**Garbage Collection**

**Why GC in Java?**

Before java, most popular languages were C and C++. In C and C++, you are supposed to manage your own memory.

**In C:** we have methods like **realloc(), malloc(), calloc()**, that would allocate buffer memory for you, have to specify how much memory is needed for your program and say that in those API calls.

At the end of the program, at some logical point you are responsible for free of that memory as well using **free().**

Sometimes you may forget to free the memory, which causes for memory leaks and lot of issues in applications.

Also **in C++:** we use new and destructors to release resources.

So programmers had to take care or worry about whenever they were implementing programs in languages like C and C++.

**Java came with automatic memory management through a program called Garbage Collector.**

**“Remove objects that are not used anymore”.**

**Based of following hypothesis:**

**Most objects soon become unreachable.**

**References from ’old’ objects to ‘young’ objects only exist in small numbers.**

**Live object = reachable (referenced by someone else)**

**Dead object = unreachable (not referenced from anywhere)**

**Before we start GC:**

* Objects are allocated in (eg: new) in the “heap” of java memory. Static members, class definitions (metadata) etc. are stored in “method area” (PermGen/Metaspace)
* GC is carried out by a daemon thread called “Garbage Collector”
* We cannot force gc to happen (System.gc())
* When new allocations cannot happen due to full heap you end up with a java.lang.OutOfMemoryError heap space and lot of other headaches.

**Garbage Collection:**

It involves:

1. **Mark:** Start from root node of an application, walks the object graph, marks object that are reachable as live, unreachable as dead.
2. **Delete/sweep:** Delete the unreachable objects.
3. **Compacting:** Compact the memory by moving around the objects and making the allocation contiguous than fragmented.

**Performance:**

* **Responsiveness/latency:** How quickly an application response with a requested piece of data. Examples: How quickly a desktop UI responds to an event. How fast a website returns a page.

For the applications that focus on responsiveness, large pause times are not acceptable. The focus is on responding in short periods of time.

* **Throughput:** Throughput focus on maximizing the amount of work by an application in a specific period of time. Example: Number of transactions completed in a given time.

**Types of GC:**

1. **A Serial Collector:**  Basic GC that runs in single thread, can be used for basic application.
2. **A concurrent Collector:** A thread that performs GC along with application execution as the application runs does not wait for the old generation to be full. Stops the world only during mark/re-mark.
3. **A parallel Collector:** Uses multiple CPUs to perform GC. Multiple threads doing mark/sweep etc. Does not kick in until heap is full/near-full. “stop-the-world” when it runs.

**When to use concurrent collector (CMS):**

* There is more memory
* There is high number of CPUs
* Application demands short pauses

**When to use parallel collector :**

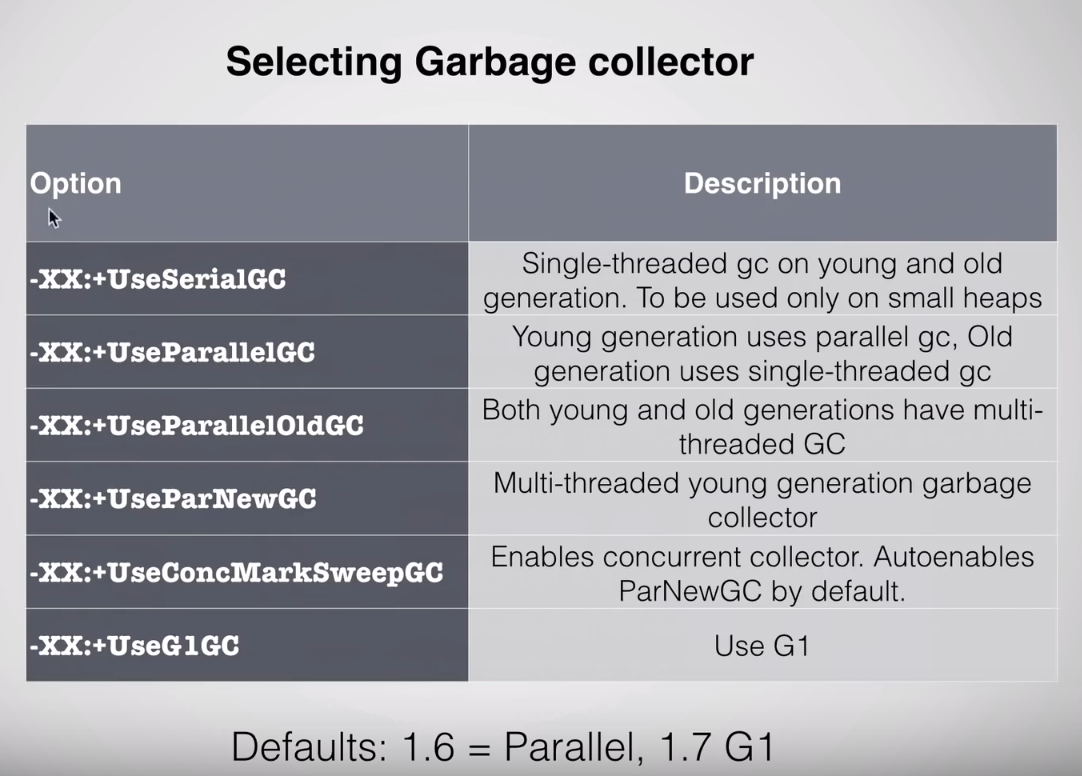
* There is less memory
* There is lesser number of CPUs
* Application demands high throughput and can withstand pauses.

