**JVM (Java Virtual Machine)**

**Virtual Machine (VM)**

> Software stimulation of a machine which can perform operations like physical machine is called Virtual Machine.

> 2 Categories of VM with respect to Programming:

1. Hardware based VM/ System based VM

2. Application based VM/ Process based VM

1. Hardware based VM: - It provides several logical systems on the same computer with the strong isolation from each other i.e. on one physical machine; we are defining multiple logical machines.

> Main Advantage of H/w based VM is H/w resources utilization & to improve utilization of H/w resources.

For E.g. KVM (Kernel based VM) for Linux system, VMWARE, Cloud Computing etc.

2. Application based VM: - These VM machines act as runtime engines to run a particular programming language application.

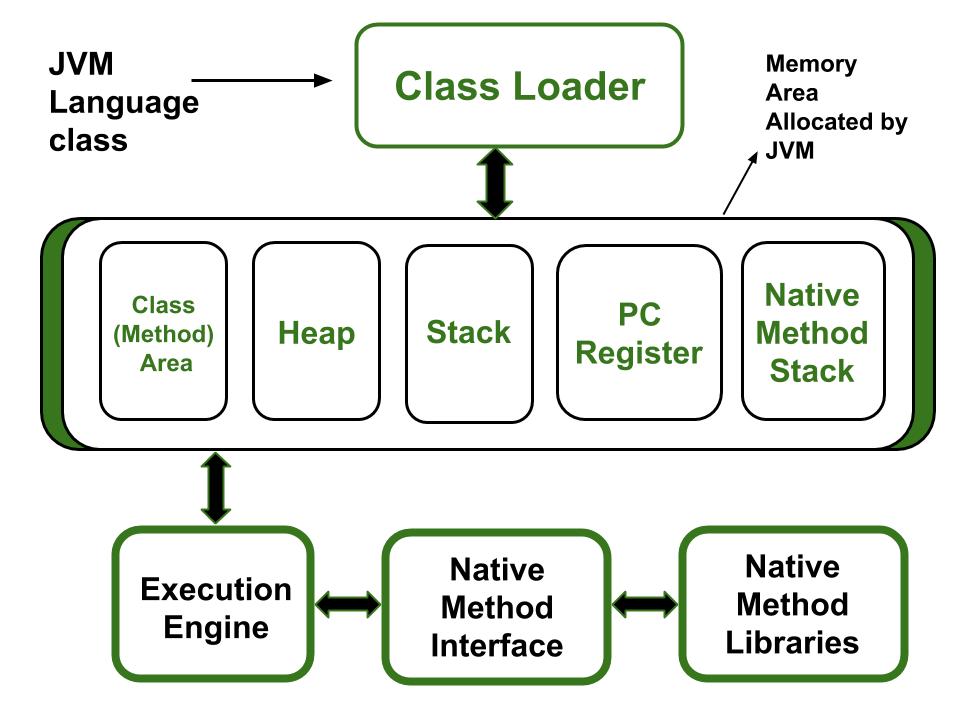
For E.g. JVM acts as runtime engine to run Java based applications.

PVM acts as runtime engine to run Perl based applications.

CLR acts as runtime engine to run .NET based applications.

> JVM is a part of JRE (part of JDK)

.class file Class Loader Subsystem



**Class loader subsystem** reads .class file & store it inside JVM Memory. **Execution Engine** is responsible for reading the .class file & executing it which results in displaying the corresponding output to the console.

**Class Loader Subsystem**

> It is responsible for 3 activities

1. Loading

2. Linking

3. Initialization

1. **Loading**: - Loading means reading class file & store the corresponding binary data in Method Area.

> For each class file, JVM will store corresponding information in the method area like Fully Qualified name of class, immediate parent class, Method information, variable information, constructors info, modifiers info, constant pool info etc.

> After loading .class file, JVM immediately creates an object for that loader class on the heap memory of type **java.lang.Class** (object called as Class class object).

> This Class class object can be used by Programmer to get class level information like method info, variable info or constructor info etc.

> Refer programs from repository.

2. **Linking**: - It mainly contain 3 activities

a) **Verify**: - It is the process of ensuring that binary representation of a class is structurally correct or not (formatted or not) i.e. JVM will check whether the .class file generated by a valid compiler or not.

> Internally, **Bytecode verifier** is responsible for this activity & it is a part of class loader subsystem.

> If verification fails, then we will get runtime exception saying **java.lang.verifyError.**

> **This is why Java is Secure.**

b) **Prepare**: - In this process, JVM will allocate memory to the class level static variables & assign default values (Not original value).

c) **Resolve**: - It is the process of replacing symbolic names in our program with original memory references from method area.

