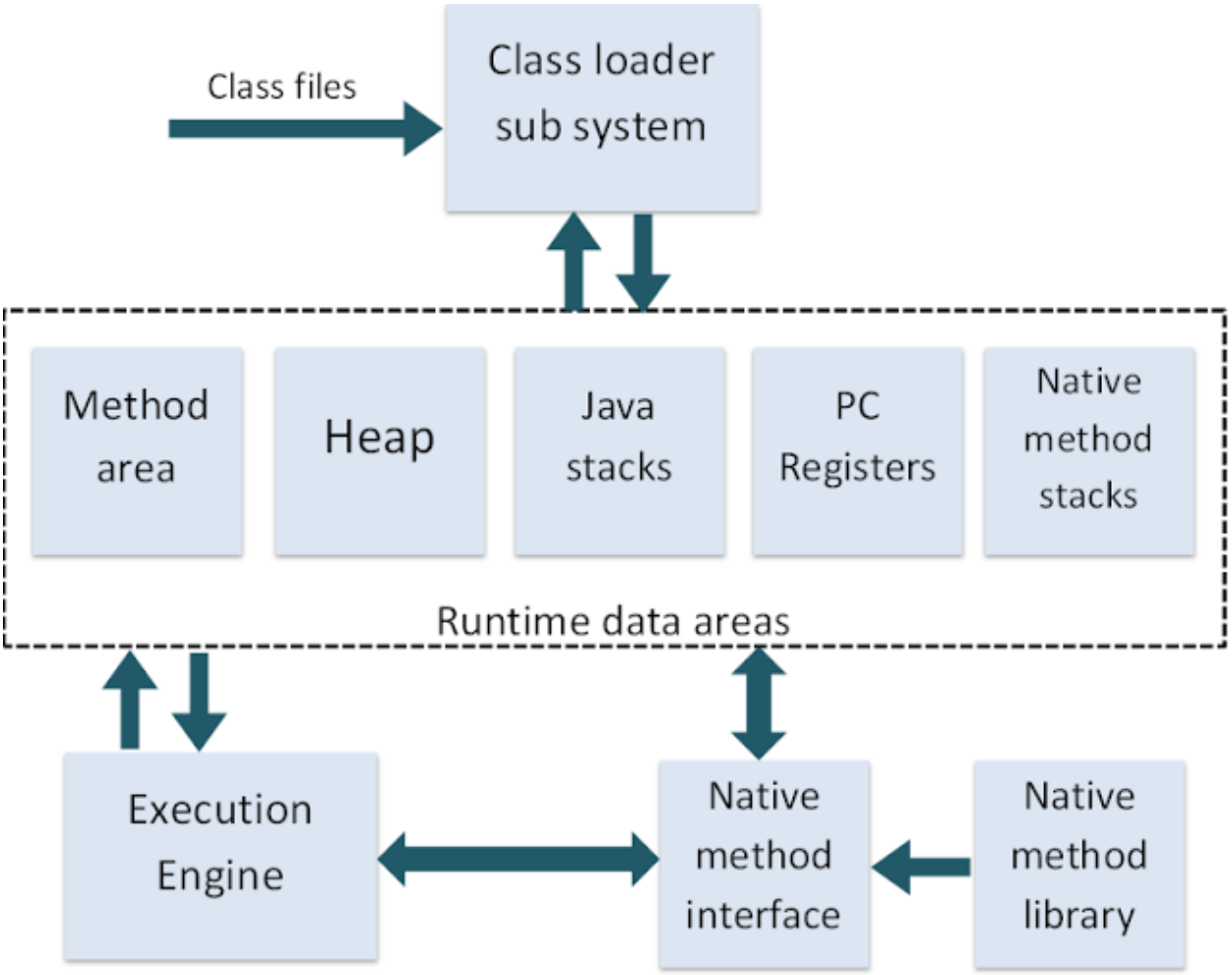
JVM Architecture



Method Area: Method area contains:

1. Type Information
2. Constant Pool
3. Field Information
4. Method Table
5. Method information
6. Class variable

For example for a simple java program:

```
public class HelloWorld {
        private int number;
        private String name;
        public HelloWorld() {
                number = 547;
                name = “suresh”;
        }
        public void printHelloWorld() {
                System.out.println(name + “  Hello World      “  + number);
        }
```

Type information constitutes:

1. Name of the class (HelloWorld)
2. What is this? (java.lang.Object)
3. isClass = true
4. modifier = 4 (since it is class)

Constant Pool constitutes:

Ordered set of constants:

String, int, float and final variables

Symbolic references to:

Types, fields and methods

For the above example constant pool constitutes:

a. number, name, HelloWorld, printHelloWorld.

