** **static** **void** gc(){}

#### Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

Q 4 b)What is the difference between System.gc() and Runtime.getRuntime().gc();?

Both are same. System.gc() is effectively equivalent to Runtime.getRuntime().gc();

 System.gc()internally calls Runtime.getRuntime().gc();

.

The only difference is System.gc() is a class method where as Runtime.getRuntime().gc();

 is an instance method. So, System.gc() is more convenient.

### Q 5) Simple Example of garbage collection in java

class Demofinalize

{

int i;

Demofinalize (int x)

{

i=x;

}

public void finalize()

{

System.out.println("In Finalize method" );

System.out.println(i);

}

public static void main(String[] args)

{

Demofinalize obj1;

obj1=new Demofinalize (10);

obj1=new Demofinalize (20);

System.gc(); // request to gc

}

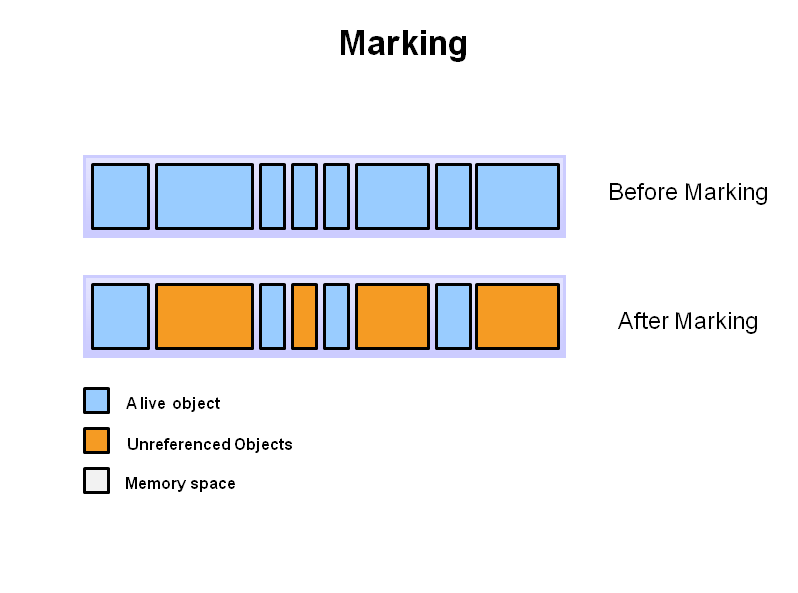
}

**Q 6) The basic process (steps) for Automatic Garbage collection?**

. The basic process can be described as follows.

#### Step 1: Marking

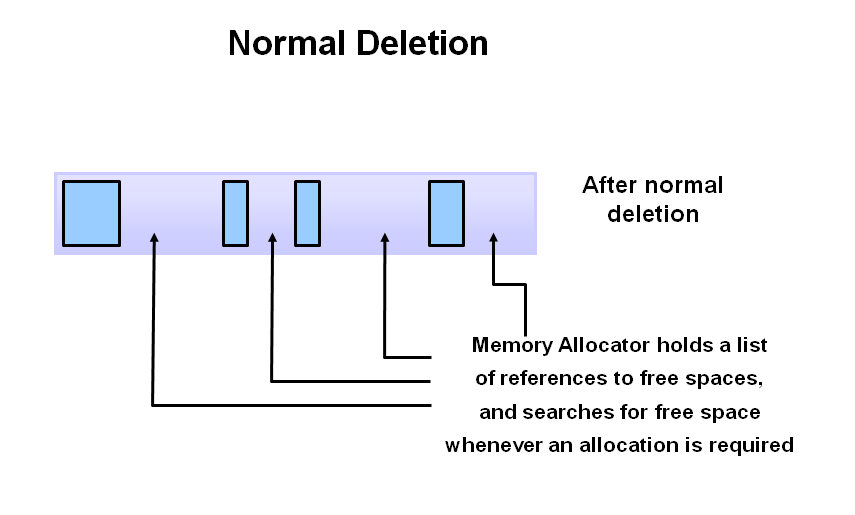
The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.



