Java Memory Management

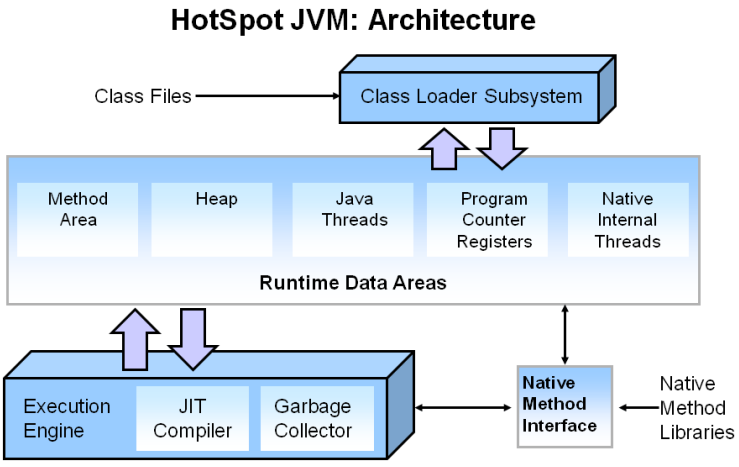
By

Praveen Oruganti

Blog: https://praveenoruganti.blogspot.com

Facebook Group: https://www.facebook.com/groups/2340886582907696/

JVM Architecture



JVM Class loader subsytem

Classloader subsystem does the below steps.

1. Loading
2. Linking
3. Initialization

How many types of class loaders available?

1. Bootstrap Classloader -> jre/lib/rt.jar
2. Extenstions Classloader -> jre/lib/ext
3. System Classloader -> -classpath or -cp

How does classloader works?

1.When JVM requests for a class, it invokes loadClass function of the ClassLoader by passing the fully classified name of the Class.
2.The loadClass function calls for findLoadedClass() method to check that the class has been already loaded or not. It's required to avoid loading the class multiple times.
3.If the Class is not already loaded then it will delegate the request to parent ClassLoader to load the class.
4. If the parent ClassLoader is not finding the Class then it will invoke findClass() method to look for the classes in the file system.

For Linking, it performs the below steps
a) Verify -> looks the byte code and checks whether Compatible with JVM
b) Prepare -> Memory is allocated to class variables(static)
c) Resolve -> Symbolic references will be resolved and will throw ClassNotFound Error if not available

Initialization
Initialize the class variables to proper values. Execute the static initializers and initialize the static fields to the configured values.

JVM Runtime Data Areas

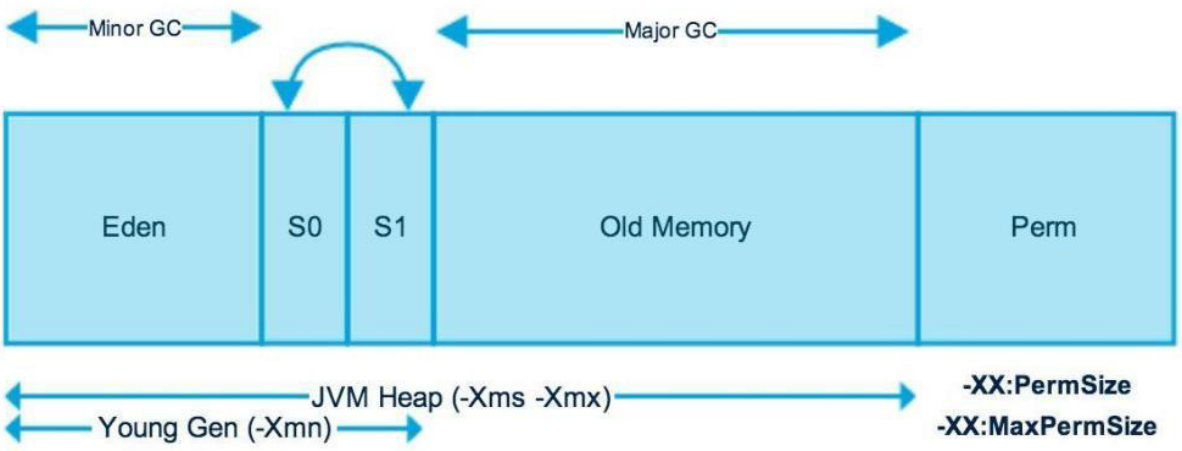
1. Stack Area
2. Heap Area
3. Permanent Generation (Replaced by Metaspace since Java 8)
4. PC Registers
5. Native method stack

