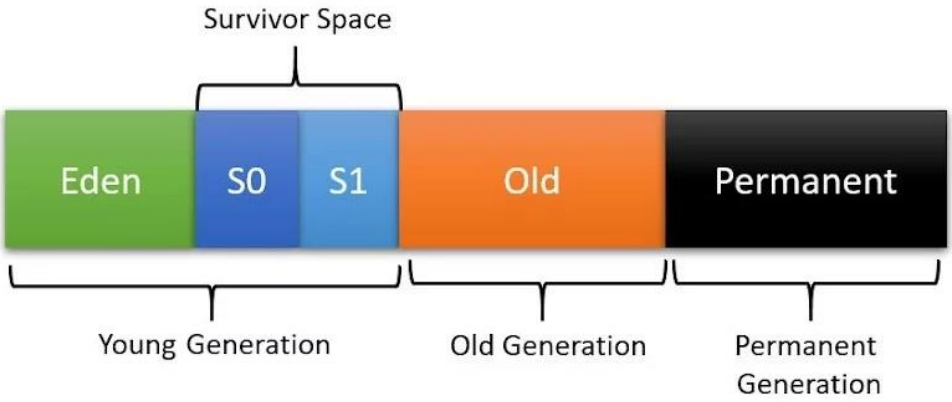
Memory management:

In Java, memory management is the process of allocation and de-allocation of objects, called Memory management.

Java memory management divides into two major parts:

- **JVM Memory Structure**
- **Working of the Garbage Collector**

JVM Memory Structure:



In JVM Memory Structure have two separate parts:

- Heap
- Permanent Generation

Heap Memory:

- Heap stores the actual objects. It creates when the JVM starts up.
- It can be of fixed or dynamic size.
- When you use a new keyword, the JVM creates an instance for the object in a heap. While the reference of that object stores in the stack.
- There exists only one heap for each running JVM process. When heap becomes full, the garbage is collected.

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

Young Generation:

- The young generation is the place where all the new objects are created.
- When the 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.

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 a longer time.

Memory pool:

- Memory pool is used to store immutable objects: String Class and String Pool
- String uses a special memory location to reuse of String objects called String Constant Pool.
- String objects created without the use of new keyword are stored in the String Constant Pool part of the heap.

Permanent Generation/ Meta Space:

- 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.

Method Area

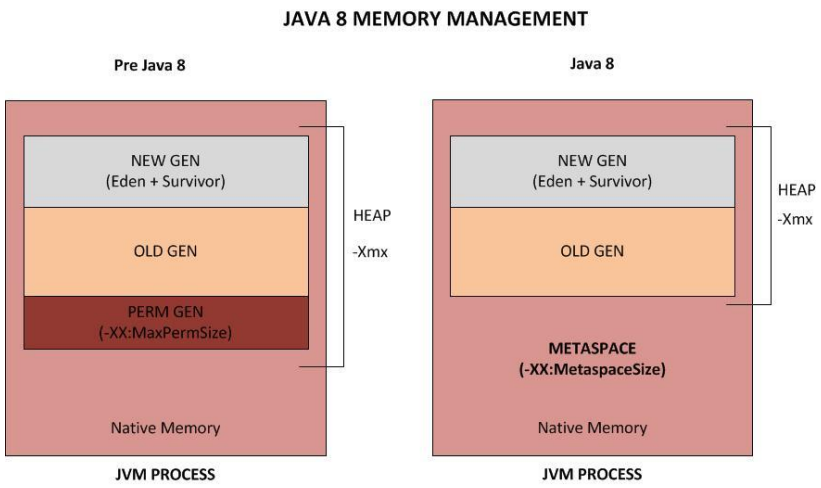
- Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

Runtime Constant Pool

- Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods.
- Runtime constant pool is part of the method area.

