Singlton with ClassLoaders

ClassLoaders:-

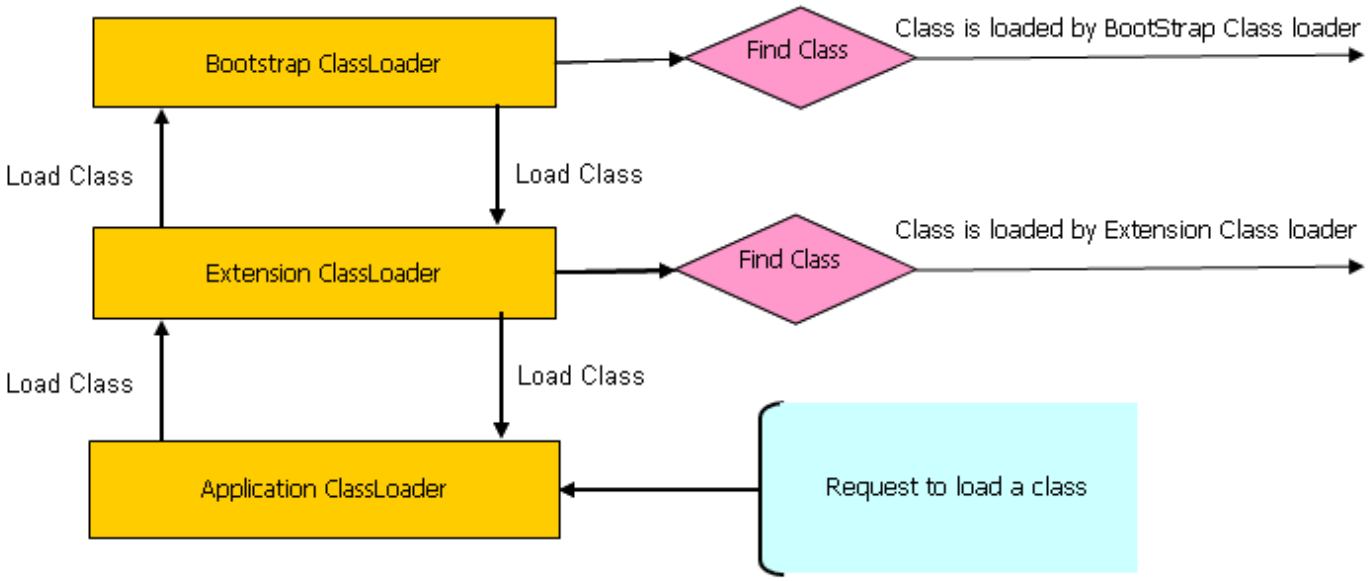
Jvm internally uses classLoaders sub system to load the java class…

Upto java 8 we hdave 3 classLoaders and from java 9 we have 2 class Loaders

1. *Bootstrap ClassLoader (can load the classes from rt.jar of <java_home>\jre\lib folder)*
2. *Extension class loader (can oad the classes frim jar file added to <java_home>\jre\lib\ext folder)*
3. *System/Application/classpath Classloader (can load the classes from the directories and jar file add to classpath env…variable)*

Note:- "from java 9 onwords Extension classLoaders is removec from java..so we do not "ext" folder in jdk's installation."

⇨ *These classLoaders are hierarchal…*



Flow of class loader:-

Jvm get the request to load the class it will handover to the class loaders subsystem

- *First it will give the class to Application Class loader-> application class loader pass to extention class loader that will pass to bootstrap class loader because class loader are hierarchal.*
- *This boot strap class loader first search given class is there in rt.jar or not yes there loads it if not there*

- *It will pass to extention class loader.. Extention class loader will check weather given class is there jar added ext folder or not there then loads it ..if not there*
- *Passes to Appplication Class loader nothing but classpath class loader this class loader will search weather given class in there in directories and jars that are added to classpath or not yes there then loads it if not there…if it is a direct class then throw classnotFoundException if indirect class then it will throw no class definition found exception.*

There are 3 principals of classloaders sub system:-

1. *Delegation principal ::*
   since classLoaders are hierarchal, so if class not loaded by one classLoaders then it deleges another ClassLoader in the hierarchy.
2. *Visible principal ::*
   The classes loaded by parent class loader are visible to child class loaders but reverse is not true.
3. *Uniqueness principal ::*
   Only ClassLoader will load the given class..i.e same class will not be loaded by multiple Class Loaders.