Stack Area
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.
Each Thread has its own stack. **All local variables & function calls are stored in stack**. It's life depends upon Thread's life as thread will be alive it will also and vice-versa. It can also be increased by manually:-
java -Xss=512M "Class Name"

When stack memory is full, Java runtime throws java.lang.StackOverFlowError

Heap Area
Heap Memory is created by JVM in start of program and used for storing objects. Heap Memory can be accessed by any thread is further divided into three generations Young Generation,Old & PermGen(Permanent Generation). When object is created then it first go to Young generation(especially Eden space) when objects get old then it moves to Old/tenured Generation. In PermGen space all static & instance variables name-value pairs(name-references for object) are stored. can manually increase heap size by some JVM parameters
java -Xms=1M -XmX=2M "Class Name"



When Heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error

Permanent Generation (also called as Method Area and 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.

PermGen Space used to store 3 things

- Class level data (meta-data)
- interned strings
- static variables

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 a 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.

Memory Pool
Memory Pools are created by JVM memory managers to create pool of immutable objects. Memory Pool can belong to Heap or Perm Gen, depending on JVM memory manager implementation.

Metaspace
With Java 8, there is no Perm Gen, that 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. That is, as long as the classloader is alive, the metadata remains alive in the Metaspace and can't be freed. Please note the interned strings and static variables are moved into the heap itself.

PC Registers
It is also associated by its thread. It basically is a address of current instruction is being executed. Since each thread some sets of method which is going to be executed depends upon PC Register. It has some value for each instruction and undefined for native methods. It is usually for keep tracking of instructions.

Native Method Stack
Native methods are those which are written in languages other than java. JVM implementations cannot load native methods and can't rely on conventional stacks . It is also associated with each thread. In short it same as stack but it is used for native methods.

JVM Execution Engine

The bytecode that is assigned to the runtime data areas in the JVM via class loader is executed by the execution engine. The execution engine reads the Java Bytecode in the unit of instruction. It is like a CPU executing the machine command one by one. Each command of the bytecode consists of a 1-byte OpCode and additional Operand. The execution engine gets one OpCode and execute task with the Operand, and then executes the next OpCode.

But the Java Bytecode is written in a language that a human can understand, rather than in the language that the machine directly executes. Therefore, the execution engine must change the bytecode to the language that can be executed by the machine in the JVM. The bytecode can be changed to the suitable language in one of two ways.

Interpreter: Reads, interprets and executes the bytecode instructions one by one. As it interprets and executes instructions one by one, it can quickly interpret one bytecode, but slowly executes the interpreted result. This is the disadvantage of the interpret language. The 'language' called Bytecode basically runs like an interpreter.

JIT (Just-In-Time) compiler: The JIT compiler has been introduced to compensate for the disadvantages of the interpreter. The execution engine runs as an interpreter first, and at the appropriate time, the JIT compiler compiles the entire bytecode to change it to native code. After that, the execution engine no longer interprets the method, but directly executes using native code. Execution in native code is much faster than interpreting instructions one by one. The compiled code can be executed quickly since the native code is stored in the cache.
However, it takes more time for JIT compiler to compile the code than for the interpreter to interpret the code one by one. Therefore, if the code is to be executed just once, it is better to interpret it instead of compiling. Therefore, the JVMs that use the JIT compiler internally check how frequently the method is executed and compile the method only when the frequency is higher than a certain level.