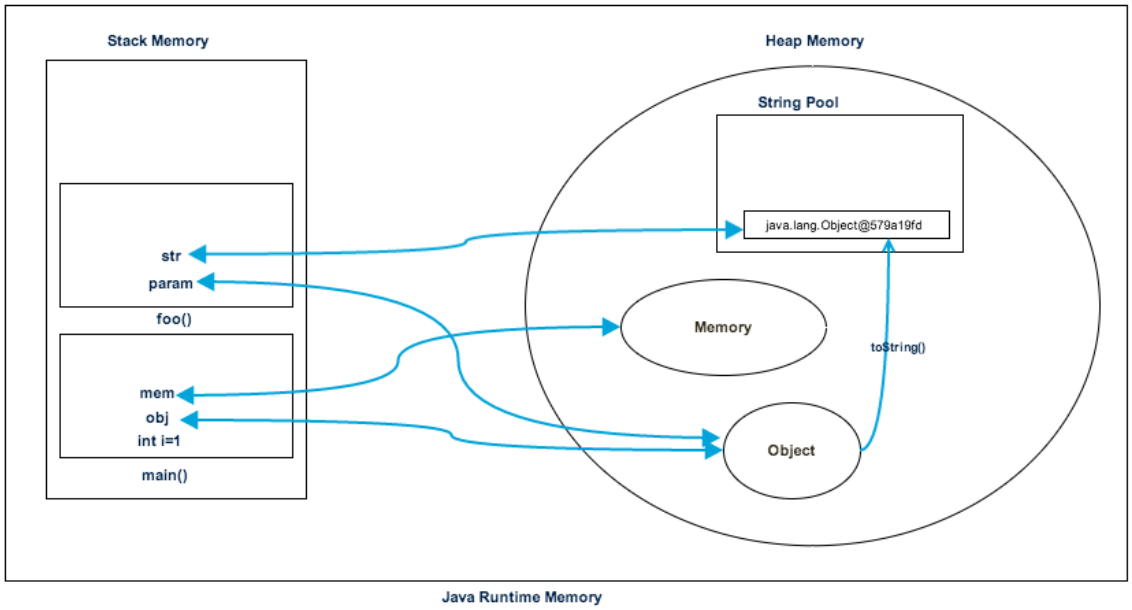
Java Stack Memory

- 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 is getting referred from the method.



Example of Memory Management:

```
public class Memory {
      public static void main(String[] args) {                    // Line 1
            int i=1;                                              // Line 2
             Object obj = new Object();                           // Line 3
             Memory mem = new Memory();                           // Line 4
             mem.foo(obj);                                        // Line 5
 }                                                                // Line 9
 private void foo(Object param) {                                 // Line 6
       String str = param.toString();                             // Line 7
       System.out.println(str); }                                 // Line 8
 }
```