3. **Initialization**: - In this activity, all static variables are assigned with original values & static blocks will be executed from parent to child.

Note:

> While loading, linking & initialization if any error occurs, we will get runtime exception saying **java.lang.linkageError (java.lang.verifyError is child of linkageError).**

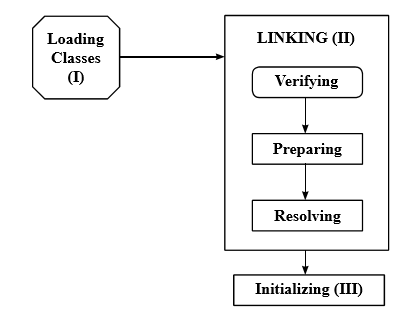


Fig: Class Loader Subsystem

Types of Class Loader Subsystem

1. **Bootstrap class loader**: - It is responsible for loading core java API classes i.e. classes present in **rt.jar** from bootstrap classpath i.e. JDK -> JRE -> lib -> rt.jar

> By default bootstrap class loader is available with every JVM.

> Not implemented in Java (implemented in native language like C/C++)

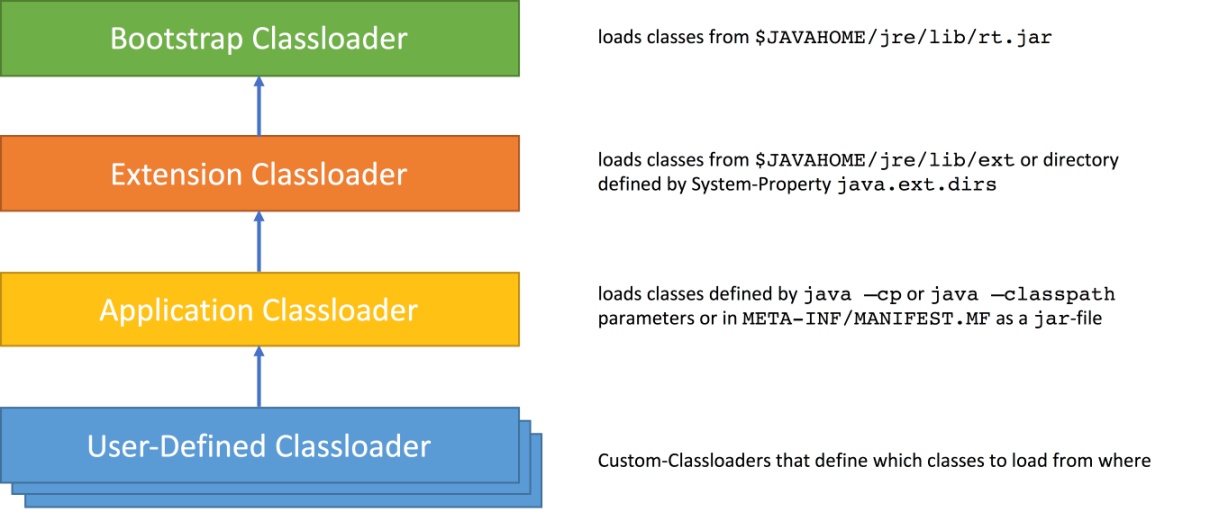
2. **Extension class loader**: - It is the child class of Bootstrap class loader. It is responsible for loading classes from extension classpath i.e. JDK -> JRE -> lib -> ext -> \*.jar

> Implemented in Java & corresponding .class file is **sun.misc.Laucher$ExtClassLoader.class**

Note: If in a class filename $ symbol is present i.e. **ExtClassLoader** is an inner class of **Laucher** outer class.

3. **Application class loader/ System class loader**: - It is the child class of Extension class loader. It is responsible to load classes from application classpath (internally uses environment variable classpath).

> Implemented in Java & corresponding .class file is **sun.misc.Laucher$AppClassLoader.class**



**Q. How class Loader works?**

> Class loader follows Delegation Hierarchy principle(algorithm).

