How Garbage Collection (GC) works internally in detail in java - BEST EXPLANATION EVER

In this tutorial we will learn **Garbage collection (Gc) in java - Internals in detail - With the best explanation ever** in java.
Garbage collection (Gc) in the **heart of core java**. It is very **important** topic. Every java developer must learn about this. This is very very important topic from **interview** perspective for **experienced** java **developers** but **freshers** also must learn about garbage collection.

Contents of page >

- **4.3) Permanent** Generation (**VM PARAMETERS for** Permanent Generation)
- **4.4) Other important VM (JVM) parameters for java heap in java >**

- **5) Let's discuss different Garbage collectors in detail >**
  - **5.1) Serial collector / Serial GC (Garbage collector)** in java
    - **5.1.1. Features of Serial GC (Garbage collector) in java >**
    - **5.1.2. When to Use the Serial GC (garbage Collector) in java >**
    - **5.1.3. Vm** (JVM) option for enabling **serial GC (garbage Collector)** in java >
  - **5.2) Throughput GC (Garbage collector) or Parallel collector in java**
    - **5.2.1. Features of Throughput GC (Garbage collector) in java >**
    - **5.2.2. When to Use the Throughput GC (Garbage collector) in java >**
    - **5.2.3. Vm** (JVM) option for enabling throughput GC (Garbage collector) in java >
    - **5.2.4. Goals** for Throughput GC (Garbage collector) in java >
    - **5.2.5.**Read in more **detail** about following features of Throughput GC (Garbage collector) in java
  - **5.3) Incremental low pause garbage collector (train low pause garbage collector) in java**
  - **5.4) Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java**
    - **5.4.1. Features** of Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java >
    - **5.4.2. When to Use the Concurrent Low Pause Collector in java**
    - **5.4.3. Vm** (JVM) option for enabling **Concurrent Mark Sweep** (CMS) Collector in java >
    - **5.4.4. Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector working in detail in java >**
      - **5.4.4.1. Major gc(garbage collection)** in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java >
      - **5.4.4.2. Minor gc (garbage collection)** in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector >
    - **5.4.5. Heap Structure for CMS garbage Collector**
    - **5.4.6. Detailed Steps** in GC (garbage collection) cycle in Concurrent Mark Sweep (CMS) Collector / concurrent low pause garbage collector in java >
    - **5.4.7.** Read in more **detail** about following features of Concurrent Mark Sweep (CMS) collector / concurrent low pause garbage collector in java
  - **5.5) G1 Garbage Collector (or Garbage First) in java**
    - 5.5.1. The G1 garbage collector **features** -
    - **5.5.2. Vm** (JVM) option for enabling G1 Garbage Collector (or Garbage First) in java >

1) Terms frequently used in Garbage Collection (GC) in java-

What is **Throughput** in gc(garbage collection) in java ?
In short, Throughput is the **time not spent** in garbage collection (GC) (in percent).

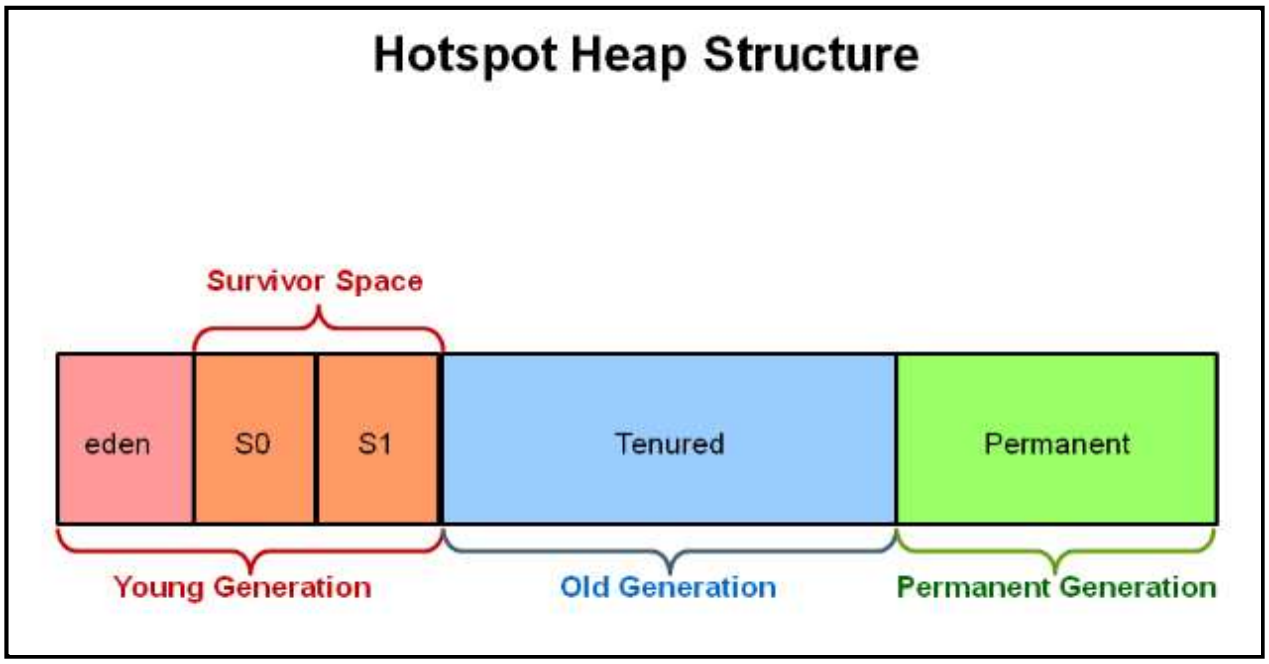
Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include >

- 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.

What are **pauses** in gc(garbage collection) in java ?

Pauses is applications pauses i.e. when **application doesn't gives any response** because of garbage collection (GC).

2) JVM Heap memory (Hotspot heap structure) with diagram in java >



2.1) JVM Heap memory (Hotspot heap structure) in java consists of following elements>

1. **Young** Generation
    - 1a) **Eden**,
    - 1b) **S0 (Survivor** space 0**)**
    - 1c) **S1 (Survivor** space 1**)**
2. **Old** Generation (Tenured)
3. **Permanent** Generation.

So, JVM Heap memory (Hotspot heap structure) is divided into three parts **Young** Generation, **Old** Generation (tenured) and **Permanent** Generation.

Young Generation is further divided into **Eden**, **S0 (Survivor** space 0**)** and **S1 (Survivor** space 1**).**

Read in detail about JVM Heap memory (**Hotspot heap structure**) with **diagram** in java

3) GARBAGE COLLECTION (**Minor and major** garbage collection) in JVM Heap memory (i.e. in young, old and permanent generation) >

3.1) **Young** Generation (***Minor garbage collection*** *occurs in* **Young Generation**)

New objects are allocated in Young generation.

