**Heap Memory**

* **Class instances and arrays are stored in heap memory**. Heap memory is also called shared memory. As this is the place where multiple threads will share the same data.

**Non-Heap Memory**

* It **comprises of Method Area** and other memory required for internal processing. It **stores runtime constants, static fields, code for methods and constructors**.

**Memory Pool**

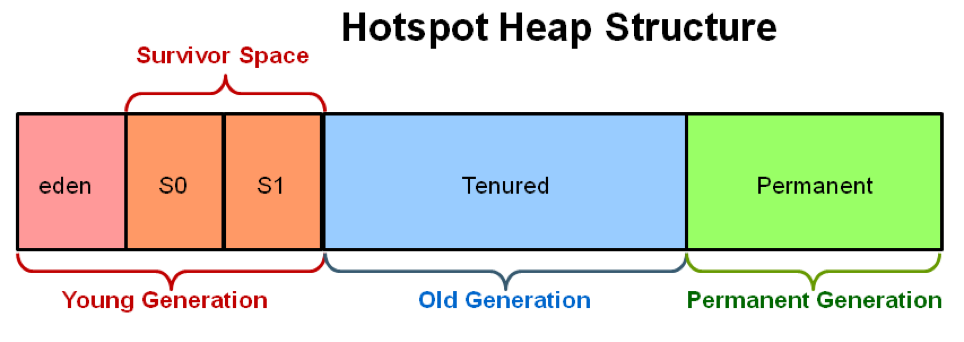
* **Memory pools are created by JVM memory managers during runtime**. Memory pool may belong to either heap or non-heap memory.

**Runtime Constant Pool**

* **A runtime constant pool is a per-class or per-interface run time representation of the constant\_pool table in a class file**. Each runtime constant pool is allocated from the Java virtual machines method area.

**Java Stacks and Frames**

* **Java stacks are created private to a thread. Every thread will have a program counter (PC) and a Java stack**. PC will use the java stack to store the intermediate values, dynamic linking, return values for methods and dispatch exceptions. This is used in the place of registers.



The **Heap is divided into young and old generations** as follows:

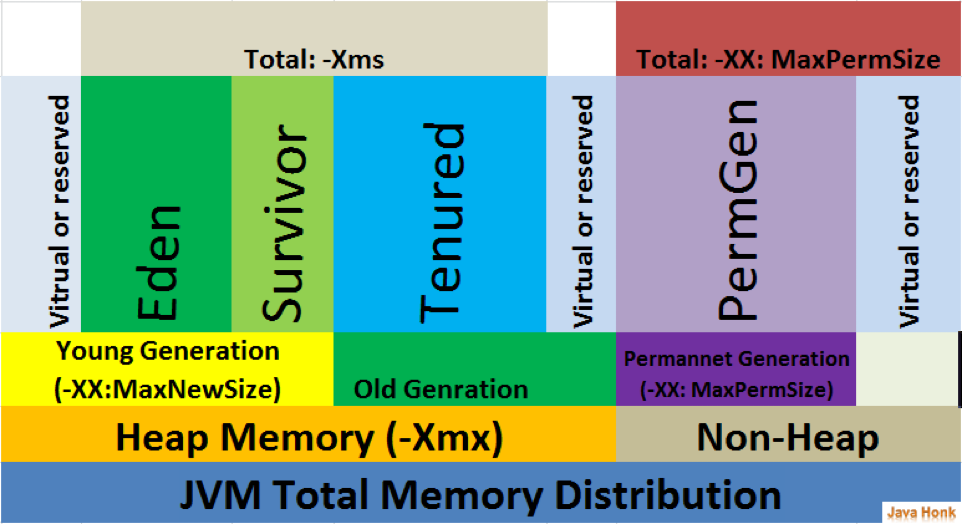
* **Young Generation**:
  + It is a place where **objects lived for a short period** and is divided into two spaces:
    - **Eden Space**: When **object created using new** keyword memory allocated on this space.
    - **Survivor Space**: This is the pool that **contains objects which have survived after java garbage collection** **from Eden space**.
* **Old Generation**:
  + This pool basically **contains tenured and virtual (reserved) space** and will be holding those **objects which survived after garbage collection from Young Generation**.
    - **Tenured Space**: This memory pool **contains objects which survived after multiple garbage collections**, means objects which survived after garbage collections from Survivor space.

