**– Java Garbage Collection**

Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space to be allocated to objects created in future processing. One of the best features of Java programming language is the **automatic garbage collection**, unlike other programming languages such as C where memory allocation and deallocation is a manual process.

**Garbage Collector** is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program. All these unreferenced objects are deleted and space is reclaimed for allocation to other objects.

One of the basic ways of garbage collection involves three steps:

1. **Marking**: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.
2. **Normal Deletion**: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.
3. **Deletion with Compacting**: For better performance, after deleting unused objects, all the survived objects can be moved to be together. This will increase the performance of allocation of memory to newer objects.

**Memory Management in Java – Java Garbage Collection Types**

There are five types of garbage collection types that we can use in our applications. We just need to use the JVM switch to enable the garbage collection strategy for the application. Let’s look at each of them one by one.

1. **Serial GC (-XX:+UseSerialGC)**: Serial GC uses the simple **mark-sweep-compact** approach for young and old generations garbage collection i.e Minor and Major GC.

Serial GC is useful in client machines such as our simple stand-alone applications and machines with smaller CPU. It is good for small applications with low memory footprint.

1. **Parallel GC (-XX:+UseParallelGC)**: Parallel GC is same as Serial GC except that is spawns N threads for young generation garbage collection where N is the number of CPU cores in the system. We can control the number of threads using -XX:ParallelGCThreads=n JVM option.

Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses a single thread for Old Generation garbage collection.

1. **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
2. **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for the Old generation. CMS collector tries to minimize the pauses due to garbage collection by doing most of the garbage collection work concurrently with the application threads.

CMS collector on the young generation uses the same algorithm as that of the parallel collector. This garbage collector is suitable for responsive applications where we can’t afford longer pause times. We can limit the number of threads in CMS collector using -XX:ParallelCMSThreads=n JVM option.

1. **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and its long term goal is to replace the CMS collector. The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

Garbage First Collector doesn’t work like other collectors and there is no concept of Young and Old generation space. It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects the region with lesser live data, hence “Garbage First”. You can find more details about it at [Garbage-First Collector Oracle Documentation](https://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html).

**Java Garbage Collection Monitoring**

We can use the Java command line as well as UI tools for monitoring garbage collection activities of an application. For my example, I am using one of the demo application provided by Java SE downloads.

If you want to use the same application, go to [Java SE Downloads](https://www.oracle.com/technetwork/java/javase/downloads/index.html) page and download **JDK 7 and JavaFX Demos and Samples**. The sample application I am using is **Java2Demo.jar** and it’s present in jdk1.7.0\_55/demo/jfc/Java2D directory. However this is an optional step and you can run the GC monitoring commands for any java application.

Command used by me to start the demo application is:

pankaj@Pankaj:~/Downloads/jdk1.7.0\_55/demo/jfc/Java2D$ java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar Java2Demo.jar

**jstat**

We can use jstat command line tool to monitor the JVM memory and garbage collection activities. It ships with standard JDK, so you don’t need to do anything else to get it.

For executing jstat you need to know the process id of the application, you can get it easily using ps -eaf | grep java command.

pankaj@Pankaj:~$ ps -eaf | grep Java2Demo.jar

501 9582 11579 0 9:48PM ttys000 0:21.66 /usr/bin/java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseG1GC -jar Java2Demo.jar

501 14073 14045 0 9:48PM ttys002 0:00.00 grep Java2Demo.jar

So the process id for my java application is 9582. Now we can run **jstat** command as shown below.

pankaj@Pankaj:~$ jstat -gc 9582 1000

S0C S1C S0U S1U EC EU OC OU PC PU YGC YGCT FGC FGCT GCT

1024.0 1024.0 0.0 0.0 8192.0 7933.3 42108.0 23401.3 20480.0 19990.9 157 0.274 40 1.381 1.654

1024.0 1024.0 0.0 0.0 8192.0 8026.5 42108.0 23401.3 20480.0 19990.9 157 0.274 40 1.381 1.654

1024.0 1024.0 0.0 0.0 8192.0 8030.0 42108.0 23401.3 20480.0 19990.9 157 0.274 40 1.381 1.654

1024.0 1024.0 0.0 0.0 8192.0 8122.2 42108.0 23401.3 20480.0 19990.9 157 0.274 40 1.381 1.654

1024.0 1024.0 0.0 0.0 8192.0 8171.2 42108.0 23401.3 20480.0 19990.9 157 0.274 40 1.381 1.654

1024.0 1024.0 48.7 0.0 8192.0 106.7 42108.0 23401.3 20480.0 19990.9 158 0.275 40 1.381 1.656

1024.0 1024.0 48.7 0.0 8192.0 145.8 42108.0 23401.3 20480.0 19990.9 158 0.275 40 1.381 1.656

The last argument for jstat is the time interval between each output, so it will print memory and garbage collection data every 1 second.