Minor garbage collection *occurs in* **Young Generation.**

When ***minor garbage collection?***
When the young generation fills up, this causes a ***minor garbage collection***.
All the unreferenced (dead) objects are cleaned up from young generation.

When objects are **moved from young to old generation** in JVM *heap**?***
Some of the objects which aren't cleaned up **survive in young generation and gets aged**.
Eventually such objects are **moved from young to old generation**.

What is **Stop the World Event?**
Minor garbage collections are called **Stop the World** events**.**
All the non-daemon threads running in application are stopped during minor garbage collections (i.e. the application stops for while)**.**

Daemon threads performs minor garbage collection. (Daemon threads are low priority threads which runs intermittently in background for doing garbage collection).

- 1a) **Eden**,
- 1b) **S0 (Survivor** space 0**)**
- 1c) **S1 (Survivor** space 1**)**

3.2) Old Generation or (tenured generation) - (***Major garbage collection*** *occurs in* **Old Generation**)

The **Old Generation** is used for storing the long surviving aged objects (Some of the objects which aren't cleaned up **survive in young generation and gets aged.** Eventually such objects are **moved from young to old generation**).

Major garbage collection *occurs in* **Old Generation.**

At what time (or what age) objects are **moved from young to old generation** in JVM *heap**?***
There is some threshold set for young generation object and when that age is met, the object gets moved to the old generation during gc(garbage collection) in java.

What is **major garbage collection** in java?
When the old generation fills up, this causes a ***major garbage collection***. Objects are cleaned up from old generation.
Major collection is much slower than minor garbage collection in jvm heap **because it involves all live objects.**

Major garbage collection are **Stop the World Event** in java**?**
Major garbage collections are also called Stop the World events.
All the non-daemon threads running in application are stopped during major garbage collections.
Daemon threads performs major garbage collection.

Major gc(garbage collections) **in responsive applications** in java?
Major garbage collections should be minimized for responsive applications because applications must not be stopped for long.

Optimizing Major gc(garbage collections) in responsive applications in java?
Selection of appropriate garbage collector for the old generation space affects the length of the “Stop the World” event for a major garbage collection.

3.3) **Permanent** Generation or (Permgen) - (**full garbage collection** occurs in permanent generation in java).

Permanent generation Space contains metadata required by JVM to describe the classes and methods used in the application.

The permanent generation is included in a **full garbage collection** in java.

The permanent generation space is populated at runtime by JVM based on classes in use in the application.

The permanent generation space also contains **Java SE library classes and methods** in java.

JVM garbage collects those classes when classes are no longer required and space may be needed for other classes in java.

Read in more detail about >

What are **Young**, **Old** (tenured) and **Permanent Generation** in JVM in java

What are **Minor**, **Major** and **Full garbage collection** in JVM in java

4) Most important **VM** (JVM) **PARAMETERS in** JVM Heap memory >

Read: How to write java program to pass VM parameters through CMD

Learn how to pass **vmargs** (VM parameters) to java program in **eclipse**?

-Xms : Xms is **minimum heap size** which is allocated at initialization of JVM.

Examples of using **-Xms** VM (JVM) option in java >
Example1 of using **-Xms** VM (JVM) option in java >
java -Xms512m MyJavaProgram
It will set the minimum heap size of JVM to 512 megabytes.

Example2 of using **-Xms** VM (JVM) option in java >
java -Xms1g MyJavaProgram
It will set the minimum heap size of JVM to 1 gigabyte.

-Xmx : Xmx is the **maximum heap size** that JVM can use.

Examples of using **-Xmx** VM option in java >
Example1 of using **-Xmx** VM (JVM) option in java >
java -Xmx512m MyJavaProgram
It will set the maximum heap size of JVM to 512 megabytes.

Example2 of using **-Xmx** VM (JVM) option in java >
java -Xmx1g MyJavaProgram
It will set the maximum heap size of JVM to 1 gigabyte.

For more explanation and example - Read : What are **-Xms** and **-Xmx** JVM parameters and differences between them

4.1) **Young** Generation(**VM PARAMETERS for** Young Generation)

-Xmn : -Xmn sets the size of young generation.

Examples of using **-Xmn** VM (JVM) option in java >
Example1 of using **-Xmn** VM (JVM) option in java >
java -Xmn512m MyJavaProgram

Example2 of using **-Xmn** VM (JVM) option in java >
java -Xmn1g MyJavaProgram

For more explanation and example - Read : **-Xmn** JVM parameters

-XX:NewRatio : NewRatio controls the size of young generation.

Example of using -XX:NewRatio VM option in java >

-XX:NewRatio=3 means that the ratio between the young and old/tenured generation is 1:3.

In other words, the combined size of the eden and survivor spaces will be one fourth of the total heap size.

For more explanation and example - Read : What is **-XX:NewRatio** JVM parameters in java

-XX:NewSize - NewSize is **minimum size of young generation** which is allocated at initialization of JVM.

Note : If you have specified -XX:NewRatio than minimum size of the young generation is allocated automatically at initialization of JVM.

-XX:MaxNewSize - MaxNewSize is the **maximum size of young generation** that JVM can use.

For more explanation and example - Read : What are **-XX:NewSize** and **-XX:MaxNewSize** JVM parameters in java

- 1a) **Eden**,
- 1b) **S0 (Survivor** space 0**)**
- 1c) **S1 (Survivor** space 1**)**

-XX:SurvivorRatio : (for survivor space)

SurvivorRatio can be used to **tune the size of the survivor spaces**, but this is often not as important for performance.

Example of using -XX:SurvivorRatio > **-XX:SurvivorRatio=6 sets the ratio between each survivor space and eden to be 1:6.**

In other words, each survivor space will be one eighth of the young generation (not one seventh, because there are two survivor spaces).

For more explanation and example - Read : **-XX:SurvivorRatio** JVM parameters in java

4.2) **Old** Generation (tenured) - (**VM PARAMETERS for** Old Generation)

-XX:NewRatio : NewRatio controls the size of young and old generation.

Example of using -XX:NewRatio, **-XX:NewRatio=3 means that the ratio between the young and old/tenured generation is 1:3**. In other words, the combined size of the eden and survivor spaces will be one fourth of the total heap size.

For more explanation and example - Read : What is **-XX:NewRatio** JVM parameters

4.3) **Permanent** Generation (**VM PARAMETERS for** Permanent Generation)

-XX:PermSize: It's is initial value of Permanent Space which is allocated at startup of JVM.