**In Java 1.7, GC1 (Garbage-first) was introduced. Reasons are:**

* More predictable (tunable) GC pause
* Low pauses with fragmentation
* Parallelism and concurrency together
* Better heap utilization

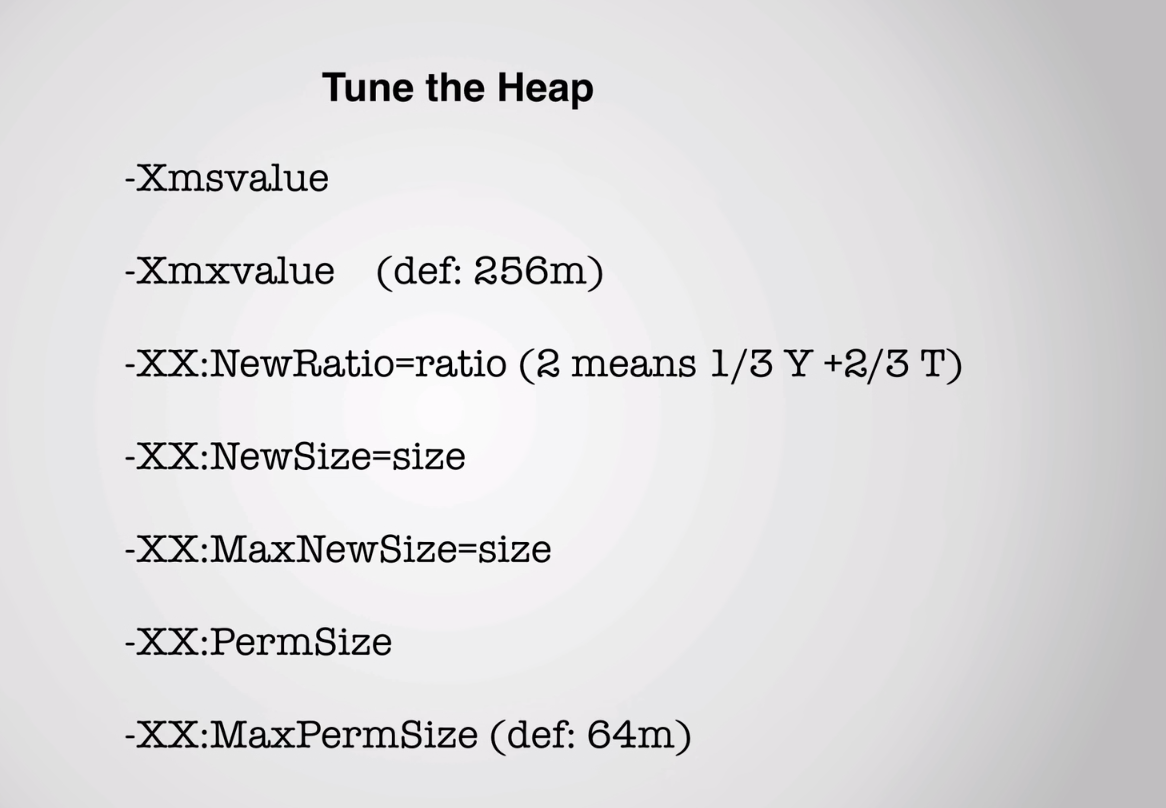


Selecting the Garbage Collector:



**Tune the Heap:**

* -Xmsvalue: Min amount of heap allocated to your program
* -Xmxvalue (default: 256MB): Max amount of heap allocated to your program



**GC logging:**

* -verbose:gc
* -XX:+PrintGCDetails
* Xloggc:gc.log – file name

**JVM (Java Virtual Machine) Architecture**

It is a specification that provides runtime environment in which java byte code can be executed.