- ⇨ *Every ClassLoader is class that extends from java.lang.ClassLoader class..*
- ⇨ *To develop CustomClassLoader we need to take java class that extends from java.lang/ClassLoader class..*
- ⇨ *We are some readymade custom ClassLoaders*
  - *Eg: java.net.URLClassLoader…(useful to load classes from jars/directories that are not added to classPath)*

<u>Q:- When we go for custom classLoader:-</u>

Ans:- We don't want to add jars to ext folder I don't want to add jars to classpath because if I add they are given class to other jar but still I want to use those classes I want to use my application so that time I want to go for custom class loader concept.

"by default custom class loader is not in the hierarchy os standard ClassLoaders"

"within a jvm we can break singleton java class nehaviour by taking the support of customClassLoader…i.e. we can create second object in the same jvm for singleton class with support of Customer ClassLoader."

Singlton with ClassLoaders

For classloaders we have to prepared jar file of project by using command class:-

Pathof project to bin directory:- *jar -cf jarname.jar .*

After that we have to copy created jar file in eclips ide project structure

We can develop singleton class with Class Loaders below 3 approaches:-

1. ***Using thraditional approach***
2. ***Using innerclass approach***
3. ***Using enum based approach***

" place singleton classes in jar file like sdp.jar and gives its location to custom class loader"

Here we will use Redimade custom class loader which is URLClassLoader.

1. Using Thraditional approach:-

```
1. package com.nit.dp;
2.
3. public class Printer extends CloneTest {
4.
5.    private static final long serialVersionUID = 8262877753773712150L;
6.    public static Printer instance;
7.    public boolean condition=false;
8.
9.     //constructor
10.        private Printer() {
11.             if(condition==true) {
12.             throw new IllegalArgumentException("object alredy created");
13.             }
14.             condition=true;
15.             System.out.println("0 param constructor");
16.        }
17.
18.        //static factory method
19.        public static Printer getInsance() {
20.             if(instance==null) {
21.                  synchronized (Printer.class) {
22.                       if(instance==null) {
23.                            instance=new Printer();
24.                       }//if
25.                  }//synchronized
26.             }//if
27.             return instance;
28.        }
29.
30.        //clone method
31.        @Override
32.        public Object clone() throws CloneNotSupportedException {
33.             System.out.println("Printer.clone()");
34.             return instance;
35.        }
36.
37.        //calling readResolve method
38.        public Object readResolve() {
39.             System.out.println("Printer.readResolve()");
40.             return instance;
41.        }
42.
43.        //creating business method
44.        public static String printData(String data) {
45.             System.out.println("Entered data is="+data);
46.             return data;
47.        }
48.   }
```

ClassLoader Test:-

```
1. package com.nit.test;
2. import java.lang.reflect.Method;
3. import java.net.URL;
4. import java.net.URLClassLoader;
5. import com.nit.dp1.Printer;
6. public class Test {
7.
8.    public static void main(String[] args) throws Exception {
9.          com.nit.dp1.Printer p1=null;
10.               Object p2=null;
11.               URLClassLoader loader1=null;
12.               //creating object
13.               p1=Printer.getInstance();
14.
15.               System.out.println("hashcode="+p1.hashCode());
16.          loader1=newURLClassLoader(newURL[] {new URL
   ("file:/D:\\design pattern\\sigltonClas with ClassLoder\\bin\\sdb.jar")},null);
      //null says this is not any parent class and this a independent class loader
17.          Class<?> class1=loader1.loadClass("com.nit.dp1.Printer");
18.          Method method=class1.getDeclaredMethod("getInstance", new Class[] {});
19.               p2= method.invoke(null);
20.               System.out.println(p2.hashCode());
21.               System.out.println(p1.hashCode()==p2.hashCode());
22.          }
23.   }
```

Output :-

```
Printer.Printer()
hashcode=366712642
Printer.Printer()
2101973421
false
```

“by using Class Loader it break singleton behavior also because we are using two classes in same jvm first class loaded by application Class loader as normal and the other class is loaded but costume class loader so at time single jvm two object got created and constructor also calling two times….the reason behind is custom class loader in not in the hierarchy of Class Loader there are no parent class for custom class loader.”