Examples of using -XX:PermSize VM (JVM) option in java >

Example1 of using **-XX:PermSize** VM (JVM) option in java >
java -XX:PermSize=512m MyJavaProgram
It will set initial value of Permanent Space as 512 megabytes to JVM

Example2 of using **-XX:PermSize** VM (JVM) option in java >
java -XX:PermSize=1g MyJavaProgram
It will set initial value of Permanent Space as 512 gigabyte to JVM

-XX:MaxPermSize: It's maximum value of Permanent Space that JVM can allot up to.

Examples of using -XX:MaxPermSize VM option in java >

Example1 of using **-XX:MaxPermSize** VM (JVM) option in java >
java -XX:MaxPermSize=512m MyJavaProgram
It will set maximum value of Permanent Space as 512 megabytes to JVM

Example2 of using **-XX:MaxPermSize** VM (JVM) option in java >
java -XX:MaxPermSize=1g MyJavaProgram
It will set maximum value of Permanent Space as 1 gigabyte to JVM

For more explanation and example - Read : What are **-XX:PermSize** and **-XX:MaxPermSize** with Differences

4.4) Other important VM (JVM) parameters for java heap in java >

-XX:MinHeapFreeRatio and -XX:MaxHeapFreeRatio

JVM can grows or shrinks the heap to keep the proportion of free space to live objects within a specific range.

-XX:+AggressiveHeap is used for Garbage Collection Tuning setting.
This VM option inspects the server resources and attempts to set various parameters in optimal manner for for long running and memory consuming applications. There must be minimum of 256MB of physical memory on the servers before the AggressiveHeap can be used.

For more explanation and example - Read : **-XX:+AggressiveHeap** VM parameters

-Xss > Use this VM option to **adjust the maximum thread stack size.**

Also you must know that -Xss option is same as **-XX:ThreadStackSize.**

Examples of using -Xss **VM option in java >**
Example1 of using -Xss >
java -Xss512m MyJavaProgram
It will set the default stack size of JVM to 512 megabytes.

Example2 of using -Xss >
java -Xss1g MyJavaProgram
It will set the default stack size of JVM to 1 gigabyte.

For more explanation and example - Read :

5) Let's discuss different Garbage collectors in detail >

5.1) Serial collector / Serial GC (Garbage collector) in java

5.1.1. Features of Serial GC (Garbage collector) in java >

- Serial collector is also called **Serial GC** (Garbage collector) in java.
- Serial collector is simply also called **Serial collector** in java.

- Serial GC (Garbage collector) is **rarely used** in java.
- Serial GC (Garbage collector) is designed **for** the **single threaded environments** in java.
- In Serial GC (Garbage collector) , both **minor and major garbage collections are done serially** by **one thread** (using a single virtual CPU) in java.
- Serial GC (Garbage collector) 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 in java.
- The serial garbage collector is the **default** for client style machines in **Java SE 5 and 6**.

5.1.2. When to Use the Serial GC (garbage Collector) in java >

- 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 in java.
- Serial GC (garbage collector) is also popular in **environments where a high number of JVMs are run on the same machine**. In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs in java.

5.1.3. **Vm** (JVM) option for enabling **serial GC (garbage Collector)** in java >

-XX:+UseSerialGC

Example of Passing Serial GC in Command Line for starting jar>

java -Xms256m -Xms512m **-XX:+UseSerialGC** -jar d:\MyJar.jar

Read in detail about **Serial collector / Serial GC (Garbage collector)**

5.2) Throughput GC (Garbage collector) or Parallel collector in java

5.2.1. Features of Throughput GC (Garbage collector) in java >

- Throughput collector is also called
  - Throughput GC (garbage collector)
  - ParalleIGC (garbage collector)
  - Throughput collector
  - ParalleIGC collector
- Throughput garbage collector is the **default garbage collector for JVM** in java**.**
- Throughput garbage collector **uses multiple threads to execute a minor collection** and so reduces the serial execution time of the application in java.
- The throughput garbage collector is **similar to the serial** garbage collector but uses multiple threads to do the minor collection in java.
- This garbage collector uses a **parallel version of the young** generation garbage collector in java.
- The **tenured** generation collector is the **same as the serial** garbage collector in java.

5.2.2. When to Use the Throughput GC (Garbage collector) in java >

The Throughput garbage collector should be used when application can **afford low pauses** in java.
And application is running on host with multiple CPU's (to derive advantage of using multiple threads for garbage collection) in java.

5.2.3. **Vm** (JVM) option for enabling throughput GC (Garbage collector) in java >

-XX:+UseParallelGC

Example of using **throughput** collector in Command Line for starting jar>
java -Xms256m -Xms512m **-XX:+UseParallelGC** -jar d:\MyJar.jar

With this **Vm** (JVM) option you get a

- **Multi-threaded young** generation garbage collector in java,
- **single-threaded old** generation garbage collector in java and
- **single-threaded compaction** of **old** generation in java.

Vm (JVM) option for enabling **throughput collector** with **n number of threads** in java >

-XX:ParallelGCThreads=<numberOfThreads>

Another Vm (JVM) option for enabling **throughput collector** in java >
-XX:+UseParallelOldGC

5.2.4. **Goals** for Throughput GC (Garbage collector) in java >

- Maximum pause time goal (Highest priority)
- Throughput goal
- Minimum footprint goal (Lowest priority)

5.2.5. Read in more **detail** about following features of Throughput GC (Garbage collector) in java

Performance of Throughput GC (garbage Collector) host with different number of CPU's in java
Vm (JVM) option for enabling **throughput collector** with **n number of threads** in java >
Another Vm (JVM) option for enabling **throughput collector** in java >
Controlling maximum pause time and throughput for the application in java >
Vm (JVM) option for **maximum pause time** in java **>**
Vm (JVM) option for **throughput** in java **>**
Adjusting Generation Sizes in throughput GC (Garbage collector).

5.3) Incremental low pause garbage collector (train low pause garbage collector) in java :

We won't be discussing in detail about incremental low pause garbage collector because is **not used these days** in java**.** Incremental low pause collector was used in Java 4.

Vm (JVM) option which was used for enabling Incremental low pause garbage collector in java >
-XX:+UseTrainGC.