Main function of JVM is to load and execute your Application.

Your application is generally a **.class** file.

How do you generate a **.class** file?

Edit MyApp.java file (MyApp.java is your standalone)

javac MyApp.java -- Compiles the java file

java MyApp.java – JVM instance (At the time when you call class through java command you have created JVM instance.)

**How JVM instance load and execute your class file?**

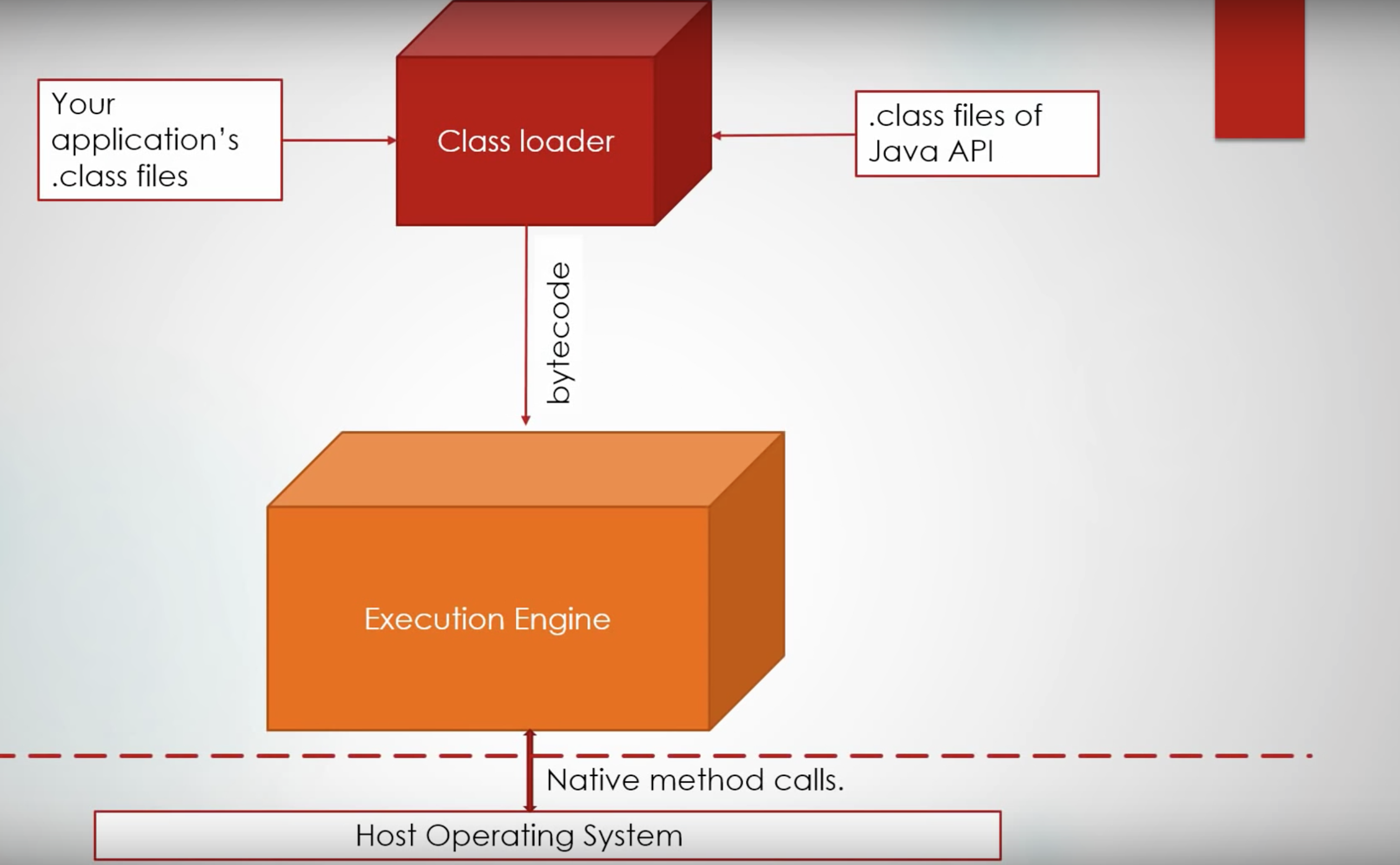
From above example:

Class loader is responsible for loading your Application class files.