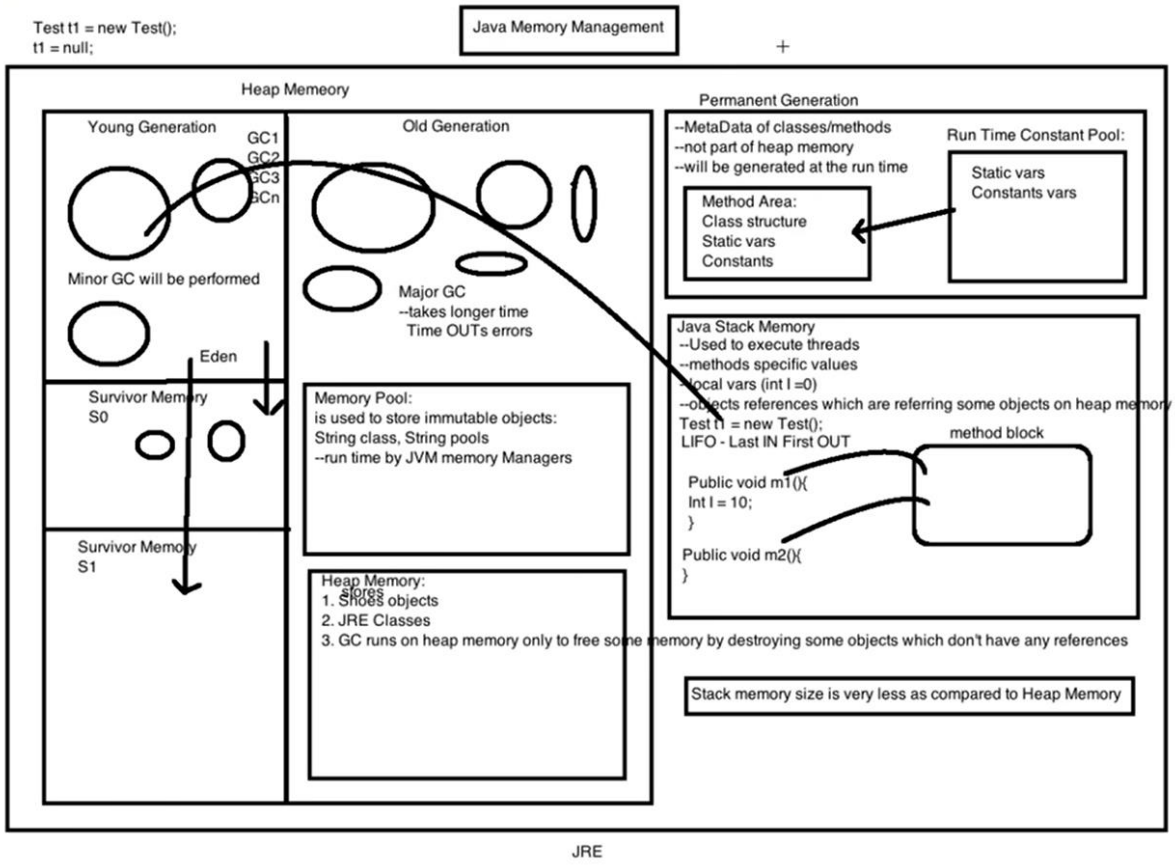


- As soon as we run the program, it loads all the Runtime classes into the Heap space. When the **main() method is found at line 1**, Java Runtime **creates stack memory** to be used by main() method thread.
- We are creating primitive **local variable at line 2**, so it's created and **stored in the stack** memory of main() method.
- Since we are **creating an Object in the 3rd line**, it's **created in heap memory** and **stack memory contains the reference** for it. A similar process occurs when we create Memory object in the 4th line.

- Now when we **call the foo() method** in the 5th line, a block in the **top of the stack is created to be used by the foo() method**. Since Java is pass-by-value, a new reference to Object is created in the foo() stack block in the 6th line.
- A **string is created in the 7th line**, it goes in the **String Pool in the heap space** and a reference is created in the foo() stack space for it.
- **foo() method is terminated** in the 8th line, at this time **memory block allocated for foo() in stack becomes free.**
- In line 9, main() method terminates and the stack memory created for main() method is destroyed. Also, the program ends at this line, hence Java Runtime frees all the memory and ends the execution of the program.

For Referenced :

https://www.youtube.com/watch?v=aAjkJW08BGQ



Working of the Garbage Collector:

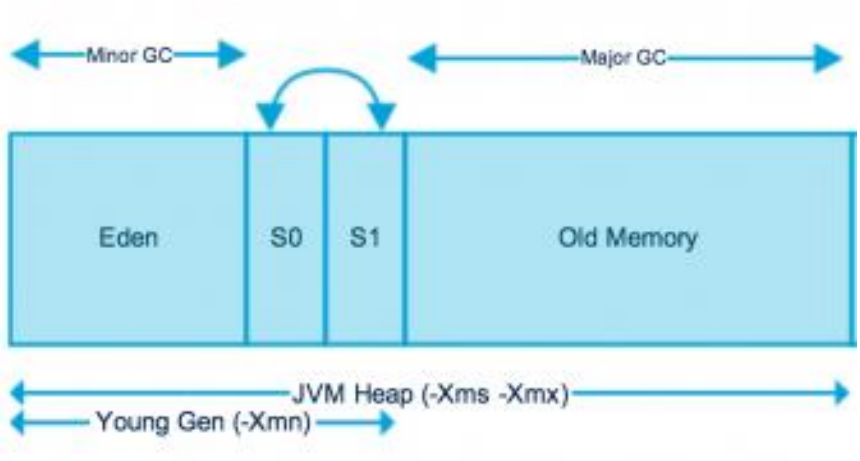
- Java Garbage Collection is the process to identify and remove the unused objects from the memory and free space to be allocated to objects created in future processing.
- One of the best features of Java programming language is the automatic garbage collection
- Garbage Collector is the program running 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.