5.4) Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java

5.4.1. **Features** of Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java >

- Concurrent Mark Sweep Collector is also called
  - concurrent low pause collector
  - concurrent low pause GC (garbage collector)

  - CMS GC (garbage Collector)
  - CMS Collector

  - concurrent low pause collector
  - concurrent low pause GC (garbage collector)

- Concurrent Mark Sweep (CMS) collector **collects the old/tenured generation** in java**.**
- Concurrent Mark Sweep (CMS) Collector **minimize the pauses** by doing most of the **garbage collection work concurrently with the application threads** in java**.**
- Concurrent Mark Sweep (CMS) Collector **on live objects >**

  Concurrent Mark Sweep (CMS) Collector **does not copy or compact the live objects.** A garbage collection is done **without moving the live objects**. If fragmentation becomes a problem, allocate a larger heap in java**.**

5.4.2. When to Use the Concurrent Low Pause Collector in java

- Concurrent Low Pause Collector should be used if your **applications that require low garbage collection pause times** in java**.**
- Concurrent Low Pause Collector should be used when your **application can** afford to **share processor resources with the garbage collector while** the **application is running** in java**.**
- Concurrent Low Pause Collector is beneficial to applications which have a relatively **large set of long-lived data** (a large tenured generation) and run on machines with **two or more processors** in java**.**

Examples when to use Concurrent Mark Sweep (CMS) collector / concurrent low pause collector should be used for >

Example 1 - Desktop UI application that **respond to events**,
Example 2 - Web server responding to a request and
Example 3 - Database responding to queries.

5.4.3. **Vm** (JVM) option for enabling **Concurrent Mark Sweep** (CMS) Collector in java >

-XX:+UseConcMarkSweepGC

Example of using Concurrent Mark Sweep (CMS) collector **/ concurrent low pause collector** in Command Line for starting jar>

java -Xms256m -Xms512m **-XX:+UseConcMarkSweepGC** -jar d:\MyJar.jar

5.4.4. Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector working in detail in java >

As mentioned above Concurrent Mark Sweep (CMS) collector **collects the old/tenured generation (i.e.** performs ***Major garbage collection*** process**).**

5.4.4.1. **Major gc(garbage collection)** in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java >

For each major collection the CMS collector will **pause all** the **application threads for a brief period** at the **beginning** of the collection and toward the **middle** of the collection**.**

The **second pause** tends to be the **longer** than first pause and **uses multiple threads to do the collection** work during that pause in java**.** The remainder of the collection is done with a garbage collector thread that runs concurrently with the application**.**

5.4.4.2. **Minor gc (garbage collection)** in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector >

The minor collections is done in a manner **similar to the serial collector** although **multiple threads are used** to do the collection in java**.**

5.4.5. Heap Structure for CMS garbage Collector

CMS garbage collectors didies heap into three sections: **young** generation, **old** generation, and **permanent** generation of a fixed memory size.

Young Generation is further divided into **Eden**, **S0 (Survivor** space 0**)** and **S1 (Survivor** space 1**).**

5.4.6. **Detailed Steps** in GC (garbage collection) cycle in Concurrent Mark Sweep (CMS) Collector / concurrent low pause garbage collector in java >

Young Generation GC (garbage Collection) in java

- **Live objects** are **copied** from the **Eden space and survivor space** to the **other survivor space**.
- Any **older objects** that have reached their aging threshold are **promoted to old generation**.

After Young generation GC (garbage Collection) in java

- After a young GC, the **Eden space and one of the survivor spaces is cleared**.
- promoted objects (**older objects** that have reached their aging threshold in young GC) are are **available in old generation**.

Old Generation GC (garbage Collection) **with CMS** in java

1. **Initial mark** phase **-** (**First pause** happens/ stop the world event) **- mark live/reachable objects** (Example **-** objects on thread stack, static objects etc.) and elsewhere in the **heap** (Example **-** the young generation).

2. **Concurrent marking** phase **- (**No pause phase) **-** finds **live objects** while the application continues to execute.

3. **Remark -** (**Second pause** happens/ stop the world events) - It **finds objects** that were **missed during the concurrent marking phase due to the concurrent execution of the application threads**.

   Old Generation GC (garbage Collection) **- Sweep phase** (Concurrent Sweep phase) in java

4. **Sweep** phase **-** do the concurrent **sweep**, memory is freed up.
   - Objects that were not marked in the previous phase are deallocated in place.
   - There is no compaction

   Unmarked objects are equal to **Dead Objects.**

   Old Generation GC (garbage Collection) **- After Sweeping**

5. **Reset** phase **-** do the concurrent **reset**.

5.4.7. Read in more **detail** about following features of Concurrent Mark Sweep (CMS) collector / concurrent low pause garbage collector in java

- **Young Generation Guarantee** in Concurrent Mark Sweep (CMS) garbage Collector / concurrent low pause garbage collector in java >
- **Full Collections** in Concurrent Mark Sweep (CMS) garbage Collector / concurrent low pause garbage collector in java >

- **Vm** (JVM) option for enabling Concurrent Mark Sweep (CMS) garbage Collector **/ concurrent low pause collector** with **n number of threads** in java >
- **Fallback** with CMS garbage collector in java >
- **Pauses (STW/ stop the world events)** in Concurrent Mark Sweep (CMS) garbage Collector / concurrent low pause garbage collector in java >
- **Scheduling pauses** in Concurrent Mark Sweep (CMS) Collector / concurrent low pause garbage collector in java >
- **More Vm** (JVM) **option** for Concurrent Mark Sweep (CMS) collector **/ concurrent low pause garbage collector** in Command Line for starting jar in java>

5.5) G1 Garbage Collector (or Garbage First) in java

5.5.1. The G1 garbage collector **features** -

- G1 garbage collector is also called
  - G1 garbage collector
  - G1 collector
  - G1 GC (garbage collector)
  - Garbage first collector

- G1 garbage collector was **introduced in Java 7**
- G1 garbage collector was designed to replace CMS collector(Concurrent Mark-Sweep garbage Collector).
- G1 garbage collector is **parallel**,
  - G1 garbage collector is **concurrent**, and
  - G1 garbage collector is **incrementally compacting low-pause** garbage collector in java.
- G1 garbage collector has much better layout from the other garbage collectors like serial, throughput and CMS garbage collectors in java.
- G1(Garbage First) collector **compacts sufficiently to completely avoid the use of fine-grained free lists for allocation**, and instead relies on regions.
- G1(Garbage First) collector **allows customizations** by allowing users to specify pause times.
- G1 Garbage Collector (or Garbage First) limits GC **pause times and maximizes throughput.**

5.5.2. **Vm** (JVM) option for enabling G1 Garbage Collector (or Garbage First) in java >

-XX:+UseG1GC

Example of using G1 Garbage Collector in Command Line for starting jar>
java -Xms256m -Xms512m **-XX:+UseG1GC** -jar d:\MyJar.jar

5.5.3. G1(Garbage First) collector functioning >

CMS garbage collectors divides heap into three sections: young generation, old generation, and permanent generation of a fixed memory size.

All memory objects end up in one of these three sections.

The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.

