Class Loading in Java – 2022

ClassLoader in Java works on three principle: delegation, visibility and uniqueness (deVU).

**Delegation principles**

Suppose you have an application specific class called Abc.class, first request of loading this class will come to Application ClassLoader which will delegate to its parent Extension ClassLoader which further delegates to Primordial or Bootstrap class loader. Primordial will look for that class in **rt.jar** and since that class is not there, request comes to Extension class loader which looks on **jre/lib/ext** directory and tries to locate this class there, if class is found there than Extension class loader will load that class and Application class loader will never load that class but if its not loaded by extension class-loader than Application class loader loads it from Classpath in Java.

**Visibility Principle**

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true.

**Uniqueness Principle**

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again.

#### Linking

1. **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
2. **Prepare** – For all static variables memory will be allocated and assigned with default values.
3. **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

**Types of Class Loaders in Java**

There are three default class loader used in Java, **Bootstrap , Extension and System or Application** class loader. Every class loader has a predefined location, from where they loads class files. **Bootstrap ClassLoader is responsible for loading standard JDK class files from rt.jar** and it is parent of all class loaders in Java. Bootstrap class loader is also known as Primordial ClassLoader in Java. **Extension ClassLoader delegates class loading request to its parent, Bootstrap and if unsuccessful, loads class form jre/lib/ext directory** or any other directory **pointed by java.ext.dirs** system property. **Third default class loader used by JVM to load Java classes is called System or Application class loader and it is responsible for loading application specific classes from CLASSPATH environment variable**, -classpath or -cp command line option, Class-Path attribute of Manifest file inside JAR. Application class loader is a child of Extension ClassLoader and its implemented by sun.misc.Launcher$AppClassLoader class.

**JVM Architecture**

|  |
| --- |
| [http://2.bp.blogspot.com/-4g8GW68TQy4/T0J4DOqkE1I/AAAAAAAAJGE/k62CUFwPtRc/s640/JVM-arc1.png](http://2.bp.blogspot.com/-4g8GW68TQy4/T0J4DOqkE1I/AAAAAAAAJGE/k62CUFwPtRc/s1600/JVM-arc1.png) |
|  |

JVM has various subcomponents internally. You can see all of them from the above diagram.  
  
**1. Class loader sub system:** JVM's class loader sub system performs 3 tasks  
      a. It loads .class file into memory.  
      b. It verifies byte code instructions.  
      c. It allots memory required for the program.  
  
**2. Run time data area:** This is the memory resource used by JVM and it is divided into 5 parts  
      **a. Method area:** Method area stores class code and method code.  
      **b. Heap:** Objects are created on heap.  
      **c. Java stacks:** Java stacks are the places where the Java methods are executed. A Java stack contains frames. On each frame, a separate method is executed.  
      **d. Program counter registers:** The program counter registers store memory address of the instruction to be executed by the microprocessor.  
      **e. Native method stacks:** The native method stacks are places where native methods (for example, C language programs) are executed. Native method is a function, which is written in another language other than Java.  
  
**3. Native method interface:** Native method interface is a program that connects native methods libraries (C header files) with JVM for executing native methods.  
  
**4. Native method library:** holds the native libraries information.  
  
**5. Execution engine:** Execution engine contains interpreter and JIT compiler, which covert byte code into machine code. JVM uses optimization technique to decide which part to be interpreted and which part to be used with JIT compiler. The HotSpot represent the block of code executed by JIT compiler.

**Java (JVM) Memory Structure**

JVM memory is divided into multiple parts: Heap Memory, Non-Heap Memory, and Other.



### ****Heap Memory****

Heap memory is the run time data area from which the memory for all java class instances and arrays is allocated. The heap is created when the Java Virtual Machine starts up and may increase or decrease in size while the application runs. The size of the heap can be specified using –Xms VM option. The heap can be of fixed size or variable size depending on the garbage collection strategy. Maximum heap size can be set using –Xmx option. By default, the maximum heap size is set to 64 MB.