Class loader is responsible for loading MyApp.class (From previous example) file and it also has to load built in class files that are part of JVM (Ex: String.class, Object.class, All Collection classes etc.)

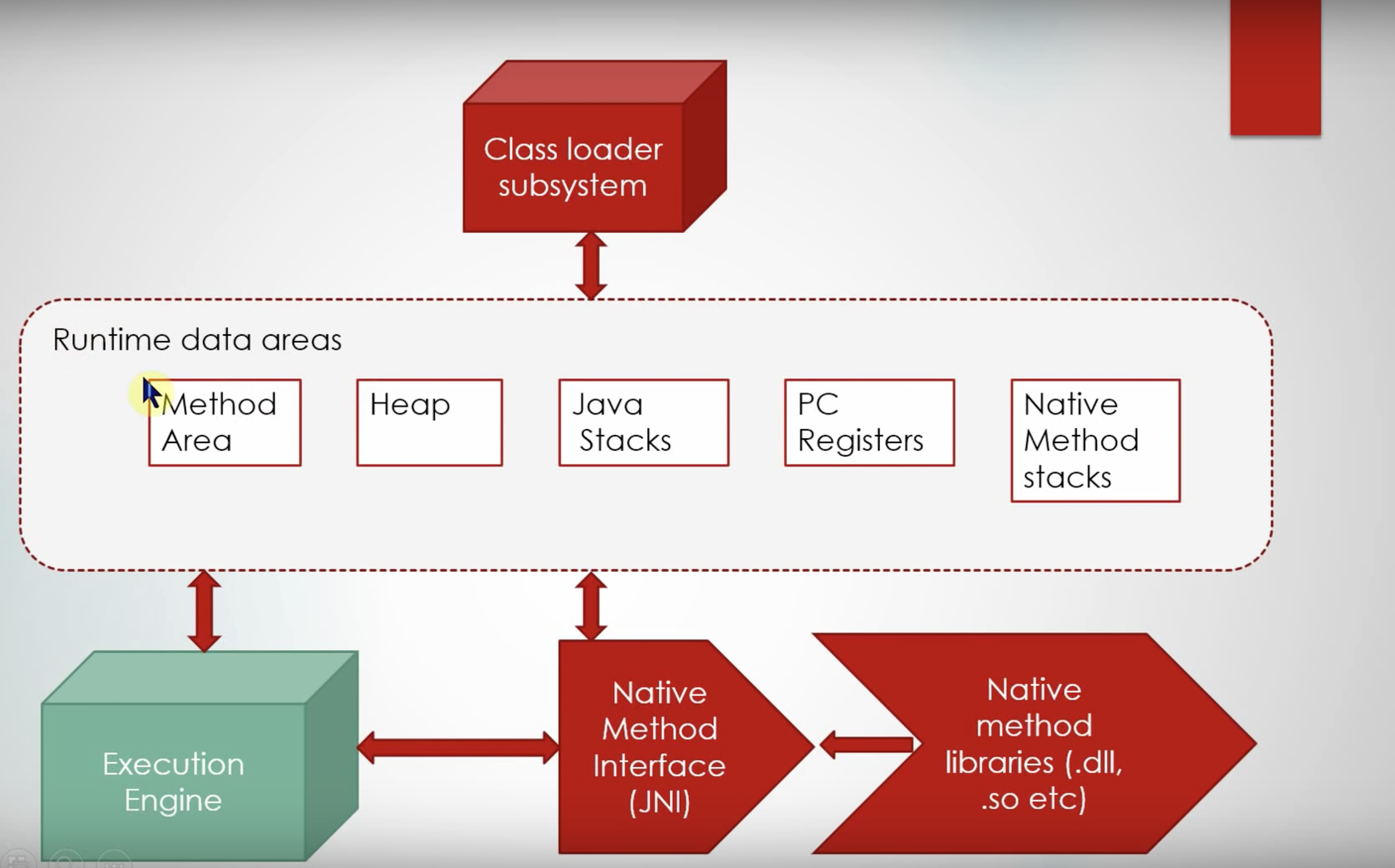
Then, the loader classes, which are basically byte code instructions are fed into Execution Engine. And Execution engine is responsible for executing byte code. In order to execute byte code, Execution engine has to talk to Host Operating System, in order to execute instructions finally against the machine instruction set. For that Execution engines makes use of Native method calls.

When you say class loading, the class’s byte code or the data will be loaded to a memory area. In actual JVM architecture one more component involved which is the DATA AREAs to MEMORY AREAs.