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

#### Step 2: Normal Deletion

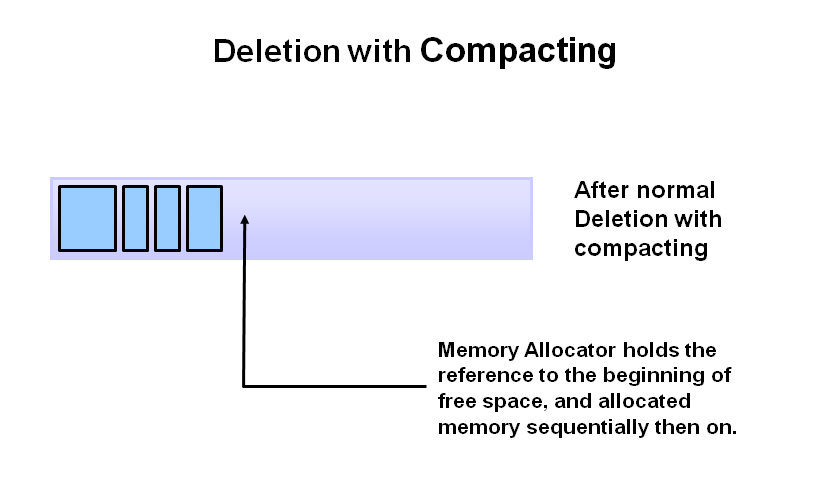
Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space



The memory allocator holds references to blocks of free space where new object can be allocated

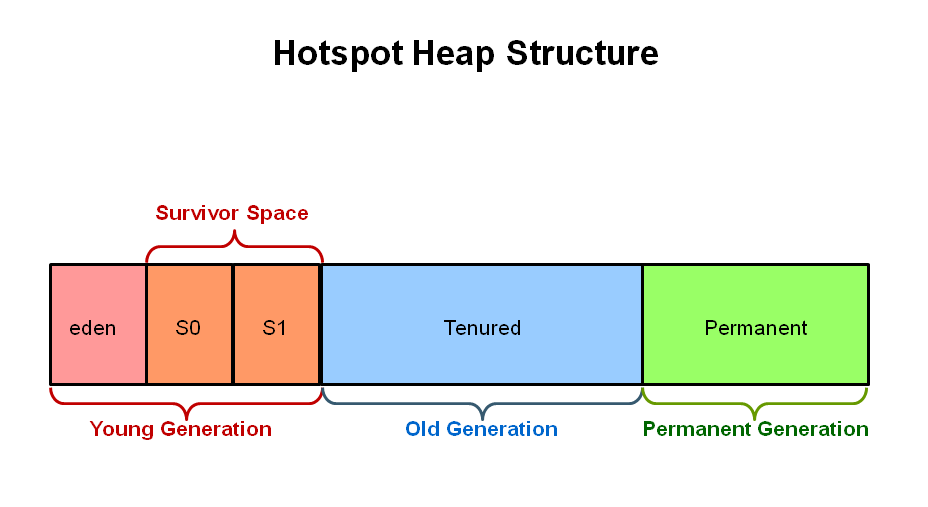
#### Step 2a: Deletion with Compacting

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.



**Q 7) JVM Generations?**

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.

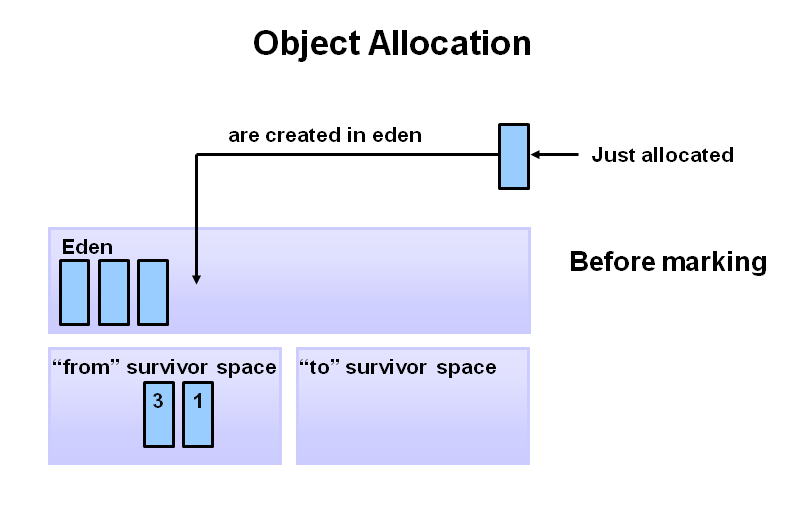
The **Permanent generation** contains metadata required by the JVM to describe the classes and methods used in the application. The permanent generation is populated by the JVM at runtime based on classes in use by the application. In addition, Java SE library classes and methods may be stored here.