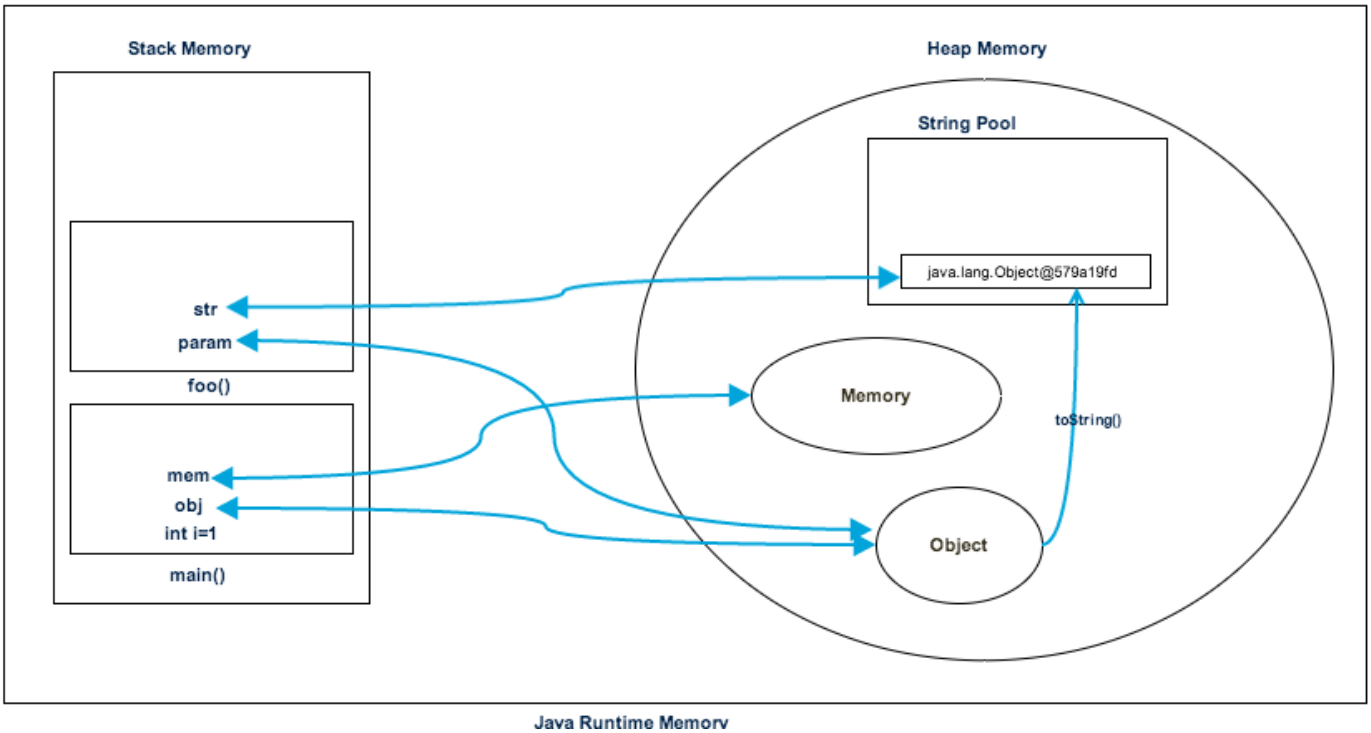
Let's see Stack and Heap diagram for better understanding with a simple example.

```
package com.praveen;

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
```