The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions), **but** there is not a **fixed size** for them. This provides **greater flexibility in memory usage.**

5.5.4. When to use G1 garbage collector >

G1 must be used when applications that require **large heaps** with limited GC latency.
Example - Application that require

- **heaps around 5-6GB or larger** and
- **pause time** required **below 0.5 seconds**

5.5.5. When to switch from CMS (or old garbage collectors) to G1 garbage collector >

Applications using CMS garbage collector may switch to G1 when >

- **Full GC** durations are too **long** or too **frequent**.
- The **rate** of **object allocation** or **promotion varies** significantly.
- **Long garbage collection** (longer than 0.5 to 1 second)

5.5.6. The G1(Garbage First) collector working Step by Step >

The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.

5.5.6.1. G1(Garbage First) garbage collector Heap Structure >

The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions), but there is not a fixed size for them. This provides greater flexibility in memory usage**.**

Each **region's size** is chosen **by JVM at startup**.
Generally heap is divided into **2000 regions** by JVM varying in size from **1 to 32Mb**.

5.5.6.2. G1(Garbage First) garbage collector Heap Allocation >

As mentioned above there are following region in heap >

Eden, **survivor** and **old** generation region. Also,
Humongous and unused regions are there in heap.

5.5.6.3. Young Generation in G1 garbage collector

Generally heap is divided into **2000 regions** by JVM.
Minimum size of region can be **1Mb** and
Maximum size of region can be **32Mb**.

Regions are not required to be contiguous like CMS garbage collector.

Young GC in G1 garbage collector

- **Live objects** are copied or moved **to survivor regions.**
- If objects aging threshold is met it get promoted to **old** generation regions.
- It is **STW** (stop the world) event. Eden size and survivor size is calculated for the next young GC.
- The young GC is done parallely using multiple threads.

End of a Young GC with G1 garbage collector

At this stage **Live objects have been evacuated (copied or moved) to** >

- **survivor** regions or
- **old** generation regions.

5.5.6.4. Old Generation Collection with G1 garbage collector

G1 collector is low pause collector for old generation objects.

Initial Mark -

- It is **STW** (stop the world) event.
- With G1, it is **piggybacked on a normal young GC**. Mark survivor regions (root regions) which may have references to objects in old generation.

Root Region Scanning -

- **Scan survivor regions for references into the old generation**.
- This happens while the **application continues to run**. The phase must be **completed before** a young GC can occur.

Concurrent Marking -

- **Find live objects over the entire heap.**
- This happens while the **application is running**.
- This phase can be interrupted by young generation garbage collections.

Remark (Stop the World Event) -

- **Completes the marking of live object in the heap**.
- Uses an algorithm called **snapshot-at-the-beginning** (SATB) which is much **faster** than algorithm used in the **CMS** collector.

Cleanup (Stop the World Event and Concurrent) -

- **Performs accounting on live objects and completely free regions**. (Stop the world)
- Young generation and old generation are reclaimed at the same time
- Old generation regions are selected based on their liveness.

- **Scrubs** the Remembered Sets. (Stop the world)
- **Reset** the **empty regions** and return them to the free list. (Concurrent)

5.5.7. Read in more **detail** about following features of G1 garbage collector / Garbage first collector in java.

5.6) Difference between Serial GC (Garbage collector) vs Throughput GC (Garbage collector) in java?

Difference Serial and Throughput gc (garbage Collector)>

Serial collector **uses one thread to execute garbage collection.**
Throughput collector **uses multiple threads to execute garbage collection.**

Serial GC is the garbage collector of choice for applications that do **not** have **low pause time requirements** and run on client-style machines.
Throughput GC is the garbage collector of choice for applications that have **low pause time requirements**.

Similarity between **Serial and Throughput gc (garbage Collector) in java>**

Throughput collector is similar to the serial collector in terms of garbage collection process.

5.7) ParNew collector - is the young generation collector. It is the parallel copy collector, it uses multiple threads in parallel. Vm parameter for enabling ParNew collector is -XX:+UseParNewGC.

5.8) PS Scavenge and PS MarkSweep

Read : PS Scavenge and PS MarkSweep

6) What is Automatic Garbage Collection in JVM heap memory in java?

Automatic garbage collection is the process of

- **Identifying objects which are in use** in java heap memory and
- **Which objects are not in use** in java heap memory and
- **deleting the unused objects** in java heap memory.

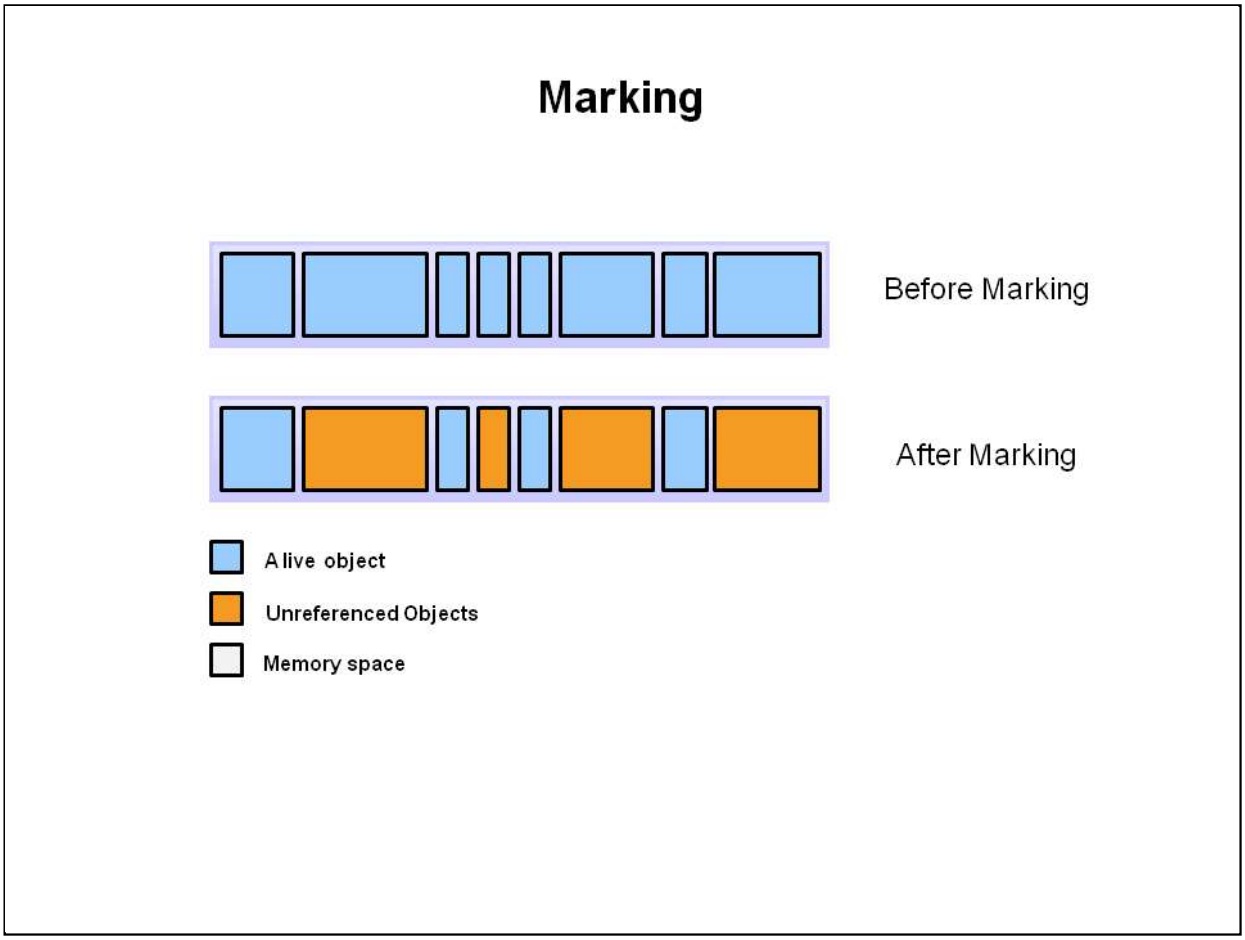
How to **Identify objects which are in use** in JVM heap memory in java?
Objects in use (or **referenced objects**) are those objects which are still needed by java program, some part of java program is still pointing to that object.