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.
- **Delete/sweep:** Garbage Collector removes the unused objects.
- **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

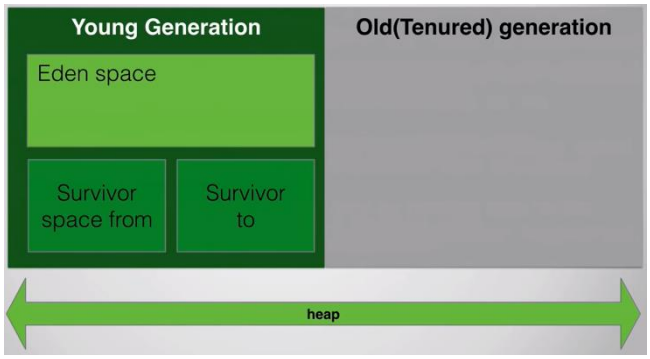
Garbage Collectors runs in the heap memory, they are two types of Garbage Collectors:

- Minor GC performs in Younger Generation
- Major GC performs in Old Generation

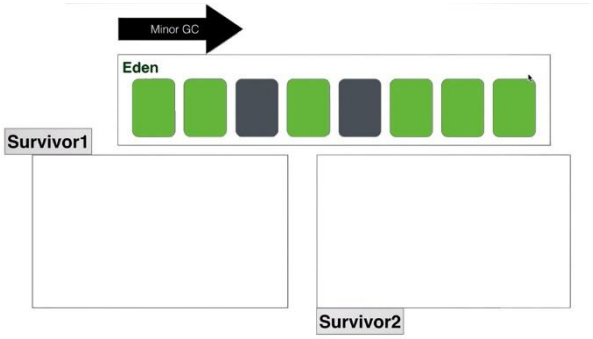


Minor GC:

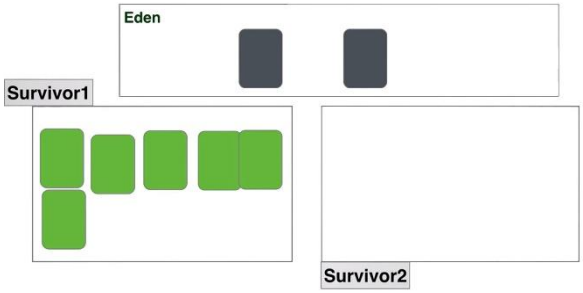
In Younger Generation have the three-part Eden space, Survivor space1, Survivor space2



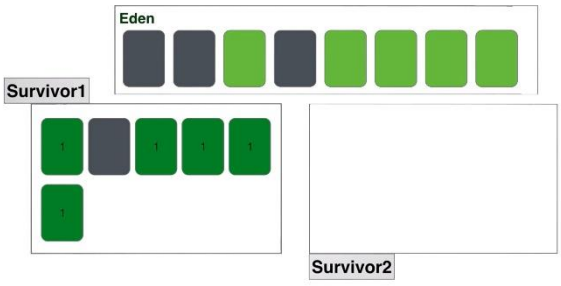
- If the class file has a multiple number of object created in the Eden block in heap, once Eden is get fulled automatically Minor GC will performed. Then all the survivor object are relocated to survivor1.



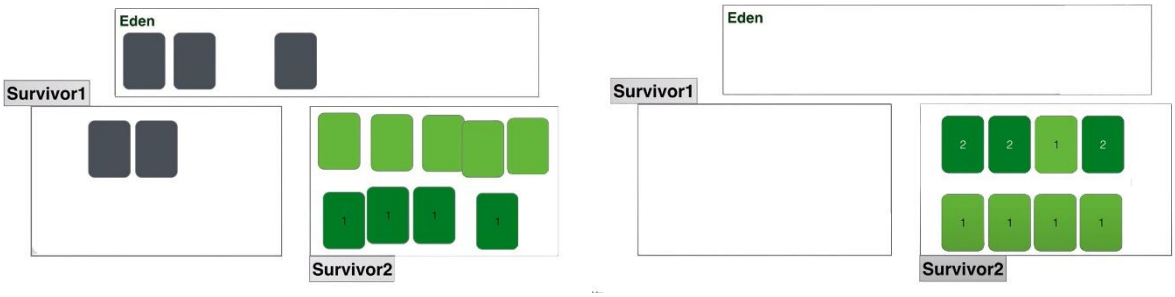
- After that minor GC remove the unused objects from the Eden memory



