**JVM Architecture:**

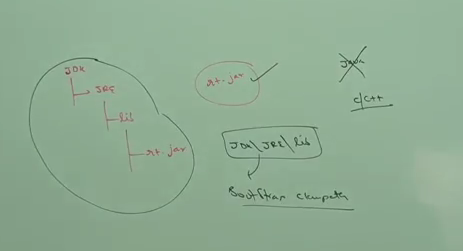
**Types of Class Loaders:**

The Class loader subsystem have three types of class loader:

1. Bootstrap class loaders or Primordial class loader (native class loader)
2. Extension class loader
3. Application class loader or System class loader

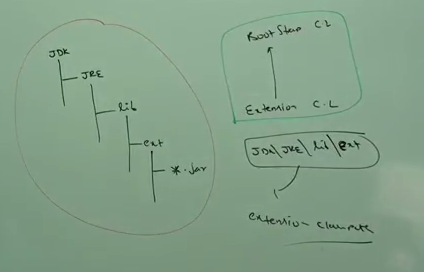
**1. Bootstrap class loader:**

1. Bootstrap class loader is responsible to load core java api classes from bootstrap classpath. That is the classes present in rt.jar
2. Rt.jar under jdk/jre/lib/rt.jar -> contains all core java api classes like String, StringBuffer, etc.,
3. Jdk/jre/lib – this location is called bootstrap class path. That is bootstrap class loader is responsible to load the core java api classes from bootstrap class path.
4. Bootstrap class loader is by default available with every JVM.
5. It is implemented in native languages like C or C++ and not implemented in JAVA.



**2. Extension class loader:**

1. Extension class loader is the child class of bootstrap class loader.
2. Extension class loader is responsible to load the classes from extension classpath.(jdk.jre/lib/ext/\*.jar)
3. jdk.jre/lib/ext is called extension classpath
4. It is implemented in java. The corresponding .class file is **sun.misc.Launcher$ExtClassLoader.class**

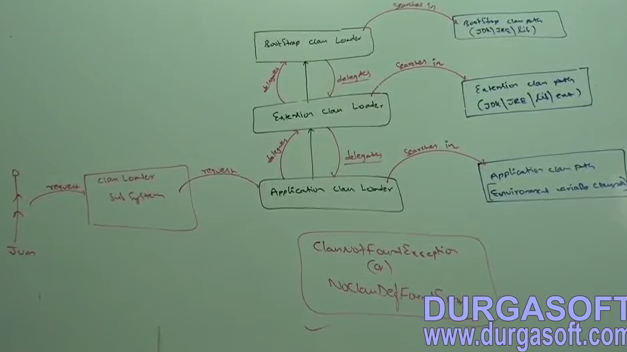


**3. Application or System class loader:**

1. It is the child class of Extension class loader
2. It is responsible to load the classes from application class path.
3. It internally uses environment variable classpath. That is the classpath that we set in environment variables in the application.
4. It is implemented in java and the corresponding .class file is **sun.misc.Launcher$AppClassLoader.class**

**How class loader works:**

1. **Class loader follows delegation hierarchy principle or algorithm**
2. Whenever JVM comes across a particular class, first it will check whether the corresponding .class file is already or not. If it is already loaded in method area, then JVM will consider that loaded classes. If it is not loaded, then the jvm request class loader sub system to load that particular class.
3. Then class loader subsystem handover the request to Application class loader. Application class loader delegates the request to Extension class loader which in turn delegates the request to Bootstrap class loader.
4. Bootstrap class loader will search in bootstrap classpath. If it is available, then it will be loaded by bootstrap class loader. If it is not available, then it delegates the request to extension class loader.
5. Extension class loader will search in extension class path. If it is available, then it will be loaded by extension class loader. If it is not available, then again it delegates the request to application class loader.
6. Applicaion class loader will search in application class path that is environment variables that we set in our project workstapce. If it is available, then it will be loaded, otherwise we will get runtime exception saying **ClassNotFoundException or NoClassDefFoundError**.



**Example: To find class loaders**

Class Test{

Public static void main(String a[]){

Sop(**String**.class.getClassLoader()); -> we get **null. String class present in bootstrap class path and bootstrap class loader loads it. But we get null because bootstrap class loader is not a java object and it is not implemented in java. So we get null.**

Sop(**Test**.class.getClassLoader());

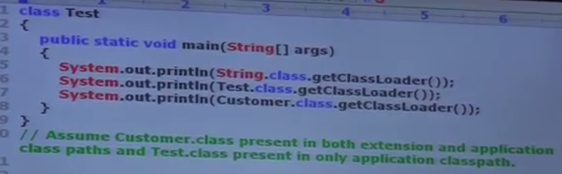
//Create Customer class and make a jar and put into extension class path (jdk/jre/lib/ext)