Field Information constitutes:

1. field name
2. field type
3. field modifier
4. index

for the above example field information constitutes:

Name	Type	Modifier	Index
number	int	5	0
name	string	4	1

Method Info constitutes:

a. Method's name

b. Method's return type

c. Number and type of parameters

d. modifiers

for the above example method info constitutes:

Name	Return type	Number of parameters	modifiers	Parameter List
HelloWorld	void	0	5	0
printHelloWorld	Void	0	5	0

Method Table: holds the reference of **Method Information**

for the above example method table constitutes:

Name	index
Helloworld	2
printHelloWorld	3

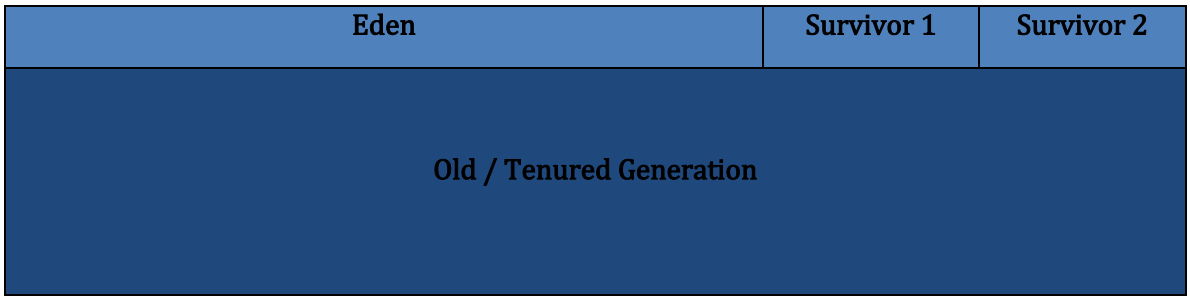
Class Variable: constitutes the static variables and class variable information

For the above example class variable information has null value

Heap: this is the area used by the JVM, specifically, HotSpot, for dynamic memory allocation.

Heap is divided into two generations namely:

1. Young generation
2. Old or Tenure generation



Permanent Generation: ***for VM and class meta data***

Newly created variables or methods will be allocated in **Eden generation** area. Upon minor garbage collections objects are moved to **Survivor 1.**

JVM uses certain algorithm to move the values from Eden to Survivor area. All the threads are paused during the minor garbage collection process.

Frequency of minor garbage collection is dictated by:

i. Rate of allocation of objects

ii. Size of Eden space

Frequency of object promotion from **Young generation** to **Tenure generation** is dictated by:

i. Frequency of minor garbage collection

ii. Size of **Survivor** space

iii. Tenuring Threshold (7 times between survivor 1 and 2)

Algorithms used by JVM for garbage collection are:

a. Serial Collector

b. Parallel Collector

c. Concurrent Mark-Sweep Collector

d. GC1

Collectors operated on **Young Generation** are:

Serial Collector	-XX:+UseSerialGC
Parallel Collector	-XX:+UseParallelGC
ParNew Collector	-XX:+UseParNewGC

Collectors operated on **Old/Tenure Generation** are:

ParallelOld Collector -XX:+UseParallelOldGC

Concurrent Mark Sweep collector -XX:+UseConcMarkSweepGC

Serial Collector: is a default copying collector which uses only one GC thread for the GC operation

Parallel Collector: collector uses multiple GC threads for the GC operation.

CMS Collector: collector performs the following steps (all made by only one GC thread): -

- initial mark
- concurrent marking
- remark
- concurrent sweeping

Initial Mark - Pauses all application threads and marks all objects directly reachable from root objects as live. This phase stops the world.

Concurrent Mark - Application threads are restarted. All live objects are transitively marked as reachable by following references from the objects marked in the initial mark.

Concurrent Preclean - This phase looks at objects which have been updated or promoted during the concurrent mark or new objects that have been allocated during the concurrent mark. It updates the mark bit to denote whether these objects are live or dead. This phase may be run repeatedly until there is a specified occupancy ratio in Eden.

Remark - Since some objects may have been updated during the preclean phase its still necessary to do stop the world in order to process the residual objects. This phase does a retrace from the roots. It also processes reference objects, such as soft and weak references. This phase stops the world.

Concurrent Sweep - This looks through the Ordinary Object Pointer (OOP) Table, which references all objects in the heap, and finds the dead objects. It then re-adds the memory allocated to those objects to its freelist. This is the list of spaces from which an object can be allocated.

Concurrent Reset - Reset all internal data structures in order to be able to run CMS again in future.

Java Stacks: this is the place where java methods are executed.

Program Counter: stores the address of each instruction.

Native Method Stacks: place where native methods are executed.

Execution Engine: Execution engine contains interpreter and JIT compiler, which covert byte code into machine code.