Which objects are not in use in JVM heap memory in java?
Objects not in use (or **unreferenced objects**) are those objects which are not needed by java program, no part of java program is pointing to that object.
So, these unused objects can be cleaned in GC (garbage collection) process and memory used by an unreferenced object can be reclaimed.

7) Now let's understand how garbage collection is done using **Marking** and **deletion** in java**.**

7.1) Step 1 > Marking

Marking is a process in which gc (garbage collector) identifies which parts of memory (occupied by objects) are in use and which are not**.**



Before Marking >

All the objects are shown in **blue,** at this stage

- some of objects might be **in use** (referenced objects) and
- some of objects might **not be in use** (unreferenced objects) **.**

After Marking >

Objects in use (or **referenced objects** or **Alive objects**) are shown in **blue**.

Objects not in use (or **unreferenced objects**) objects are shown in **Orange**.

7.2) Step 2 > Deletion

Step 2a : Normal Deletion

- **Normal deletion removes all the unreferenced objects** and
- **leaves referenced objects and pointers to free space**.

Step 2b : Deletion with Compacting

Deletion with Compacting is done to improve the performance than normal deleting.
Deletion with Compacting is done to improve the performance than normal deleting.

- Deletion with Compacting **removes all the unreferenced objects** and
- **compacts the remaining referenced objects** by **moving all the referenced objects together.**
- As all the referenced objects are moved together **new memory allocation becomes easier and much faster.**

Read in more detail about - **Marking and deleting objects** for garbage collection in java - **Mark and sweep algorithm**

8) Very **important** points about GC (Garbage Collection) in Java

1. All Java objects are always created on heap in java.

2. **What is GC (Garbage collection) process in java**?

   GC (Garbage collection) is the process by which JVM cleans objects (unused objects) from heap to reclaim heap space in java.

   OR
   What is Automatic Garbage Collection in JVM heap memory in java?
   Automatic garbage collection is the process of
   - **Identifying objects which are in use** in java heap memory and
   - **Which objects are not in use** in java heap memory and
   - **deleting the unused objects** in java heap memory.

3. How to **Identify objects which are in use** in java heap memory?

   Objects in use (or **referenced objects**) are those objects which is still needed by java program, some part of java program is still pointing to that object.

4. **Which objects are NOT in use** in java heap memory?

   Objects not in use (or **unreferenced objects** or **unused objects**) are those objects which is not needed by java program, no part of java program is pointing to that object. So, these unused objects can be cleaned in garbage collection process and memory used by an unreferenced object can be reclaimed.

5. GC (Garbage collection) process **automatically clears objects from heap to reclaim heap space**. You just need to specify the type of garbage collector type you want to use at JVM startup.

6. **Gc** (garbage collector) **calls finalize** method **for garbage collection**. finalize **method is called only once by garbage collector for an object in java**.

7. ***Daemon threads*** are low priority threads which **runs intermittently in background** for doing **garbage collection (gc) in java**.

8. We can ***force early gc*** *(garbage collection) in java* by using following methods >

```
System.gc();
Runtime.getRuntime().gc();

System.runFinalization();
Runtime.getRuntime().runFinalization();
```

Read : Is it good practice to call System.gc() in Java?

9. *By calling these methods JVM runs the finalize() methods of any objects pending finalization i.e. objects which have been discarded but there finalize method is yet to be run. After finalize method is executed JVM reclaims space from all the discarded objects in java.*

   ***Note** : Calling these methods does **not guarantee** that it will **immediately start performing garbage collection**.*

10. **Finalize method** execution is not assured - We must not override finalize method to write some critical code of application because methods execution is not assured. Writing

some critical code in finalize method and relying on it may make application to go horribly wrong in java.

11. Dealing with **OutOfMemoryError** in java.

12. **WeakhashMap** in java - java.util.WeakHashMap is hash table based implementation of the Map interface, with *weak keys*.

    An entry in a WeakHashMap will be automatically removed by garbage collector when its key is no longer in ordinary use. Read in detail about WeakhashMap.

13. Object which is set explicitly set to **null** becomes **eligible for gc** (garbage collection) in java .

Example 1 >

String s="abc"; //s is currently **not eligible** for gc (garbage collection) in java.
s = null; //Now, s is currently **eligible** for gc (garbage collection) in java.

Example 2 >

List list =new ArrayList(); //list is currently **not eligible** for gc (garbage collection).
list = null; //Now, list is currently **eligible** for gc (garbage collection).

14. **Difference in garbage collection in C/C++ and Java** (Hint : In terms of memory allocation and deallocation of objects)?

    In java garbage collection (memory allocation and deallocation of objects) is an **automatic** process.
    But, In C and C++ memory allocation and deallocation of objects) is a **manual** process.

15. All the **variables** declared **inside block** becomes **eligible for gc** (garbage collection) **when** we **exit** that **block** (As scope of those variable is only that block) in java.

    Example of garbage collection while using block in java -

```
class MyClass {
    public static void main(String[] args) {
            boolean var = false;
            if (var) { // begin block 1
                    int x = 1; // x is declared inside block
                    //.........
                    //code inside block...
```

```
                //.........
            } // end block 1 //And now x is eligible for gc (garbage collection)
            else { // begin block 2
                int y = 1;
                //.........
                //code inside block...
                //.........
            } // end block 2 //And now y is eligible for gc (garbage collection)
    }
}
```

Understanding Garbage Collection (GC) in java using VM argument **-verbose:gc** -

- **[GC 325407K->83000K(776768K), 0.2300771 secs]**
  - **GC** - GC indicates **minor Garbage Collection** (i.e. in young generation).
  - 325407K - The combined size of **live objects before** gc(garbage collection).
  - 83000K - The combined size of **live objects after** gc(garbage collection).
  - 0.2300771 secs - **time** it took for gc(garbage collection) to occur.