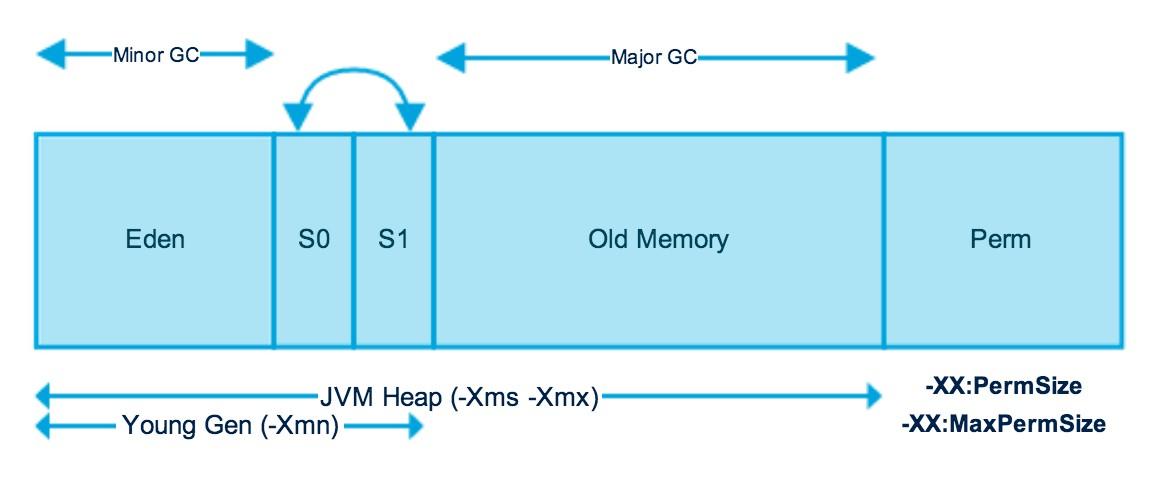
### ****Non-Heap Memory****

The Java Virtual Machine has memory other than the heap, referred to as Non-Heap Memory. It is created at the JVM startup and **stores per-class structures such as runtime constant pool, field and method data, and the code for methods and constructors, as well as interned Strings.** The default maximum size of non-heap memory is 64 MB. This can be changed using –XX:MaxPermSize VM option.

### ****Other Memory****

**Java Virtual Machine uses this space to store the JVM code itself, JVM internal structures, loaded profiler agent code, and data**, etc.

### ****Java (JVM) Heap Memory Structure****



The JVM heap is physically divided into two parts (or generations): nursery (or young space/young generation) and old space (or old generation).

The nursery is a part of the heap reserved for the allocation of new objects. When the nursery becomes full, garbage is collected by running a special young collection, where all the objects that have lived long enough in the nursery are promoted (moved) to the old space, thus freeing up the nursery for more object allocation. This garbage collection is called **Minor GC**. The nursery is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

Important points about the nursery space:

* 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 space is always empty
* Objects that have survived many cycles of GC, are moved to the old generation memory space. Usually, it is done by setting a threshold for the age of the nursery objects before they become eligible to promote to the old generation

When the old generation becomes full, garbage is collected there and the process is called as old collection. 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 as **Major GC** and usually takes longer.

### Permanent Generation (Replaced by Metaspace since Java 8)

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. 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.

### ****Metaspace****

With Java 8, there is no Perm Gen, which means there is no more “java.lang.OutOfMemoryError: PermGen” space problems. Unlike Perm Gen which resides in the Java heap, Metaspace is not part of the heap. Most allocations of the class metadata are now allocated out of native memory. Metaspace by default auto increases its size (up to what the underlying OS provides), while Perm Gen always has fixed maximum size. Two new flags can be used to set the size of the metaspace, they are: “**-XX:MetaspaceSize**” and “**-XX:MaxMetaspaceSize**”. The theme behind the Metaspace is that the lifetime of classes and their metadata matches the lifetime of the classloaders.

## **Metaspace**

Simply put, Metaspace is a new memory space – starting from the Java 8 version; **it has replaced the older PermGen memory space**. The most significant difference is how it handles memory allocation.

Specifically, **this native memory region grows automatically by default**.

We also have new flags to tune the memory:

* MetaspaceSize and MaxMetaspaceSize – we can set the Metaspace upper bounds.
* MinMetaspaceFreeRatio – is the minimum percentage of class metadata capacity free after [**garbage collection**](https://www.baeldung.com/jvm-garbage-collectors)
* MaxMetaspaceFreeRatio – is the maximum percentage of class metadata capacity free after a garbage collection to avoid a reduction in the amount of space