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**

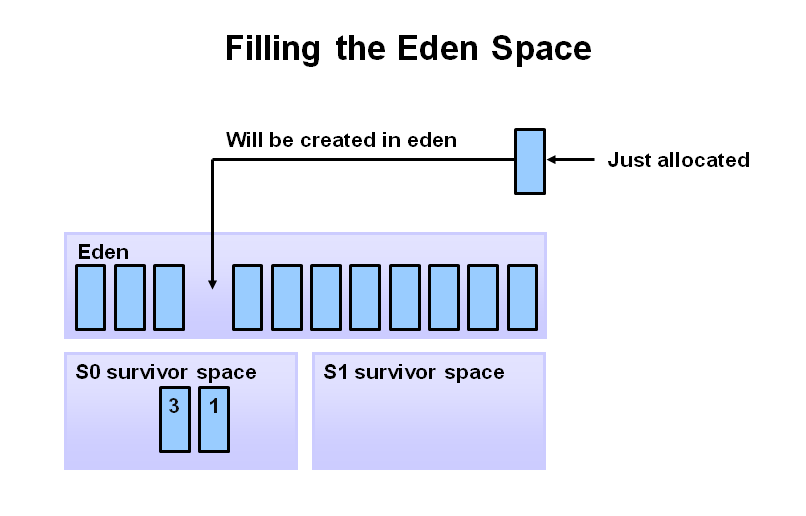
**Q 8) The Generational Garbage Collection Process**

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.

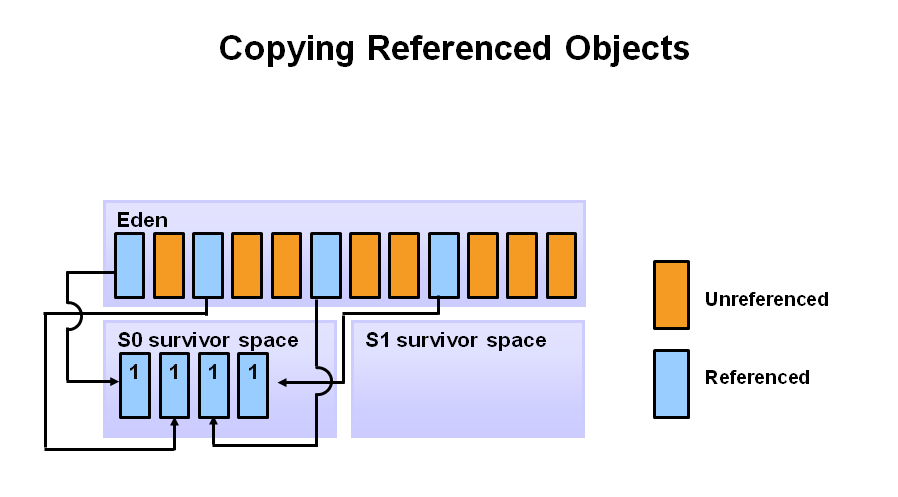
1. **First, any new objects are allocated to the eden space.** Both survivor spaces start out empty.