- **[Full GC 325407K->83000K(776768K), 0.2300771 secs]**
  - **Full GC** - Full GC Indicates **major garbage collection** (i.e. in tenured generation).

Read more about **-verbose:gc**

java.lang.OutOfMemoryError : unable to create new native Thread in java

You must know that each and every thread has its own stack, which makes the methods thread-safe as well.

You can resolve “java.lang.OutOfMemoryError : unable to create new native Thread” by setting the appropriate size using -Xss vm option in java.

click here to read more on **OutOfMemoryError : unable to create new native Thread**

How to solve OutOfMemoryError in eclipse? What is eclipse.ini file? How to pass vmargs to eclipse?

The file **eclipse.ini** contains the information which is passed to >

- Java Virtual Machine (JVM) as vmargs (because when eclipse is started it loads JVM as well) and
- **Eclipse** platform as well.

Eclipse.ini contains eclipse start up information,
-Xms (-Xms is **minimum heap size** which is allocated at initialization of JVM), -Xmx (-Xmx is the **maximum heap size** that JVM can use.)

-XX:PermSize: It's is initial value of Permanent Space which is allocated at startup of JVM..
-XX:MaxPermSize: It's maximum value of Permanent Space that JVM can allot up to.

Read : Most important and frequently used **VM (JVM) PARAMETERS** with examples in JVM Heap memory in java

eclipse.ini file is located in **eclipse directory.**

Read more about location and contents of **Eclipse.ini**

Read What is **eclipse.ini** file? How to **pass vmargs to java program in eclipse**? **Changing** the **eclipse setting**, What are **best eclipse setting**?

9) Summary of garbage collection in java-

I am highlighting most of the summary portion to keep it separate from whole tutorial. Please refer above for complete detail.

- **1) Terms frequently used in Garbage Collection (GC) in java-**
  - What is **Throughput** in gc(garbage collection) in java ?

Throughput is the **time not spent** in garbage collection (GC) (in percent).

  - What are **pauses** in gc(garbage collection) in java ?

Pauses is applications pauses when **application is paused beacuse** because of garbage collection (GC).

- **2) JVM Heap memory (Hotspot heap structure) with diagram in java >**
  - **2.1) JVM Heap memory (Hotspot heap structure) in java consists of following elements>**

JVM Heap memory (Hotspot heap structure) is divided into three parts **Young** Generation, **Old** Generation (tenured) and **Permanent** Generation.

Young Generation is further divided into **Eden**, **S0 (Survivor** space 0**)** and **S1 (Survivor** space 1**).**

- **3) GARBAGE COLLECTION (Minor and major** garbage collection) in JVM Heap memory (i.e. in young, old and permanent generation) >
  - **3.1) Young** Generation ***(Minor garbage collection*** *occurs in* ***Young Generation)***

New objects are allocated in Young generation.

When the young generation fills up, this causes a ***minor garbage collection***.
All the unreferenced (dead) objects are cleaned up from young generation.

  - **3.2) Old** Generation or (tenured generation) - (***Major garbage collection*** *occurs in* **Old Generation**)

The **Old Generation** is used for storing the long surviving aged objects.

When the old generation fills up, this causes a ***major garbage collection***. Objects are cleaned up from old generation.

  - **3.3) Permanent** Generation or (Permgen) - (**full garbage collection** occurs in permanent generation in java).

Permanent generation Space contains metadata required by JVM to describe the classes and methods used in the application.

The permanent generation is included in a **full garbage collection** in java.

- 4) Most important **VM** (JVM) **PARAMETERS in** JVM Heap memory >

-Xms : Xms is **minimum heap size** which is allocated at initialization of JVM.

-Xmx : Xmx is the **maximum heap size** that JVM can use.

- **4.1) Young** Generation(**VM PARAMETERS for** Young Generation)

-Xmn : -Xmn sets the size of young generation.
-XX:NewRatio : NewRatio controls the size of young generation.
-XX:NewSize - NewSize is **minimum size of young generation** which is allocated at initialization of JVM.
-XX:MaxNewSize - MaxNewSize is the **maximum size of young generation** that JVM can use.
-XX:SurvivorRatio : (for survivor space)
SurvivorRatio can be used to **tune the size of the survivor spaces.**

- **4.2) Old** Generation (tenured) - (**VM PARAMETERS for** Old Generation)

-XX:NewRatio : NewRatio controls the size of young and **old** generation.

- **4.3) Permanent** Generation (**VM PARAMETERS for** Permanent Generation)

-XX:PermSize: It's is initial value of Permanent Space which is allocated at startup of JVM.
-XX:MaxPermSize: It's maximum value of Permanent Space that JVM can allot up to.
-XX:PermSize: It's is initial value of Permanent Space which is allocated at startup of JVM.
-XX:MaxPermSize: It's maximum value of Permanent Space that JVM can allot up to.

- **4.4) Other important VM (JVM) parameters for java heap in java >**

-Xss > Use this VM option to **adjust the maximum thread stack size.**
-XX:MinHeapFreeRatio and -XX:MaxHeapFreeRatio
JVM can grows or shrinks the heap to keep the proportion of free space to live objects within a specific range.
-XX:+AggressiveHeap is used for Garbage Collection Tuning setting.

- **5) Let's discuss different Garbage collectors in detail >**

- **5.1) Serial collector / Serial GC (Garbage collector)** in java
  - **5.1.1. Features of Serial GC (Garbage collector) in java >**
    Serial GC (Garbage collector) is designed **for** the **single threaded environments** in java.
    In Serial GC (Garbage collector) , both **minor and major garbage collections are done serially** by **one thread** (using a single virtual CPU) in java.
  - **5.1.2. When to Use the Serial GC (garbage Collector) in java >**
    applications that do not have low pause time requirements

- **5.1.3. Vm** (JVM) option for enabling **serial GC (garbage Collector)** in java >

  -XX:+UseSerialGC

- **5.2) Throughput GC (Garbage collector) or Parallel collector in java**
  - **5.2.1. Features of Throughput GC (Garbage collector) in java >**

    Throughput garbage collector **uses multiple threads to execute a minor collection** and so reduces the serial execution time of the application in java.
  - **5.2.2. When to Use the Throughput GC (Garbage collector) in java >**

    The Throughput collector should be used when application can **afford low pauses** in java.
  - **5.2.3. Vm** (JVM) option for enabling throughput GC (Garbage collector) in java >

    -XX:+UseParallelGC
  - **5.2.4. Goals** for Throughput GC (Garbage collector) in java >

    Maximum pause time goal (Highest priority)

    Throughput goal

    Minimum footprint goal (Lowest priority)
  - **5.2.5.**Read in more **detail** about following features of Throughput GC (Garbage collector) in java

- **5.3) Incremental low pause garbage collector (train low pause garbage collector) in java :**

Not used these days, was used in java 4.