Solution :-

Solution of this custom class loader is we have to define custom class loader also in the hierarchy of Class loader so that we can control of creating second object in same jvm indirect way we have to create parent class loader for custom class loader in traditional approach.

Singlton with ClassLoaders

```
1. package com.nit.test;
2.
3. import java.lang.reflect.Method;
4. import java.net.URL;
5. import java.net.URLClassLoader;
6. import com.nit.dp1.Printer;
7. public class Test {
8.
9.    public static void main(String[] args) throws Exception {
10.             com.nit.dp1.Printer p1=null;
11.             Object p2=null;
12.             ClassLoader loader=null;
13.             URLClassLoader loader1=null;
14.             //creating object
15.             p1=Printer.getInstance();
16.
17.             System.out.println("hashcode="+p1.hashCode());
18.         //    loader=p1.getClass().getClassLoader();
19.             //System.out.println(loader);
20.             loader1=new URLClassLoader(new URL[] {new
   URL("file:/D:\\design pattern\\sigltonClas with ClassLoder\\bin\\sdb.jar")},nul
   l);  //null says this is not any parent class and this a independent class
   loader
21.             //loader1=new URLClassLoader(new URL[] {new
   URL("file:/D:\\design_pattern\\sigltonClas_with_ClassLoder\\bin\\sdb.jar")},loa
   der);
22.             Class<?> class1=loader1.loadClass("com.nit.dp1.Printer");
23.             Method method=class1.getDeclaredMethod("getInstance", new Class[]
   {});
24.             p2= method.invoke(null);
25.             System.out.println(p2.hashCode());
26.
27.             System.out.println(p1.hashCode()==p2.hashCode());
28.         }
29.   }
```

Output:-

```
Printer.Printer()
hashcode=366712642
sun.misc.Launcher$AppClassLoader@73d16e93
366712642
true
```

“in the above ClassLoader Test we have solved issue of creating second object by custom class loader … we have called Classloader as the ApplicationClass Loader and given as parent class of custom class loader so that creating second object issue gone solved.”

Note:- “by using inner class and enum also we can not control to create second object by using class loader we have to define parent class loader of the custom class loader in all 3 approach so that we can control of creating second object.”

For inner class:-

Ex:-

```
1. package com.nit.dp;
2. public class Printer extends CloneTest {
3. private static boolean flag=false;
4.
5.    private Printer(){
6.          if(flag==true) {
7.                throw new IllegalArgumentException("object already created");
8.          }
9.          flag=true;
10.               System.out.println("0 param constructor");
11.         }
12.         public static Printer getInstance() {
13.               return printerHolder.instance;
14.         }
15.         //nested/static inner class
16.         private static class printerHolder{
17.               //egar instantiation
18.               private static Printer instance=new Printer();
19.               public printerHolder() {
20.                     System.out.println("Printer.printerHolder.printerHolder()");
21.               }
22.         }
23.         @Override
24.         public Object clone() throws CloneNotSupportedException {
25.               throw new CloneNotSupportedException("can not clonable");
26.               //return this;
27.         }
28.               public Object readResolve() {
29.               System.out.println("Printer.readResolve()");
30.         throw new IllegalArgumentException("printer do not want to support deserialization");
31.         }
32.   }
```

Singlton with ClassLoaders

For Enum:-

```
1. package com.nit.dp2;
2. public enum Printer {
3.       instance;
4.
5.       public static  Printer getInstance() {
6.            return instance;
7.       }
8.       //business method
9.       public String printData(String data) {
10.                System.out.println("given data is="+data);
11.                return data;
12.           }
13.      }
```

Note:- “for all 3 approaches classLoader Test is acting same it creating second object for all tree approaches”

Q:- can we develop 100% Perfect singleton java class?
Ans:-

- ⇨ Using traditional, inner class approach we can develop for 98%
- ⇨ Using enum we can develop for 99%

“Reflection api solution is also not a perfect solution”