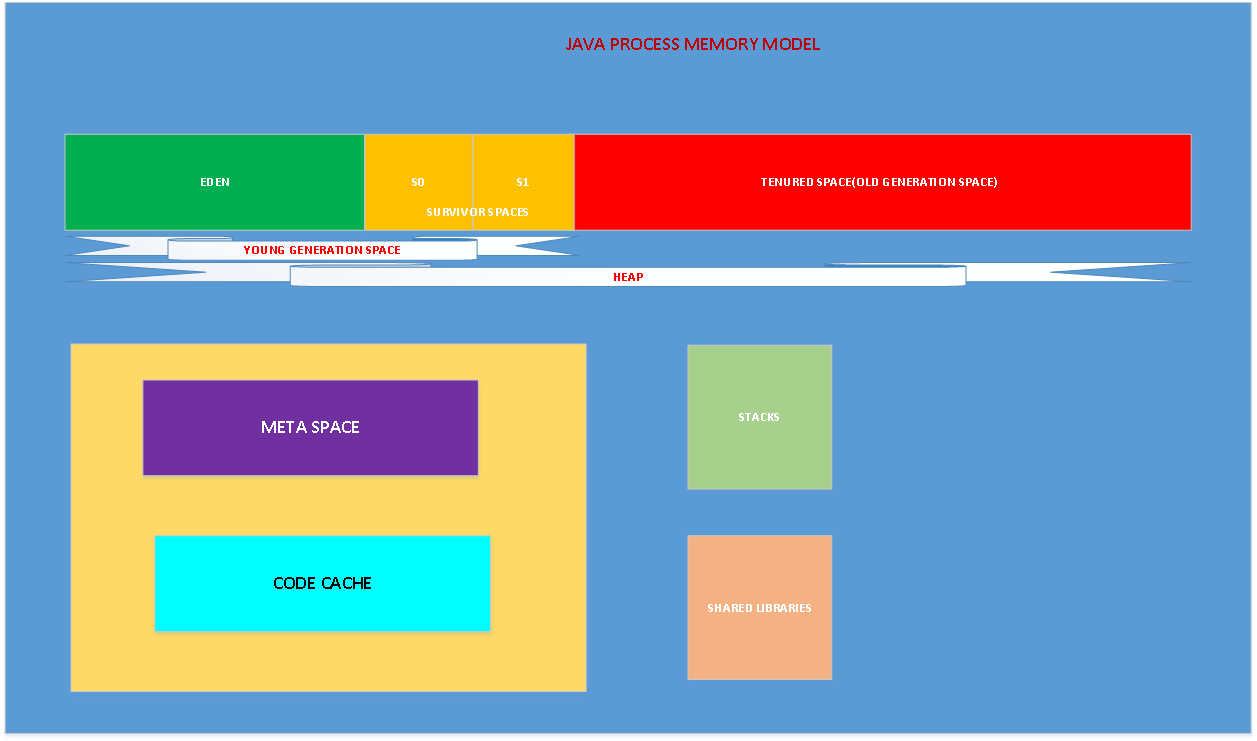
**Java Memory Model**

****

Once we launch the JVM, the **operating system allocates memory for the process**. Here, the JVM itself is a process, and the memory allocated to that process includes the **Heap**, **Meta Space**, **JIT code cache**, **thread stacks**, and **shared libraries**. We call this native memory. “**Native Memory**” is the memory provided to the process by the operating system. How much memory the operating system allocates to the Java process depends on the operating system, processor, and the JRE.

**Heap Memory:** JVM uses this memory to store objects. This memory is in turn split into two different areas called the “**Young Generation Space**” and “**Tenured Space**“.

**Young Generation:** The Young Generation or the New Space is divided into two portions called “**Eden Space**” and “**Survivor Space**".

**Eden Space:**When we create an object, the memory will be allocated from the Eden Space.

**Survivor Space:** This contains the objects that have survived from the Young garbage collection or Minor garbage collection. We have two equally divided survivor spaces called S0 and S1.

**Tenured Space:** The objects which reach to max tenured threshold during the minor GC or young GC, will be moved to “**Tenured Space**” or “**Old Generation Space**“.

