**What is Singleton**

“A class which can have only one object within the scope of **class loader** is called singleton class.”

Now if we observe the definition so first question arise in our mind is what scope of class loader is, second question is how is it possible to create only one object.

So, first we will understand what class loader is. How it works.

**ClassLoader**

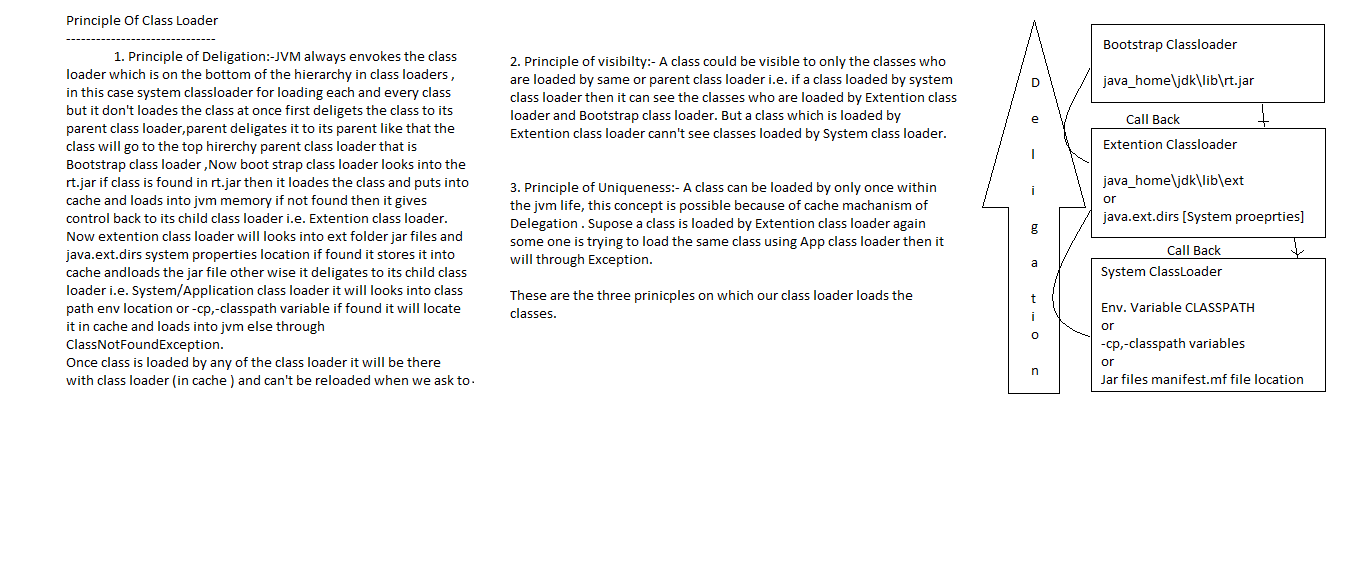
Whenever we are trying to execute any java class, first that class will be loaded into the jvm. So to load the classes into the jvm memory there is a module in jvm which is called **Class Loader**. That means Class Loader’s are again some java classes which will load the byte code in jvm memory. JVM can execute only those classes which are loaded in jvm memory. Actually JVM is divided into multiple parts based on the functionality like for memory dislocation garbage collector is a separate module for memory allocation related issues Memory management system. Like that JVM have provided a module called ClassLoader which is responsible for loading the classes into JVM memory.

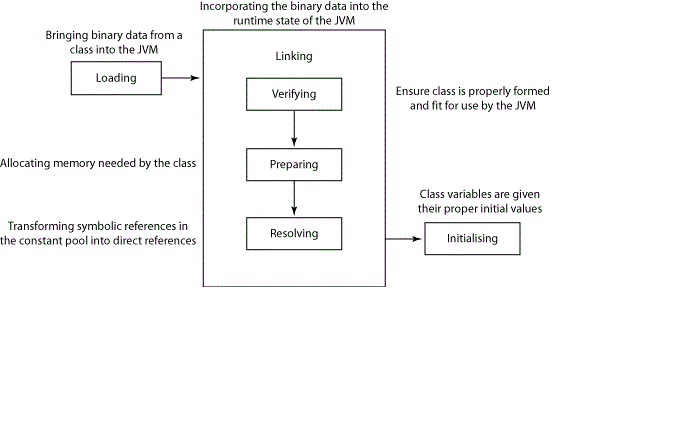
So as part of core jdk there are totally three Class Loader’s all these three class loader will load the classes from their specific locations. So lets have a look on these three.

**Bootstrap class loader**:-This is a native class loader which is starter of a jvm it will load all the classes from **rt.jar** on demand whenever we are trying to execute any class.

**Extension Class Loader**:- This class loader is responsible for loading the classes from jdk\lib\ext directory jars and java.ext.dirs system properties location.

**System/Application class loader**:- This class loader will load the classes from class path location , or classpath variable location or from the jar file related manifest file provided classes.





**Class Loading Phases**

There are totally three phases in loading of a class.

1. Loading

In this phase class loader will read physical byte code from the location.

1. Linking

In linking phase there are three steps

1. Verifying:-In this phase loaded byte code will be verified based on various aspects like It will checks whether the loaded byte code is matching based on checksum number or not.

In case of Jar file it matches whether signature of Jar is matching or not If all these are done then it will be ready for defining class definition for that.