//To createjar:  **jar –cvf cust.jar Customer.class**

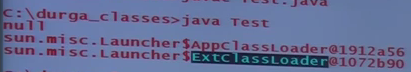
Sop(**Customer**.class.getClassLoader());

}

}



**Output:**

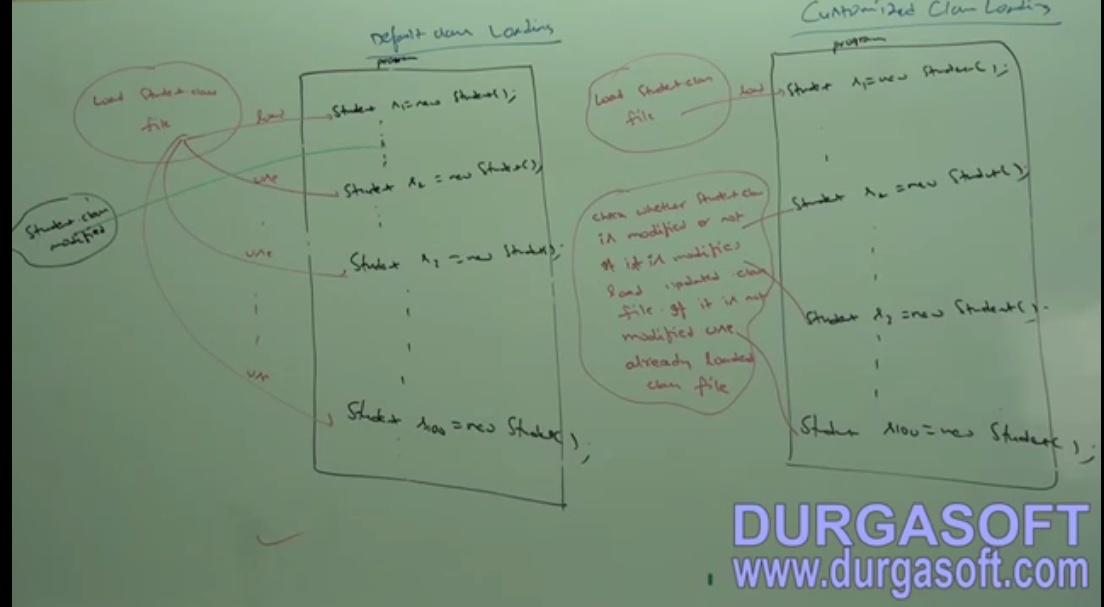


The highest priority for class loader is bootstrap class loader and then Extension class loader and finally Application class loader.

**Customized Class Loader:**

**Need:**

1. **Default Class loading mechanism: ->** Default class loader will load .class file only once eventhough we are using multiple times that class in our program. After loading .class file, if it is modified outside, then default class loader wont load updated version of .class file (because .class file is already available in method area or jvm memory).
2. We can resolve these problem, by defining our own customized class loader
3. The main advantage of customized class loader is we can control class loading mechanism based on lur requirement.
4. For Example, we load .class file separately every time, so that updated version available to our program.

****

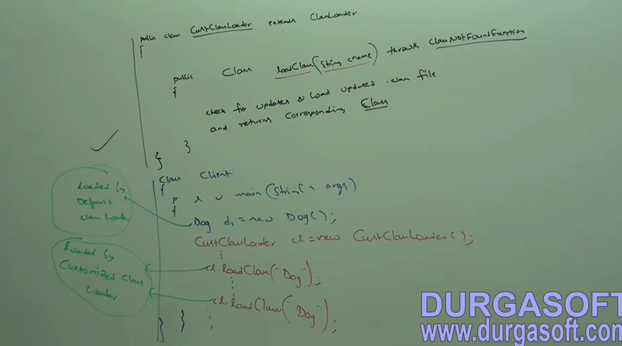
In Customized class loading -> First load the Student class if it is not loaded. If it is already loaded, check student class is modified outside. If it is modified use updated .class file else if it is not modified, use the already loaded class file.

Mostly customized class loading is useful in developing JVM, webserver, application server etc.,

**Weblogic 7 uses default class loading mechanism and Weblogic 10 uses customized class loading given above.**

**How to define customized class loader:**

1. Our class loader should extend **java.lang.ClassLoader**
2. Override **Class loadClass(Sring name) throws ClassNotFoundException (if .class file is not present in system, it throws this exception)**



**Various memory Area:**

Whenever jvm load s and runs a java program, it needs memory to store several things like byte code, objects, variables, etc.

Total jvm memory organized into following 5 categories.

1. Method area
2. Heap area
3. Stack memory
4. Pc registers
5. Native method stacks

Method area:

1. For every jvm, one method area is available which will be created at jvm startup.
2. Total .class will be dumbed into method area (class level binary information).
3. At the time of static variables will be created and it is stored in method area.
4. Constant pools of a class will be stored here.
5. The data stored in method area is not threadsafe.
6. It need not be continuous memory.