**Permanent Generation**:

* This memory pool as the name also says **contains permanent class metadata and descriptors information** so PermGen space is always **reserved for classes** and **those that are tied to the classes**, for example, **static members**.
* **Java8 Update**:
  + **PermGen** is **replaced with Metaspace** which is very similar.
  + The **main difference is that Metaspace re-sizes dynamically** i.e., **it can expand at runtime**.
  + Java Metaspace space: **unbounded** (default)

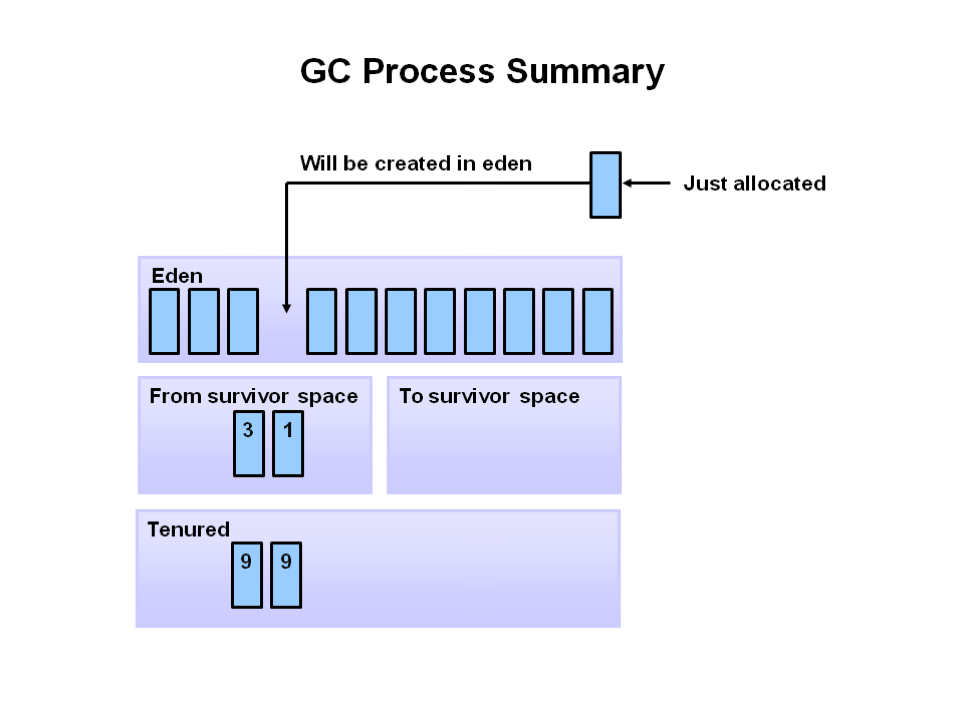
**Code Cache (Virtual or reserved)**:

* If you are using HotSpot JVM, this includes a **code cache area** that **contains memory that will be used for compilation and storage of native code**.



**Method: protected void finalize() throws Throwable { }**

* **Called by the garbage collector on an object** when garbage collection determines that **there are no more references to the object**. A **subclass overrides the finalize() to dispose of system resources** or to perform other cleanups.
* The general contract of finalize() is that it is invoked if and when the JVM has determined that there is **no longer any means by which this object can be accessed by any thread** that has not yet died, except as a result of an action taken by the finalization of some other object or class which is ready to be finalized.
* The Java programming language **does not guarantee which thread will invoke the finalize() for any given object**. It is guaranteed, however, that the thread that invokes finalize will not be holding any user-visible synchronization locks when finalize is invoked. If an uncaught exception is thrown by the finalize method, the exception is ignored and the finalization of that object terminates. The **finalize() is never invoked more than once by a Java virtual machine for any given object**.



**Garbage Collectors**

* **Serial collector**
  + The **serial collector uses a single thread to perform all garbage collection work**, which makes it **relatively efficient since there is no communication overhead between threads**. It is **best-suited to single-processor machines** since it cannot take advantage of multiprocessor hardware. The serial collector is explicitly enabled with the option **-XX:+UseSerialGC**.
* **Parallel collector**
  + The **parallel collector (also known as the throughput collector) performs minor collections in parallel,** which can significantly reduce garbage collection overhead. It is **intended for applications with medium-to-large-sized data sets that are run on multiprocessor or multi-threaded hardware**. The parallel collector is explicitly enabled with the option **-XX:+UseParallelGC**.
* **Concurrent collector**
  + The **concurrent collector performs most of its work concurrently** (i.e., while the application is still running) **to keep garbage collection pauses short**. It is **designed for applications with medium-to-large-sized data sets for which response time is more important than overall throughput** since the **techniques used to minimize pauses can reduce application performance**. The concurrent collector is enabled with the option **-XX:+UseConcMarkSweepGC**.
* **Garbage first collector**
  + The **garbage first collector is the low-pause, server-style generational garbage collector**. The **G1 GC uses concurrent and parallel phases to achieve its target pause time and to maintain good throughput**. When G1 GC determines that a garbage collection is necessary, it **collects the regions with the least live data first** (garbage first). The garbage first collector is enabled with the option **-XX:+UseG1GC**.

The **G1 GC achieves automatic memory management** through the following operations:

* **Allocating objects to a young generation** and **promoting aged objects into an old generation**.
* **Finding live objects in the old generation through a concurrent (parallel) marking phase**. The JVM triggers the marking phase when the total Java heap occupancy exceeds the threshold.
* **Recovering free memory by compacting live objects through parallel copying**.

The **G1 collector is designed for applications** that:

* Can **operate concurrently with applications threads** like CMS.
* **Compact free space without lengthy GC-induced pause times**.
* **Need more predictable GC pause durations**.
* **Do not want to sacrifice a lot of throughput performance**.