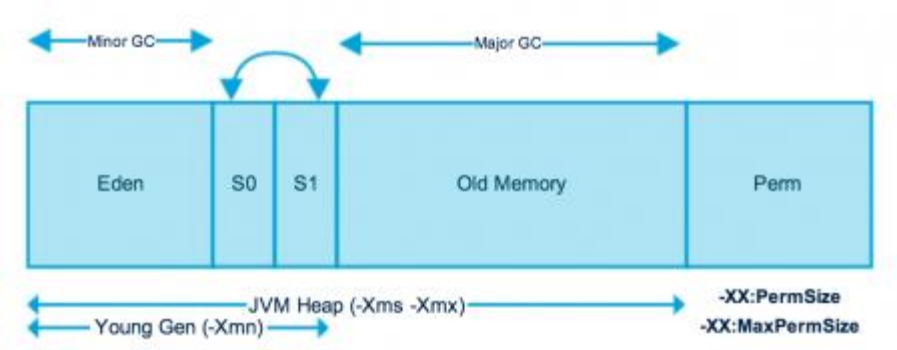
Understanding JVM Memory Model is very important if you want to understand the working of Java Garbage Collection. Today we will look into different parts of JVM memory and how to monitor and perform garbage collection tuning.

Java (JVM) Memory Model



JVM memory is divided into separate parts. JVM Heap and Perm Gen

JVM Heap

Heap memory is physically divided into two parts – Young Generation and Old Generation.

Young Generation

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called Minor GC. Young Generation is divided into three parts – Eden Memory and two Survivor Memory spaces.

Important Points about Young Generation Spaces:

Most of the newly created objects are located in the Eden memory space.
When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
Minor GC also checks the survivor objects and moves them to the other survivor space. So at a time, one of the survivor spaces is always empty.
Objects that are survived after many cycles of GC are moved to the Old generation memory space. Usually it's done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

Old Generation

Old Generation memory contains the objects that are long lived and survived after many rounds of Minor GC. Usually garbage collection is performed in Old Generation memory when it's full. Old Generation Garbage Collection is called Major GC and usually takes longer time.

Permanent Generation

Permanent Generation or "**Perm Gen**" contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java **Heap memory**
Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.
Method Area
Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.
Memory Pool
Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.
Java Stack Memory
Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.

Difference between Stack and Heap Memory

Java Heap Memory
Heap memory is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.
Java Stack Memory
Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.
Stack memory size is very less compared to Heap memory.

Java Heap Memory Switches

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

VM Switch	VM Switch Description
-Xms	For setting the initial heap size when JVM starts
-Xmx	For setting the maximum heap size.
-Xmn	For setting the size of the Young Generation, rest of the space goes for Old Generation.
-XX:PermGen	For setting the initial size of the Permanent Generation memory
-XX:MaxPermGen	For setting the maximum size of Perm Gen
-XX:SurvivorRatio	For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8.
-XX:NewRatio	For providing ratio of old/new generation sizes. The default value is 2.

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 the future processing. One of the best feature of java programming language is the automatic garbage collection, unlike other programming languages such as C where memory allocation and de-allocation 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:

Marking: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.
Normal Deletion: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.
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.
There are two problems with simple mark and delete approach.

- First one is that it's not efficient because most of the newly created objects will become unused
- Secondly objects that are in-use for multiple garbage collection cycle are most likely to be in-use for future cycles too.

Java Garbage Collection Types

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

- **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 standalone applications and machines with smaller CPU. It is good for small applications with low memory footprint.

- **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.