}

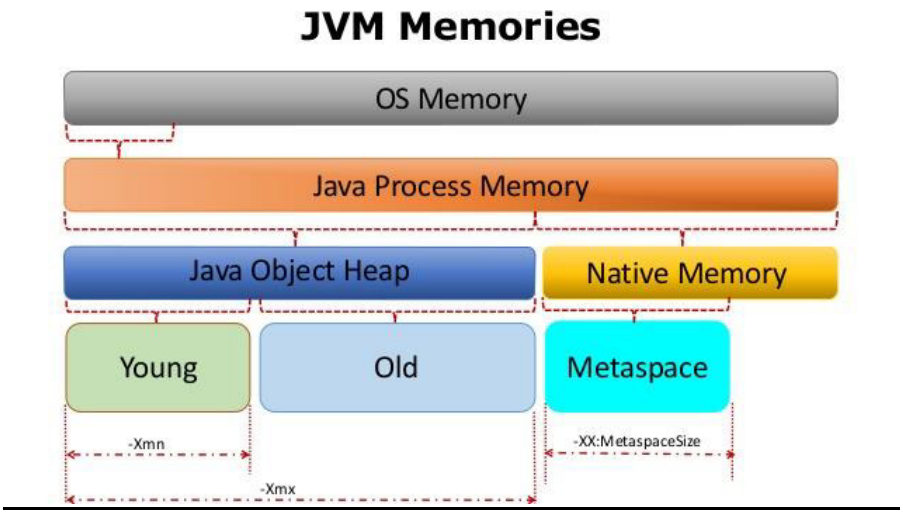
Let's go through the steps of execution of the program.

- As soon as we run the program, it loads all the Runtime classes into the Heap space. When 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 line 3, it's created in Heap memory and stack memory contains the reference for it. Similar process occurs when we create Memory object in line 4.
- Now when we call foo() method in line 5, a block in the top of the stack is created to be used by foo() method. Since Java is pass by value, a new reference to Object is created in the foo() stack block in line 6.
- A string is created in line 7, 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 line 8, 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 end the execution of the program.



Java Runtime Memory

Garbage Collection in Java



Garbage collection is the way of resource management in programming, where unused or not reference object need to be identified and removed from heap memory.
GC removes the objects in Heap if there no references to stack.

GC is daemon or low priority thread which runs in the background.

Heap memory is divided into three generation i.e... Young Generation, old or tenured generation and permanent generation.

Young - New
Old - long lasting
Perm Gen - Metadata

Type of garbage collection in Java

Minor GC
In Young generation new objects will be created and after GC all the non reference objects are removed and reference objects are moved to Old or Tenured generation. This is faster process.

Major GC
Garbage collection in old generation is known as Major GC. This is slower process.

Full GC
Removing non reference objects from Young as well old generation.

Types of Garabage collectors in Java

Serial GC
Uses single thread to scan system and designed for small heap size system.All the application threads are paused at the time of Garabage collection.

Parallel/Throughput GC
Uses multiple threads to scan system and default garbage collector for Java 8.All the application threads are paused at the time of Garabage collection.

Concurrent Mark Sweep(CMS) GC
Uses mutliple threads to scan the heap memory to mark and then sweep the objects.Application threads are paused in 2 scenarios
1.Marking the reference object in old generation.
2.Change in heap memory in parallel while in GC

G1 Garbage collector
Uses multiple threads to scan the system and default garbage collector for Java9 and used for large heap memory size system. It divides the heap memory into regions from 1MB to 32 MB.There is a concurrent global marking phase which marks all the reference objects in heap. After that G1 will start collection object with more empty and release them to get free space.G1 also uses user defined pause time target on the basis of that it selects the number of regions.

Here with the JVM Garbage collection configuration

-Xms, -Xmx: Places boundaries on the heap size to increase the predictability of garbage collection. The heap size is limited in replica servers so that even Full GCs do not trigger SIP retransmissions. -Xms sets the starting size to prevent pauses caused by heap expansion.
-Xmn : Size of young generation.
-XX:PermSize : Initial PermSize
X:MaxPermSize: Max PermSize
-XX:+UseG1GC: Use the Garbage First (G1) Collector.
-XX:+UseSerialGC : Serial GC
-XX:+UseParallelGC: Parallel GC
-XX:+UseConcMarkSweepGC : CMS GC