1. Preparing :- In this phase class definition for the loaded class will be prepared .
2. Initializing

In this phase all the class level variables and constants will be initialized, static block will be called. So if you don’t want to initialize these class level properties at loading time then you can call Class.forName(“ClassName”,false,ClassLoader) the second parameter in this method tells whether initialization should be done or not.

**Custom Class Loader’s**

So there are many cases when we need to load the class using our class loader for example if our classes are at remote location and we want to use those classes into our application or so these traditional class loaders will not be able to load the classes from that location. So we can write our custom class loader to load the classes from that location. Or if our application is web based and WarClassLoader is not able to load the classes then again we can go for custom class loader.

So for making custom loader we have to follow all the principles and all the phases provided by JAVA so that jvm can recognize our class loader.

Now lets know what happens when we are trying to execute any class :-

Whenever we are executing any class first that class should be loaded into jvm memory , and loading of a class will be done by any of the class loader , So internally jvm holds the refrence of ClassLoader type as every ClassLoader is sub class of java.lang.ClassLoader so jvm holds refrence of ClassLoader and upon this refrence jvm will always calls loadClass(String className) method which is origanlly present in ClassLoader class and further inherited in all the other class loaders. So if we want to create our own class loader then we have to extend from ClassLoader class and override loadClass() method and give the refrence of our class loader to jvm. So that JVM will call that method on our refrence and our class load method will be called. So programmatically -

Class MyClassLoader extends ClassLoader{

Public Class<?> loadClass(String className){

//now writes the logic for loading the class here

}

Now it’s clear that how we can make our class loader recognized by jvm.

So to load the classes from our class loader we have to follow the principles and phases of class loaders i.e. Delegation , Visibility and uniqueness so to maintain these all we need to create one parent for our class loader, and delegate the classes to the parent if parent is not able to load the class then only we need to load , after loading the class we should hold the reference of that class so that uniqueness could be maintained and as we are following delegation so it will be visible to the child as well. Now lets do it programaticaly.

package com.nad.class\_loader;

import java.io.ByteArrayOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

import java.util.HashMap;

import java.util.Map;

public class CustomClassLoader extends ClassLoader {

private Class<?> clazz = null;

Map<String, Class<?>> classes = null;

byte[] byteCode = {};

public CustomClassLoader(ClassLoader parent) {

// Setting the parent class loader

super(parent);//Principle of visibility

//Principle of uniqueness

classes = new HashMap<String, Class<?>>(); // Initializing cache memory

}

// Called by JVM when ever we are trying to call Class

// clazz=Class.forName("org.nad.beans.Demo", true, customClassLoader)

public Class<?> loadClass(String name) throws ClassNotFoundException {

// Check the class in cache

if (classes.containsKey(name))

return classes.get(name);

try {

// Delegate to Parent(Principle of Delegation)

return findSystemClass(name);

} catch (ClassNotFoundException e) {

// IF control comes to catch that means no parent find class at

// there location

}

return getClass(name);

}

// Returns byte code in form of byte array from file system(Loading Phase)

private byte[] loadClassFile(String name) throws IOException {

byte[] byteArray = {};

int b = -1;

// Input stream to read bytes from class file

FileInputStream fis = new FileInputStream(new File(getClassName(name)));

ByteArrayOutputStream bos = new ByteArrayOutputStream();

while ((b = fis.read()) != -1) {

// Writing the bytes into OutputStream

bos.write(b);

}

fis.close();

byteArray = bos.toByteArray();

bos.close();

return byteArray;

}

public String getClassName(String name) {

// Form the class directory

// com.nad.Demo

// com\\nad\\Demo.class

// Finally Root\_Dir+Class\_Name=F:\\class\_loder\\com\\nad\\Demo.class

String Class\_Name = name.replace(".", File.separator) + ".class";

String Root\_Dir = "F:\\class\_loder\\";

return Root\_Dir + Class\_Name;

}

// Takes the byte array and returns Class object stores class in Cache

public Class<?> getClass(String name) throws ClassNotFoundException {

try {

byteCode = loadClassFile(name);

} catch (IOException e) {

throw new ClassNotFoundException(name);

}

//It will verify the class and define the class definition for the loaded byte code

clazz = defineClass(name, byteCode, 0, byteCode.length);

//THIS CODE WILL RESOLVE THE DEFINDE CLASS

resolveClass(clazz);

classes.put(name, clazz);//Putting the loaded class in cache for future reference (Principle of uniqueness)

return clazz;

}

}

**When To go for Singleton Class**

1. When the class is stateless eg: Validator classes instead of using static methods for validation because we cant mock static methods so testing becomes difficult. instead if we have singleton we can mock the instance itself.  
   2. When  we need only one state of class at any given point of time. In this case we have to take care of synchronization.
2. a static class cannot implement an interface. When a single instance class needs to implement an interface for a business contracts or IoC purposes, this is where I use the Singleton pattern without a static class.
3. The singleton pattern can be used for anything that you don't want to repeat. If the object in question is not expected to change, it is a good candidate for the singleton pattern. The singleton pattern is preferred over a static implementation in most cases. Static implementations can cause dependency headaches. Can you provide more specifics as to what exactly you're looking to do?
4. We used the singleton pattern in our utility layers which consist of Logging, Caching, Service host repositories, Load Balancer... If the question is on how we arrived to the design. There was a performance lag on the utility layer eg, Logging, on diagnosing we observed that there are several instance getting created which are not required in my case. So we adopted Singleton pattern.