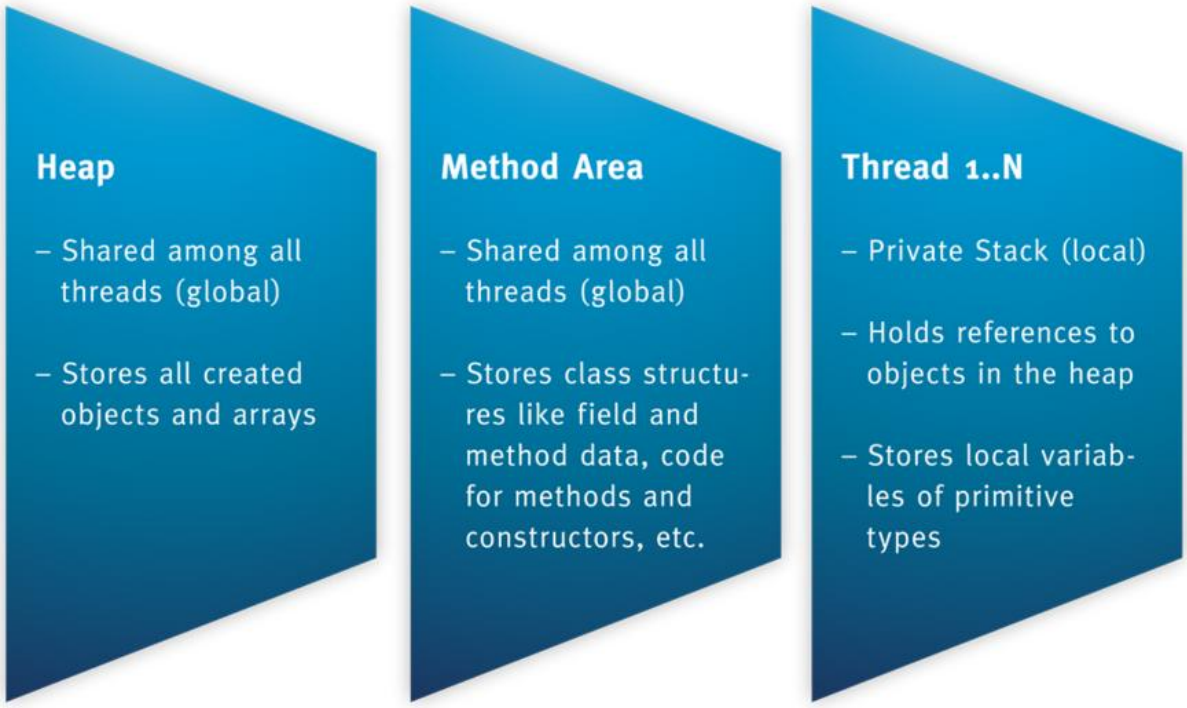
Parallel Garbage Collector is also called throughput collector because it uses multiple CPUs to speed up the GC performance. Parallel GC uses single thread for Old Generation garbage collection.

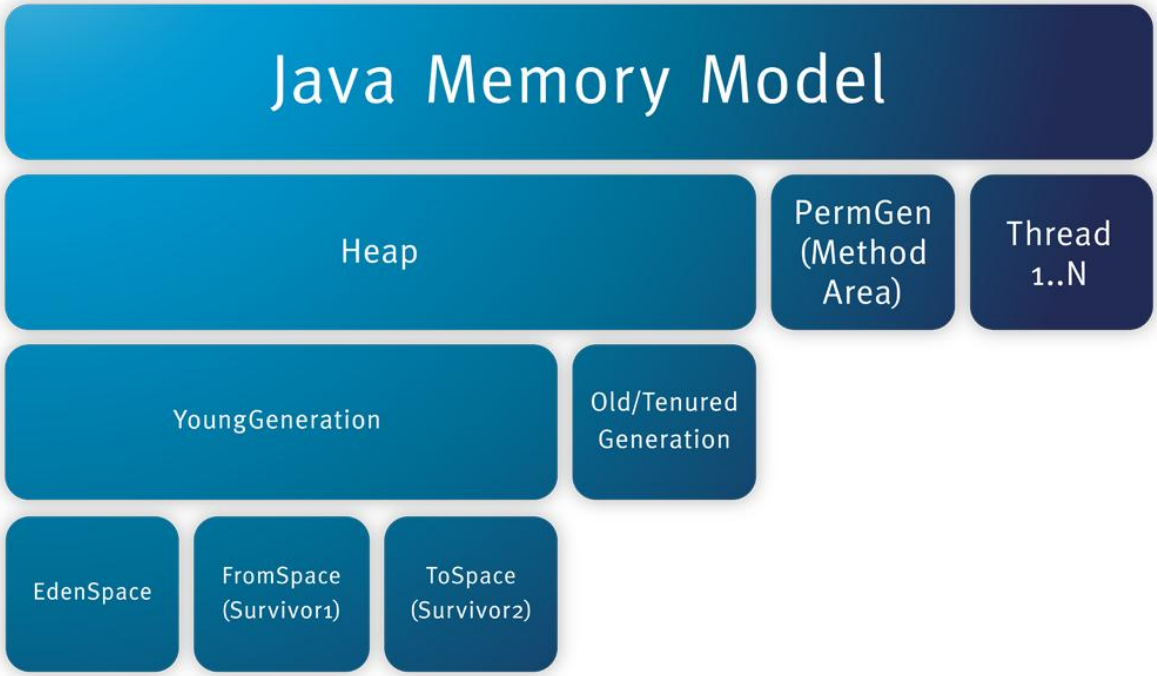
- **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.
- **Concurrent Mark Sweep (CMS) Collector (-XX:+UseConcMarkSweepGC)**: CMS Collector is also referred as concurrent low pause collector. It does the garbage collection for 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 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.