Let’s go through each of the columns one by one.

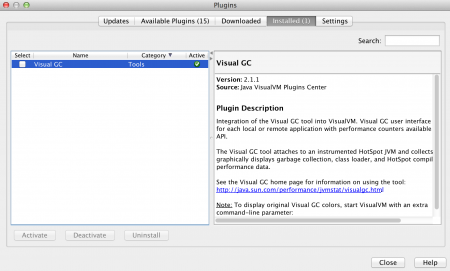
* **S0C and S1C**: This column shows the current size of the Survivor0 and Survivor1 areas in KB.
* **S0U and S1U**: This column shows the current usage of the Survivor0 and Survivor1 areas in KB. Notice that one of the survivor areas are empty all the time.
* **EC and EU**: These columns show the current size and usage of Eden space in KB. Note that EU size is increasing and as soon as it crosses the EC, Minor GC is called and EU size is decreased.
* **OC and OU**: These columns show the current size and current usage of Old generation in KB.
* **PC and PU**: These columns show the current size and current usage of Perm Gen in KB.
* **YGC and YGCT**: YGC column displays the number of GC event occurred in young generation. YGCT column displays the accumulated time for GC operations for Young generation. Notice that both of them are increased in the same row where EU value is dropped because of minor GC.
* **FGC and FGCT**: FGC column displays the number of Full GC event occurred. FGCT column displays the accumulated time for Full GC operations. Notice that Full GC time is too high when compared to young generation GC timings.
* **GCT**: This column displays the total accumulated time for GC operations. Notice that it’s sum of YGCT and FGCT column values.

The advantage of **jstat** is that it can be executed in remote servers too where we don’t have GUI. Notice that the sum of S0C, S1C and EC is 10m as specified through -Xmn10m JVM option.

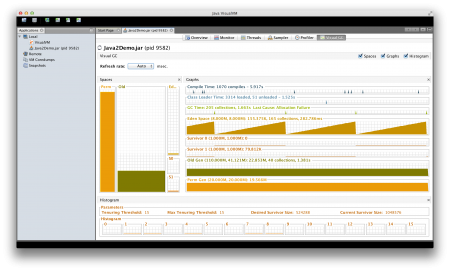
**Java VisualVM with Visual GC**

If you want to see memory and GC operations in GUI, then you can use jvisualvm tool. Java VisualVM is also part of JDK, so you don’t need to download it separately.

Just run jvisualvm command in the terminal to launch the Java VisualVM application. Once launched, you need to install **Visual GC** plugin from Tools -< Plugins option, as shown in below image.

[](https://cdn.journaldev.com/wp-content/uploads/2014/05/VisualVM-Visual-GC-Plugin.png)

After installing **Visual GC**, just open the application from the left side column and head over to **Visual GC** section. You will get an image of JVM memory and garbage collection details as shown in below image.

[](https://cdn.journaldev.com/wp-content/uploads/2014/05/Serial-GC-VisualGC.png)

**Java Garbage Collection Tuning**

**Java Garbage Collection Tuning** should be the last option you should use for increasing the throughput of your application and only when you see a drop in performance because of longer GC timings causing application timeout.

If you see java.lang.OutOfMemoryError: PermGen space errors in logs, then try to monitor and increase the Perm Gen memory space using -XX:PermGen and -XX:MaxPermGen JVM options. You might also try using -XX:+CMSClassUnloadingEnabled and check how it’s performing with CMS Garbage collector.

If you see a lot of Full GC operations, then you should try increasing Old generation memory space.

Overall garbage collection tuning takes a lot of effort and time and there is no hard and fast rule for that. You would need to try different options and compare them to find out the best one suitable for your application.

That’s all for Java Memory Model, Memory Management in Java and Garbage Collection, I hope it helps you in understanding JVM memory and garbage collection process.