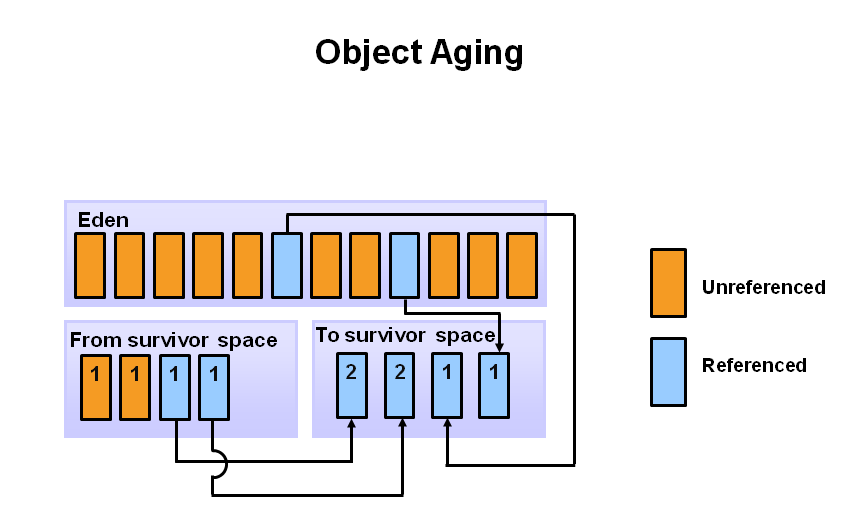
****

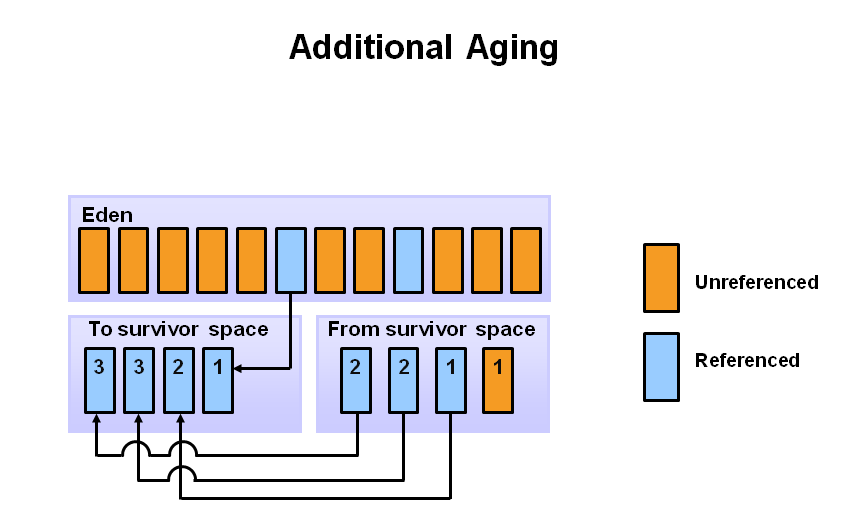
1. **When the eden space fills up, a minor garbage collection is triggered**



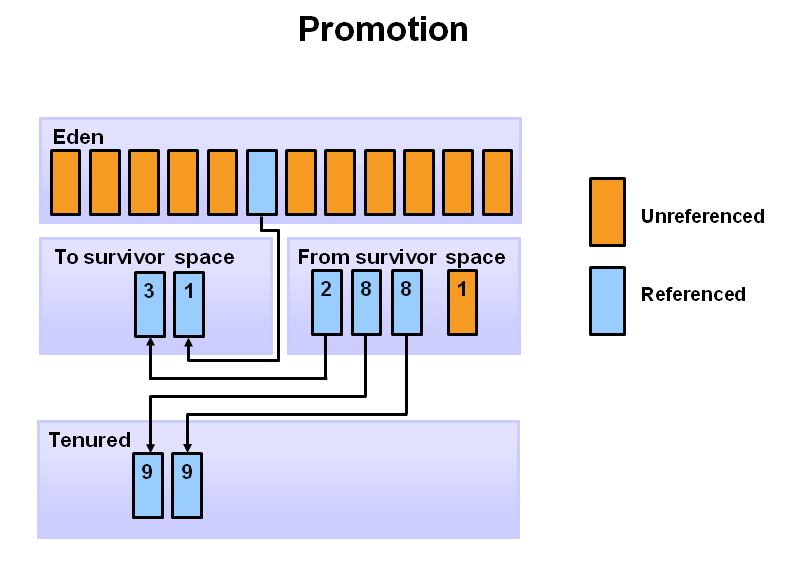
1. **Referenced objects are moved to the first survivor space. Unreferenced objects are deleted when the eden space is cleared.**