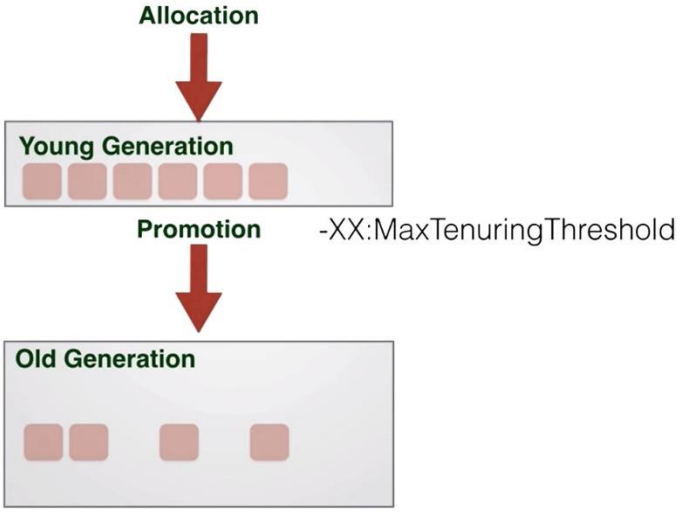
- If again created lot of object, minor GC will perfom again once eden get fulled. Now its all the referenced or useable object in eden and survivor1 moves to survivor2.



- Then remove the unused object in eden and survivor1 has to delete/sweep, this process repeate until Yonger Gerneration in heap gets full.



- If all the Yonger Gerneration of object gets full, now all the referenced or useable object moved to Old Generation



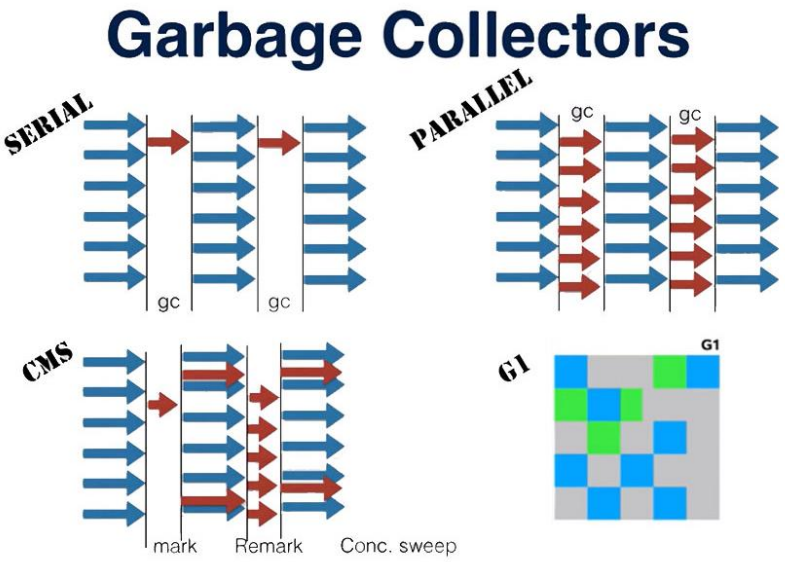
- Old Generation memory contains the objects that are long-lived and survived after many rounds of Minor GC.

- Usually, garbage collection is performed in Old Generation memory when it's full.

- Old Generation Garbage Collection is called Major GC and usually takes a longer time.

The traditional Oracle Hotspot JVM has four ways of performing the GC activity:

- **Serial** where just one thread executed the GC
- **Parallel** where multiple minor threads are executed simultaneously each executing a part of GC
- **Concurrent Mark Sweep (CMS),** which is similar to parallel, also allows the execution of some application threads and reduces the frequency of stop-the-world GC
- **G1** which is also run in parallel and concurrently but functions differently than CMS



- **Serial GC (-XX:+UseSerialGC):** Serial GC is useful in client machines such as our simple stand-alone applications and machines with smaller CPU.
- **Parallel GC (-XX:+UseParallelGC):** 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.
- **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 the Old generation. CMS collector on the young generation uses the same algorithm as that of the parallel collector.
- **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.

Selecting Garbage collector

Option	Description
-XX:+UseSerialGC	Single-threaded gc on young and old generation. To be used only on small heaps
-XX:+UseParallelGC	Young generation uses parallel gc, Old generation uses single-threaded gc
-XX:+UseParallelOldGC	Both young and old generations have multi-threaded GC
-XX:+UseParNewGC	Multi-threaded young generation garbage collector
-XX:+UseConcMarkSweepGC	Enables concurrent collector. Autoenables ParNewGC by default.
-XX:+UseG1GC	Use G1

Defaults: 1.6 = Parallel, 1.7 G1

Java Heap Memory Switches

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.