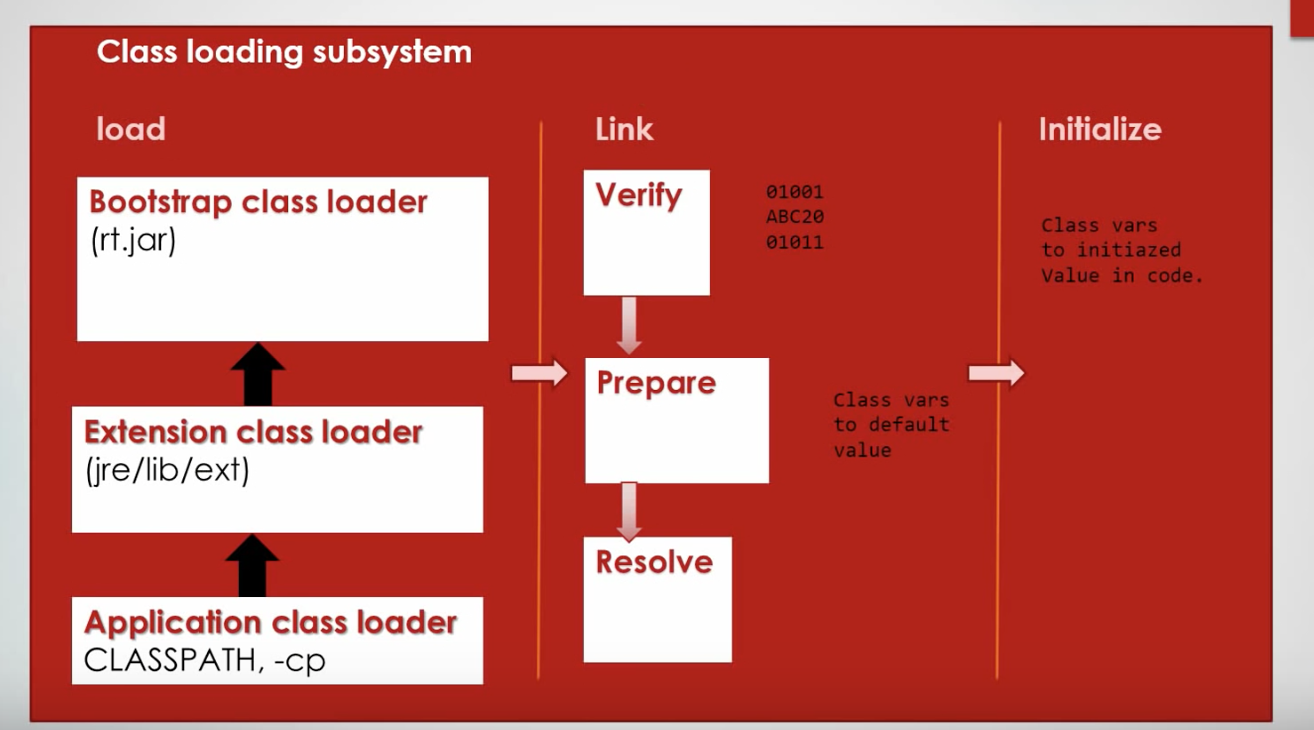
In the below figure, we have the class **loader subsystem, runtime data areas and Execution Engine**. These are the 3 main components of JVM architechture.

Also you can see runtime data areas, you have 5 different data areas mentioned here.



**Class Loader SubSystem:**

Lets zoom into **Class loader subsystem**, how it works



Class loader subsystem has 3 main phases:

1. Load
2. Link
3. Initialize

**Load:**

Load is the part, responsible of loading the bytecode into memory. So loading can load .class bytecode from various sources. In general, when you call MyApp as in the previous example, MyApp would be in .class file. So, class loader would read it from the file system or may read from .jar file which contains .class files. It can also read from various sources like Network socket depending on the class loader implementation.

Load phase involves 3 different type of class loaders.

1. **Bootstrap class loader:** Which is responsible for loading java’s internal classes (Internal classes are inside rt.jar that is distributed with your JVM implementation).
2. **Extension class loader:** Which is responsible of loading additional application jars present in jre/lib/ext folder.
3. **Application class loader:** Which is responsible for loading classes from values specified in your classpath enivronment variable.

**Link:**

Link phase is where lot of work is done. Link phase itself has 3 different phases.

1. **Verify:** Looks at the bytecode, that is loaded by the class loader and checks if that is compatability with the JVM class specifications.
2. **Prepare:** Is the place where memory is allocated for static variables inside class file. Memory for class level static variable is allocated in this phase. Its only memory allocation happening for class variables not for the instance variables. All static variables are assigned with default values in this phase.
3. **Resolve:** Is the place where all the symbolic references inside the current class are resolved. Suppose you have references to other classes, these are changed from symbolic to actual reference.