Class Loader

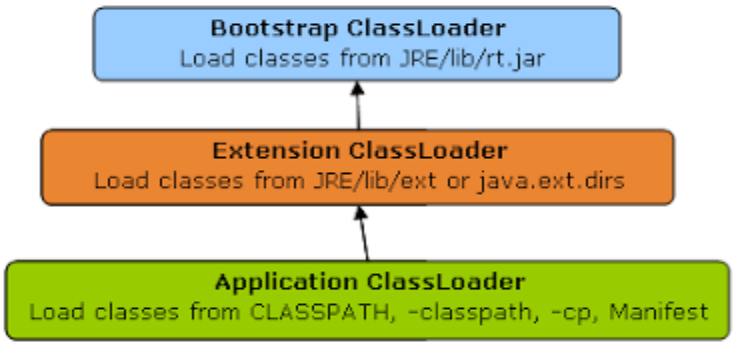
ClassLoader in Java is a class which is used to load class files in Java. Java code is compiled into class file by **javac** compiler and JVM executes Java program, by executing byte codes written in class file. ClassLoader is responsible for loading class files from file system, network or any other source. 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** don't have any parents, if you call String.class.getClassLoader() it will return null and any code based on that may throw NullPointerException 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. Extension ClassLoader in JVM is implemented by **sun.misc.Launcher$ExtClassLoader**.

System or Application ClassLoader 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. Also, except Bootstrap class loader, which is implemented in native language mostly in C, all Java class loaders are implemented usingjava.lang.ClassLoader.

Java class loaders are used to load classes at runtime. ClassLoader in Java works on three principle: delegation, visibility and uniqueness. Delegation principle forward request of class loading to parent class loader and only loads the class, if parent is not able to find or load class. Visibility principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader cannot see classes loaded by child. Uniqueness principle allows to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent.



In short here is the location from which Bootstrap, Extension and Application ClassLoader load Class files.

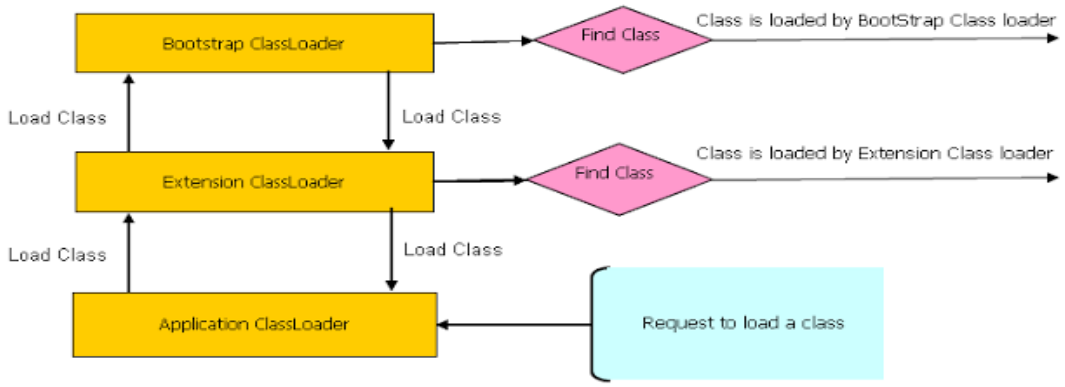
1) **Bootstrap** ClassLoader - JRE/lib/rt.jar

2) **Extension** ClassLoader - JRE/lib/ext or any directory denoted by java.ext.dirs

3) **Application** ClassLoader - CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.

How ClassLoader works in Java

Java ClassLoader works in three principles: delegation, visibility and uniqueness.



Delegation principles

When a class is loaded and initialized in Java, a class is loaded in Java, when its needed. 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. Remember Classpath is used to load class files while PATH is used to locate executable like javac or java command.

Visibility Principle

According to visibility principle, Child ClassLoader can see class loaded by Parent ClassLoader but vice-versa is not true. Which mean if class Abc is loaded by Application class loader than trying to load class ABC explicitly using extension ClassLoader will throw either java.lang.ClassNotFoundException.

Uniqueness Principle

According to this principle a class loaded by Parent should not be loaded by Child ClassLoader again. Though its completely possible to write class loader which violates Delegation and Uniqueness principles and loads class by itself, its not something which is beneficial. You should follow all class loader principle while writing your own ClassLoader.

How to load class explicitly in Java

Java provides API to explicitly load a class by **Class.forName(classname) and Class.forName(classname, initialized, classloader).** Class is loaded by **callingloadClass**() method of **java.lang.ClassLoader** class which calls **findClass**() method to locate bytecodes for corresponding class. If **findClass**() does not found the class than it **throws java.lang.ClassNotFoundException** and if it finds it calls **defineClass**() to convert bytecodes into a .class instance which is returned to the caller.

Where to use ClassLoader in Java

ClassLoader in Java is a powerful concept and used at many places. One of the popular example of ClassLoader is AppletClassLoader which is used to load class by Applet, since Applets are mostly loaded from internet rather than local file system, By using separate ClassLoader you can also loads same class from multiple sources and they will be treated as different class in JVM. J2EE uses multiple class loaders to load class from different location like classes from WAR file will be loaded by Web-app ClassLoader while classes bundled in EJB-JAR is loaded by another class loader. Some web server also supports hot deploy functionality which is implemented using ClassLoader.