**Meta Space:** This memory is out of heap memory and part of the native memory. As per the document by default the meta space doesn’t have an upper limit. In earlier versions of Java we called this “**Perm Gen Space**". This space is used to store the class definitions loaded by class loaders. This is designed to grow in order to avoid 0ut of memory errors. However, if it grows more than the available physical memory, then the operating system will use virtual memory. This will have an adverse effect on application performance, as swapping the data from virtual memory to physical memory and vice versa is a costly operation. We have JVM options to limit the Meta Space used by the JVM. In that case, we may get out of memory errors.

**Code Cache:** JVM has an interpreter to interpret the byte code and convert it into hardware dependent machine code. As part of JVM optimization, the Just In Time (JIT) compiler has been introduced. The frequently accessed code blocks will be compiled to native code by the JIT and stored it in code cache. The JIT compiled code will not be interpreted.

**Garbage Collection**

The JVM uses a separate demon thread to do garbage collection. As we said above when the application creates the object the JVM try to get the  required memory from the eden space. The JVM performs GC as minor GC and major GC. Let us understand the minor GC.

Initially the survivor space and the tenured space is empty. When the JVM is not able to get the memory from the eden space it initiates minor GC. During minor GC, the objects which are not reachable are marked to be collected. The JVM selects one of the survivor spaces as “To Space”. It might be S0/S1. Let us say the JVM selected S0 as the “To Space”. The JVM copies the reachable objects to “To Space”, S0, and increments the reachable object's age by 1. The objects which are not fit into the survivor space will be moved to the tenured space.

There is a JVM level option called “**MaxTenuringThreshold**” to specify the object age threshold to promote the object to tenured space. By default the value is 15.

So it is clear that minor GC reclaims the memory from the “**Young Generation Space***"*. Minor GC is a “**stop the world**” process. Some times the application pause is negligible. The minor GC will be performed with single thread or multi-thread, based on the GC collector applied.

If minor GC triggers several times, eventually the “**Tenured Space**” will be filled up and will require more garbage collection. During this time the JVM triggers a “**major GC**” event. Some times we call this full GC. But, as part of full GC, the JVM reclaims the memory from “Meta Space”. If there are no objects in the heap, then the loaded classes will be removed from the meta space.

Now let us see what possibilities make the JVM trigger major GC.

* If the developer calls System.gc(), or  Runtime.getRunTime().gc() suggests the JVM to initiate GC.
* If the JVM decides there is not enough tenured space.
* During minor GC, if the JVM is not able to reclaim enough memory from the eden or survivor spaces, then a major GC may be triggered.
* If we set a “MaxMetaspaceSize” option for the JVM and there is not enough space to load new classes, then the JVM triggers a major GC.

**Metaspace**

**Metaspace memory allocation model**

* Most allocations for the class metadata are now allocated out of native memory.
* The klasses that were used to describe class metadata have been removed.

**Metaspace capacity**

* By default class metadata allocation is limited by the amount of available native memory (capacity will of course depend if you use a 32-bit JVM vs. 64-bit along with OS virtual memory availability).
* A new flag is available (MaxMetaspaceSize), allowing you to limit the amount of native memory used for class metadata. If you don’t specify this flag, the Metaspace will dynamically re-size depending of the application demand at runtime.

**Metaspace garbage collection**

* Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the “MaxMetaspaceSize”.
* Proper monitoring & tuning of the Metaspace will obviously be required in order to limit the frequency or delay of such garbage collections. Excessive Metaspace garbage collections may be a symptom of classes, classloaders memory leak or inadequate sizing for your application.

**Java heap space impact**

* Some miscellaneous data has been moved to the Java heap space. This means you may observe an increase of the Java heap space following a future JDK 8 upgrade.

**Metaspace monitoring**

* Metaspace usage is available from the HotSpot 1.8 verbose GC log output.
* Jstat & JVisualVM have not been updated at this point based on our testing with b75 and the old PermGen space references are still present.

**G1GC is the default collector in Java 9.**

Types of Java Garbage Collectors

## 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** prior to java 9. 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.

## 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.

## 4. G1 Garbage Collector

G1 (Garbage First) Garbage Collector is designed for applications running on multi-processor machines with large memory space. G1 garbage collector is used for large heap memory areas. It separates the heap memory into regions and does collection within them in parallel. G1 also does compacts the free heap space on the go just after reclaiming the memory. But CMS garbage collector compacts the memory on stop the world (STW) situations. G1 collector prioritizes the region based on most garbage first.