These are done to virtualize the performance.

**Initialize:**

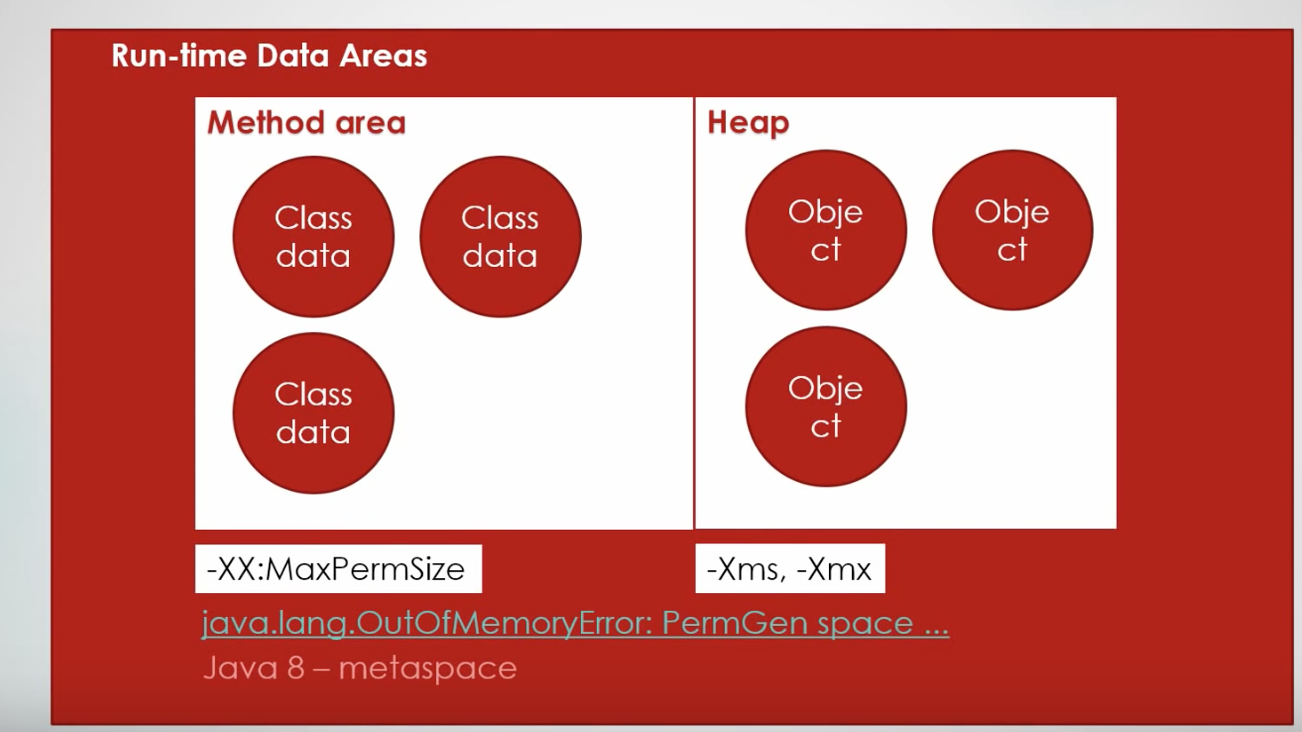
This is the place where static initializers of a class is run(static block).

And where ever you have set values for static variables those values are set into memory location in this phase.

**Note:**

* **ClassNotFoundException** happens when class loader fails to find the bytecode(verify state) correspond to the class name mentioned.
* **ClassDefNotFoundException** happens during resolve state. Suppose I am loading a class X. We went through the loading phase and we went through verify and prepare states, X is referring to Y and in resolve phase the system tries to find the class Y. If class Y cannot be found, then **ClassDefNotFoundException** thrown.

**Runtime Data Areas:**



1. **Method area:**

It is the place where metadata(class data) corresponding to a class is stored. A lot of java API’s of a class requires data in the method area. Static variables, byte code, class level constant pool everything stored in the method area.

Method area a basically memory allocated to JVM i.e., systems physical memory.