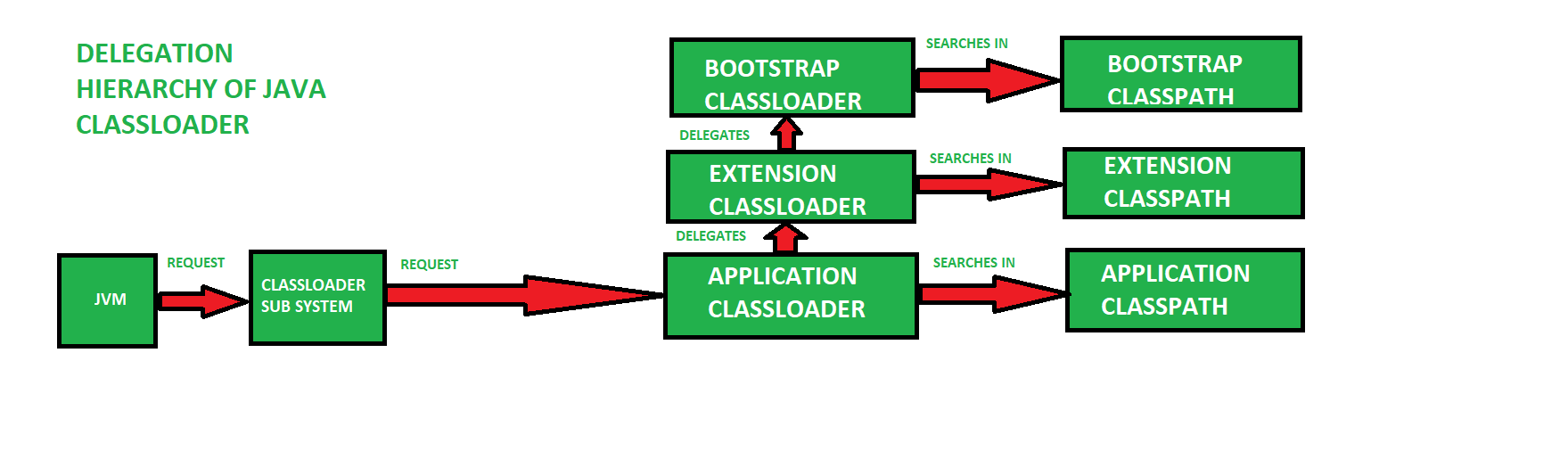
> Whenever JVM come across a particular class, first it will check whether corresponding .class file is already loaded or not.

**IF** (loaded in method Area) -> JVM will consider the loaded class.

**ELSE** JVM requests class Loader subsystem to load that particular class

> Class loader subsystem hand over the request to

**Application class loader** ---- delegates the request to ------ **Extension class loader** ---- delegates the request to --- ---**Bootstrap class loader**.



> The Bootstrap class loader will search in bootstrap classpath. If it is found, then corresponding .class file will be loaded by it else the bootstrap class loader will delegate the request to the Extension class loader & so on delegation goes.

> If the class is not found in Application class loader, we will get runtime exception saying **NoClassdefFoundError** or **ClassNotFoundException**.

**Q. Need for customizer class loader?**

> Default class loader will load .class file only once even though we’re using multiple time that class in our program.

After loading .class file, if it is modified outside then default class loader won’t load updated version of class file as it is already available in method area.

>To resolve this problem, we can define our customizer class loader.

> The main advantage of Customizer class loader is we can control class loading mechanism based on our requirement.

> To define customizer class loader

-> Extend **java.lang.ClassLoader** class (base class for all customizer class loader) & override **loadClass** method.

> While designing/developing web servers & application servers usually, we go for customizer classloader to customize class loading mechanism.

**Various Memory Areas in JVM**

> Whenever JVM loads & runs a Java program, it needs memory to store several things like bytecode, objects, variables etc.

> Total JVM memory is organized into following 5 categories :-

1. Method Area

2. Heap Area

3. Stack Area

4. PC Registers

5. Native Method stacks

**1. Method Area**

> For every JVM, one method area will be available.

> Method area will be created at the time of JVM startup.

> Inside method area, class level binary data including static variables will be stored.

> Constant pools of a class will be stored inside method area.

> Method area can be accessed by multiple threads simultaneously; hence data stored in the method area is not thread safe.

> Method area needs not to be continuous.

**2. Heap Area**

> For every JVM, one heap area is available.

> Heap area will be created at the time of JVM startup.

> Objects & the corresponding instance variables will be stored in the heap area.

> Every array in Java is object only; hence array also will be stored in the heap area.

> Heap area can be accessed by multiple threads simultaneously; hence data stored in the method area is not thread safe.

> Heap area needs not to be continuous.

> Heap memory is finite/fixed memory but based on our requirement we can set maximum & minimum heapsize.

-> We can use the following flags with java command

a) –**Xmx** to set maximum heap size.

E.g. java –Xmx512m className

-> This command will set maximum heapsize as 512 mb

b) –**Xmx** to set minimum heap size

E.g. java –Xms64m className

-> This command will set minimum heapsize (total Memory) as 64 mb

*> Refer program for “displaying heap memory statistics”.*