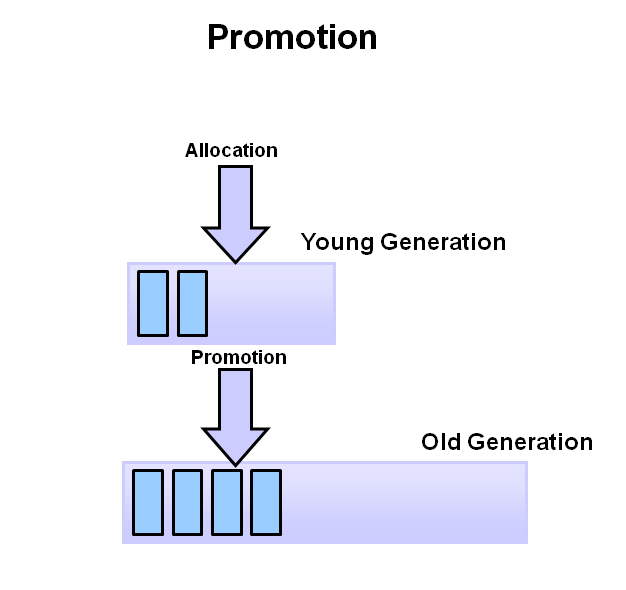
1. **At the next minor GC, the same thing happens for the eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1). In addition, objects from the last minor GC on the first survivor space (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. **
2. **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.**

****

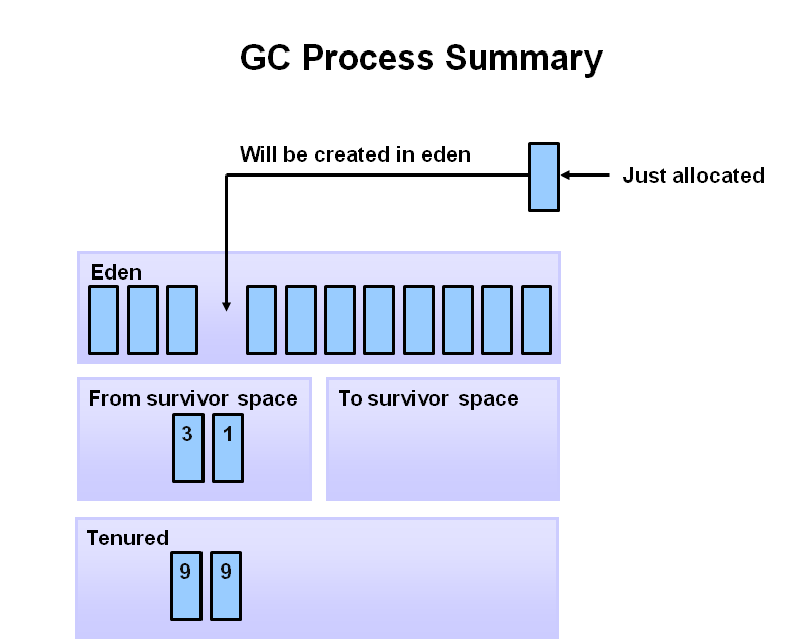
1. **After a minor GC, when aged objects reach a certain age threshold (8 in this example) they are promoted from young generation to old generation.**

****

1. **As minor GCs continue to occur objects will continue to be promoted to the old generation space**

****

1. **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**

****

**Q 9)Why 2 Survivor spaces in young Generation?**

**As an when object movement happen it is contiguous manner.**

**To avoid another run of compacting step which is expensive.**

**Q 10) Performance Basics?**

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

#### Responsiveness

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

* + 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 respsonsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

#### Throughput

Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include:

* + 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.

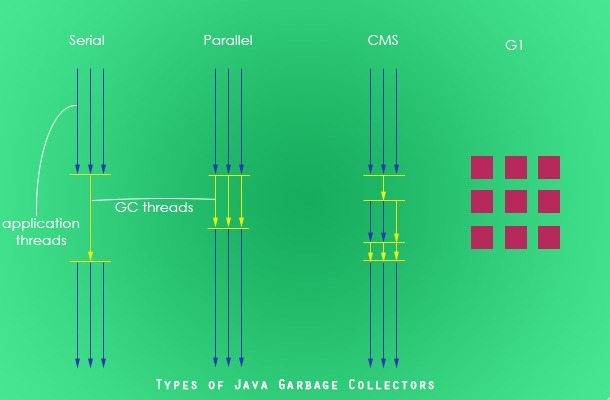
High pause times are acceptable for applications that focus on throughput.