Method is called PermGen Space. By default 64MB for method area. This can be tuned, if you have millions of classes using **–XX:MaxPermSize.** If not defined then it will throw **java.lang.OutOfMemoryError:PermGen Space.**

1. **Heap:**

It is a place where Object data is stored. Everytime when you instantiate an object, suppose MyApp app = new MyApp(), that object is created in the Heap. Anything to do with object is created in the Heap. So, all the instance variables, arrays everything are created in Heap. Heap is a very important memory area in java that most of the times to be tuned based on your application needs.

-Xms : Minimum size

-Xmx: Maximum size

By default 1/4th of the physical memory of Xms.

1. **PC Registers:**

Program Counter Register, it contains Program Counter , which is the pointer to the next insturction to be executed per Thread. Suppose, we are running T1, T2, T3 thread as per below figure, So there will be Program counter created for T1 pointing to the next instruction for T1. Similarly for T2 and T3. **This is per Thread.**

1. **Java Stacks:**

Java stacks contains the Stack Frame corresponding to the current method execution per thread. Suppose we have a Thread, that is executing in a 3 methods. Method 1 is calling method 2 so it would push the stack frame into the stack similarly for others. This is per Thread.

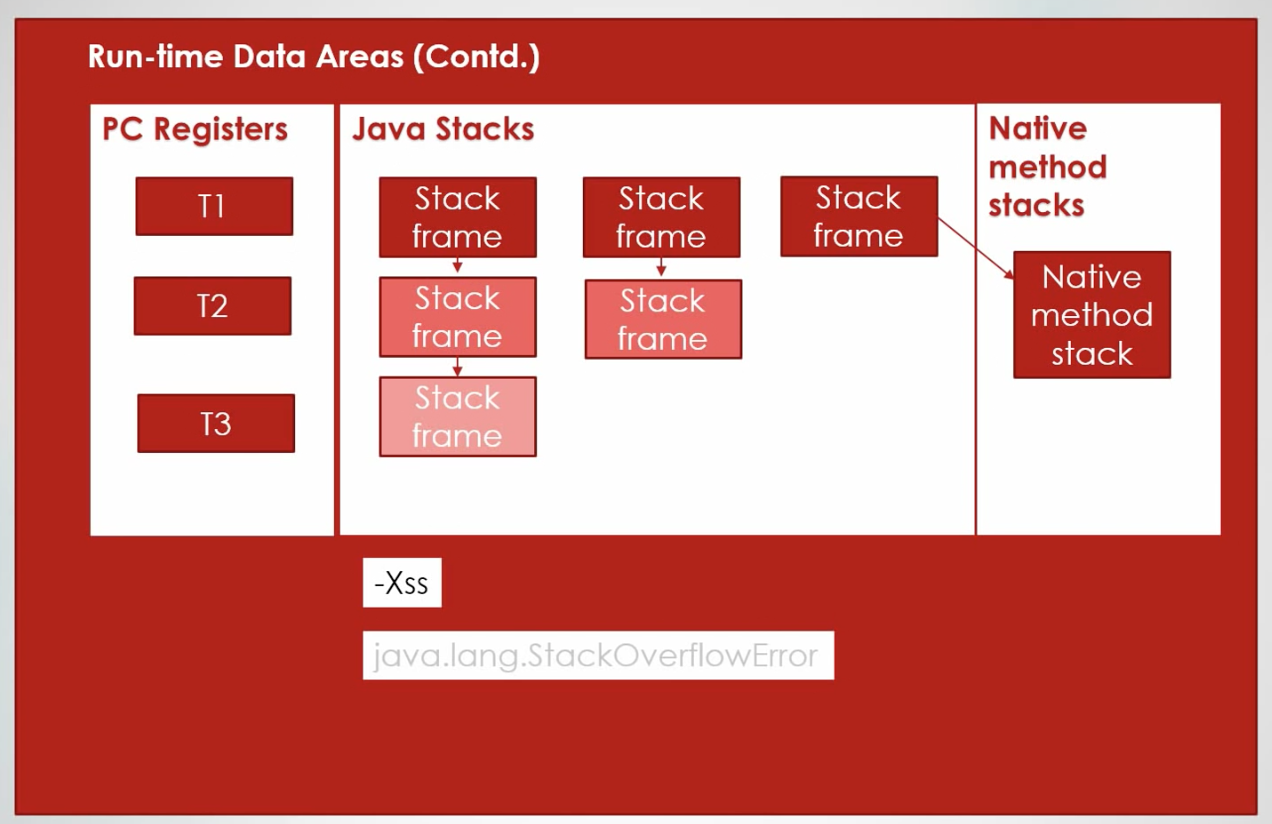
Stack frame contains data related to the currently executing method. Basically, things like local variables within method, memory area for the parameters, parameter arrays.