- **5.4) Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java**
  - **5.4.1. Features** of Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java >

    Concurrent Mark Sweep (CMS) collector **collects the old/tenured generation** in java**.** Concurrent Mark Sweep (CMS) Collector **minimize the pauses** by doing most of the **garbage collection work concurrently with the application threads** in java.

  - **5.4.2. When to Use the Concurrent Low Pause Collector in java**

    Concurrent Low Pause Collector should be used if your **applications that require low garbage collection pause times** in java**.**
  - **5.4.3. Vm** (JVM) option for enabling **Concurrent Mark Sweep** (CMS) Collector in java >

    -XX:+UseConcMarkSweepGC
  - **5.4.4. Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector working in detail in java >**
    - **5.4.4.1 Major gc(garbage collection)** in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in

java >

For each major collection the CMS collector will **pause all** the **application threads for a brief period** at the **beginning** of the collection and toward the **middle** of the collection.

- **5.4.4.2 Minor gc (garbage collection)** in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector >

  The minor collections is done in a manner **similar to the serial collector** although **multiple threads are used** to do the collection in java.

- **5.4.5. Heap Structure for CMS garbage Collector**

  CMS garbage collectors didies heap into three sections: **young** generation, **old** generation, and **permanent** generation of a fixed memory size.
  Young Generation is further divided into **Eden**, **S0 (Survivor** space 0**)** and **S1 (Survivor** space 1**).**

- **5.4.6. Steps (in short)** in GC (garbage collection) cycle in Concurrent Mark Sweep (CMS) Collector / concurrent low pause garbage collector in java >

  Young GC (Generation garbage) Collection happens. Then,
  Than Old Generation GC (garbage Collection) **happens.**

  1. **initial mark > stop** all application threads; mark all live **objects**; **resume** all application threads
  2. **concurrent mark** > do the concurrent **mark** (one processor is used for concurrent work)
  3. **Remark > stop** all application threads; do the **remark**; **resume** all application threads
  4. **sweep** > do the concurrent **sweep**, memory is freed up (one processor is used for concurrent work)
  5. **reset** > do the concurrent **reset** (one processor is used for concurrent work)

- **5.4.7.** Read in more **detail** about following features of Concurrent Mark Sweep (CMS) collector / concurrent low pause garbage collector in java.

5.5) G1 Garbage Collector (or Garbage First) in java

- 5.5.1. The G1 garbage collector **features** -

  G1 garbage collector was **introduced in Java 7**
  G1 garbage collector was designed to replace CMS garbage Collector.
  G1 garbage collector is **parallel** and **concurrent**, and
  G1 Garbage Collector (or Garbage First) **limits GC pause times** and **maximizes throughput.**

- **5.5.2. Vm** (JVM) option for enabling G1 Garbage Collector (or Garbage First) in java >

  -XX:+UseG1GC

- 5.5.3. G1(Garbage First) collector functioning >

  CMS garbage collectors divides heap into three sections: young generation, old generation, and permanent generation of a fixed memory size.
  The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions), **but** there is not a **fixed size** for them. This provides **greater flexibility in memory usage.**

- 5.5.4. When to use G1 garbage collector >

  G1 must be used when applications that require **large heaps** with limited GC latency.

- 5.5.5. When to switch from CMS (or old garbage collectors) to G1 garbage collector >

Full GC durations are too **long** or too **frequent**.

- **5.5.6.** The G1(Garbage First) collector working Step by Step >

The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.

- 5.5.6.1. G1(Garbage First) garbage collector Heap Structure >

  The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions).

- 5.5.6.2. G1(Garbage First) garbage collector Heap Allocation >

  Live objects are **moved or copied** from **one region to another.**

  As mentioned above there are following region in heap >

  Eden, **survivor** and **old** generation region.

  Also, **Humongous and unused** regions are there in heap.

- 5.5.6.3. Young Generation in G1 garbage collector

  Generally heap is divided into **2000 regions** by JVM.

  Minimum size of region can be **1Mb** and

  Maximum size of region can be **32Mb**.

  Young GC in G1 garbage collector

  - **Live objects** are copied or moved **to survivor regions.**
  - If objects aging threshold is met it get promoted to **old** generation regions.
  - It is **STW** (stop the world) event. Eden size and survivor size is calculated for the next young GC.

  End of a Young GC with G1 garbage collector

  At this stage **Live objects have been evacuated (copied or moved) to survivor** regions or **old** generation regions.

- 5.5.6.4. Old Generation Collection with G1 garbage collector

  Initial Mark -

  - It is **STW** (stop the world) event.
  - Mark survivor regions (root regions) which may have references to objects in old generation.

  Root Region Scanning -

  - **Scan survivor regions for references into the old generation.**
  - This happens while the **application continues to run.**

  Concurrent Marking -

  - **Find live objects over the entire heap.**
  - This happens while the **application is running.**

  Remark (Stop the World Event) -

  - **Completes the marking of live object in the heap.**

  Cleanup (Stop the World Event and Concurrent) -

  - **Performs accounting on live objects and completely free regions.** (Stop the world)
  - Young generation and old generation are reclaimed at the same time
  - **Reset** the **empty regions** and return them to the free list. (Concurrent)

- **5.5.7.** Read in more **detail** about following features of G1 garbage collector / Garbage first collector in java.

- **5.6) Difference between Serial GC (Garbage collector) vs Throughput GC (Garbage collector) in java?**

  Serial collector **uses one thread to execute garbage collection.**

  Throughput collector **uses multiple threads to execute garbage collection.**

- **6) What is Automatic Garbage Collection in JVM heap memory in java?**

- How to **Identify objects which are in use** in JVM heap memory in java?

  Objects in use are those objects which are still needed by java program.
- **Which objects are not in use** in JVM heap memory in java?

  Objects in use are those objects which are NOT still needed by java program.

- 7) Now let's understand how garbage collection is done using **Marking** and **deletion** in java**.**
  - **7.1) Step 1 > Marking**

    Marking is a process in which gc (garbage collector) identifies which parts of memory (occupied by objects) are in use and which are not.
  - **7.2) Step 2 > Deletion**

    Normal deletion removes all the unreferenced objects during process of garabage collection in java.