- **G1 Garbage Collector (-XX:+UseG1GC)**: The Garbage First or G1 garbage collector is available from Java 7 and it's 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".


Heap
– Shared among all threads (global)
– Stores all created objects and arrays
Method Area
– Shared among all threads (global)
– Stores class structures like field and method data, code for methods and constructors, etc.
Thread 1..N
– Private Stack (local)
– Holds references to objects in the heap
– Stores local variables of primitive types


Java Memory Model
Heap
PermGen (Method Area)
Thread 1..N
YoungGeneration
Old/Tenured Generation
EdenSpace
FromSpace (Survivor1)
ToSpace (Survivor2)

Heap memory

Since objects are stored in the heap part it is worth to have a closer look. The heap space itself is again separated into several spaces:

Young generation with eden and survivor space
Old Generation with tenured space
Each space harbors objects with different life cycles:

New/short-term objects are instantiated in the eden space.
Survived/mid-term objects are copied from the eden space to the survivor space.
Tenured/long-term objects are copied from the survivor to the old generation space.
By separating objects by their life time allows a shorter time consumption of the minor garbage collection and in return there is more cpu time for the application.

The reason is that in Java – unlike C – memory is freed (by destroying objects) automatically by two different garbage collectors: a minor and major garbage collection.

Instead of validating all objects in the heap – whether it can be destroyed or not – the minor garbage collector marks undestroyed objects with a garbage count. After a certain count the object is move to the old generation space.

A more detailed blog of the garbage collection will be discussed in another blog. For now it is sufficient to know that there are two garbage collectors.

OutOfMemoryError, but where?
Having this memory architecture in mind also helps to understand the different OutOfMemoryErrors like:

Exception in thread “main”: java.lang.OutOfMemoryError: Java heap space
Reason: an object could not be allocated into the heap space.
Exception in thread “main”: java.lang.OutOfMemoryError: PermGen space
Reason: classes and methods could not be loaded into the PermGen space. This occurs when an application requires a lot of classes e.g. in various 3rd party libraries.
Exception in thread “main”: java.lang.OutOfMemoryError: Requested array size exceeds VM limit
Reason: this occurs when an arrays is created larger than the heap size.
Exception in thread “main”: java.lang.OutOfMemoryError: request bytes for . Out of swap space?
Reason: this occurs when an allocation from the native heap failed and might be close to its limit. The indicates the source of the module where this error occurs.
Exception in thread “main”: java.lang.OutOfMemoryError: (Native method)
Reason: this error indicates that the problem originates from a native call rather than in the JVM.