Per thread Java Stacks created.

Suppose when one of our method is invoking native method, that’s where **Native method stacks** are created.

Ex: Factorial Example, Recursive algorithm, method that call itself. If it goes to infinite loop then Java Stacks area will be full and as a result **java.lang.StackOverflowError.**

This can be tuned by **–Xss.**

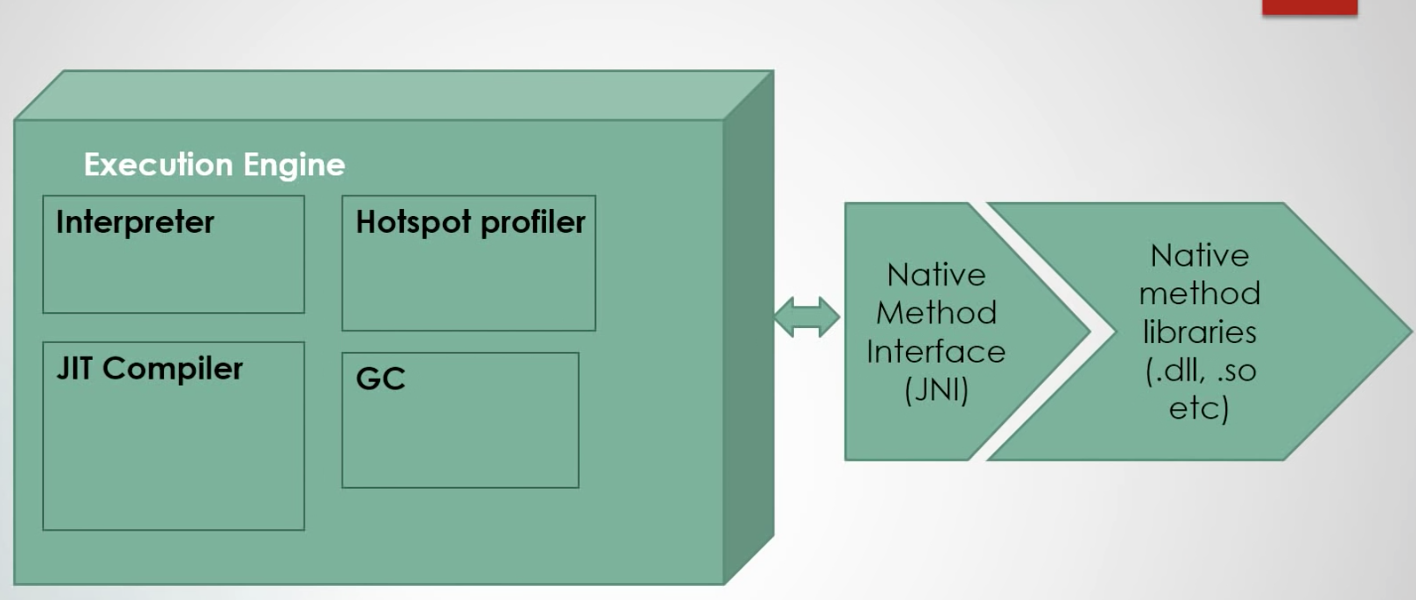
****

So now we talked about 5 different runtime data areas.

* Method area for class data
* Heap for Object data
* PC Registers per Thread for program counters
* Java Stacks per Thread for keeping current executing methods stack frame.

PC Registers, Java Stacks and Native method stacks are Per Thread. These are Thread-safe.

**Execution Engine:**



Once the data area is loaded with the instruction to be executed, the current instruction to be executed is ready, what happens is that the Java Interpreter, interprets the current instruction that is their in the byte code and excutes it.

* **Interpreter:** Reads byte code instruction, looks at it, finds out it what native operation has to be done and executes native operation. That is done by using Native Method Interface(JNI) which interfaces the Native method libraries in the JVM.
* **JIT Compiler:** Suppose we have set of instructions that are getting executed all the time repeatedly, they will not interpret again and again, instead the JIT compiler on the fly compiles the instructions and keeps the target code ready for execution. There is no more interpretion involved here. So, its only machine code execution.

It is used to improve the performance. JIT compiler parts the byte code that have similar functionality at the same time, and hence reduces the amount of the time needed for compilation.

* **Hotspot profiler:** It keeps eye on byte codes that are running and grabs a lot of different statistics from those byte codes that is useful in various formats and various ways can be used. So, that is the role of hotspot profiler. It helps the JIT compiler to find the frequently used instructions.
* **Garbage Collecter:** Which cleans unused classes, objects and memory areas.