**Q 11) Garbage Collectors**

Java has **four types of garbage collectors**,

1. [Serial Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#serial-garbage-collector)
2. [Parallel Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#parallel-garbage-collector)
3. [CMS Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#cms-garbage-collector)
4. [G1 Garbage Collector](https://javapapers.com/java/types-of-java-garbage-collectors/#g1-garbage-collector)

Each of these four types has its own advantages and disadvantages. Most importantly, we the programmers can choose the type of garbage collector to be used by the JVM. We can choose them by passing the choice as JVM argument. Each of these types differ largely and can provide completely different application performance. It is critical to understand each of these types of garbage collectors and use it rightly based on the application.



**`**

## 1) Serial Garbage Collector

Serial garbage collector works by holding all the application threads. It is designed for the single-threaded environments. It uses just a single thread for garbage collection. The way it works by freezing all the application threads while doing garbage collection may not be suitable for a server environment. It is best suited for simple command-line programs.

Turn on the -XX:+UseSerialGC JVM argument to use the serial garbage collector.

**2) Parallel Garbage Collector**

Parallel garbage collector is also called as throughput collector. It is the default garbage collector of the JVM. Unlike serial garbage collector, this uses multiple threads for garbage collection. Similar to serial garbage collector this also freezes all the application threads while performing garbage collection.

Note : Multiple threads are used for minor garbage collection. A single thread is used for major garbage collection and Old Generation compaction. Alternatively, the Parallel Old variant uses multiple threads for major garbage collection and Old Generation compaction.

**3) CMS Garbage Collector**

Concurrent Mark Sweep (CMS) garbage collector uses multiple threads to scan the heap memory to mark instances for eviction and then sweep the marked instances. CMS garbage collector holds all the application threads in the following two scenarios only,

1. while marking the referenced objects in the tenured generation space.
2. if there is a change in heap memory in parallel while doing the garbage collection.

In comparison with parallel garbage collector, CMS collector uses more CPU to ensure better application throughput. If we can allocate more CPU for better performance then CMS garbage collector is the preferred choice over the parallel collector.

Turn on the XX:+USeParNewGC JVM argument to use the CMS garbage collector.

**Note :** Multiple threads are used for minor garbage collection using the same algorithm as Parallel. Major garbage collection is multi-threaded, like Parallel Old, but CMS runs concurrently alongside application processes to minimize “stop the world” events (i.e. when the garbage collector running stops the application).

**4) G1 Garbage Collector**

G1 (Garbage First) Garbage Collector is designed for applications running on multi-processor machines with large memory space. It’s available since JDK7 Update 4 and in later releases.

G1 collector will replace the CMS collector since its more performance efficient.

Unlike other collectors, G1 collector partitions the heap into a set of equal-sized heap regions, each a contiguous range of virtual memory. When performing garbage collections, G1 shows a concurrent global marking phase (i.e. phase 1 known as Marking) to determine the liveness of objects throughout the heap.

After the mark phase is completed, G1 knows which regions are mostly empty. It collects in these areas first, which usually yields a significant amount of free space (i.e. phase 2 known as Sweeping). It is why this method of garbage collection is called Garbage-First.

To enable G1 Garbage Collector, we can use the following argument:

|  |  |
| --- | --- |
|  | java -XX:+UseG1GC -jar Application.java |

# **Q 12) One important change in Memory Management in Java 8**

Oracle’s latest edition for Java – Java 8 was released in March 2014. As usual, tons of new features have been added. There is one major change in the Memory management area

So long PermGen, Hello Metaspace !!”

Oracle has completely gotten rid of ‘PermGen’ and replaced it with Metaspace.

**What is PermGen ?**

Short form for Permanent Generation, PermGen is the memory area in Heap that is used by the JVM to **store class and method objects**. If your application loads lots of classes, PermGen utilization will be high. PermGen also **holds ‘interned’ Strings**

The size of the PermGen space is configured by the Java command line option**-XX:MaxPermSize**

Typically 256 MB should be more than enough of PermGen space for most of the applications

However, It is not unusal to see the error “**java.lang.OutOfMemoryError: PermGen space“** if you are loading unusual number of classes.

Gone are the days of OutOfMemory Errors due to PermGen space.

**With Java 8, there is NO PermGen**. That’s right. So no more OutOfMemory Errors due to PermGen

The key difference between PermGen and Metaspace is this: while PermGen is part of Java Heap (Maximum size configured by -Xmx option), **Metaspace is NOT part of Heap.** Rather Metaspace is part of **Native Memory (process memory)** which is only limited by the Host Operating System.