ClassLoader in Java

Java ClassLoader

Java ClassLoader is an abstract class. It belongs to a **java.lang** package. It loads classes from different resources. Java ClassLoader is used to load the classes at run time. In other words, JVM performs the linking process at runtime. Classes are loaded into the JVM according to need. If a loaded class depends on another class, that class is loaded as well. When we request to load a class, it delegates the class to its parent. In this way, uniqueness is maintained in the runtime environment. It is essential to execute a Java program.

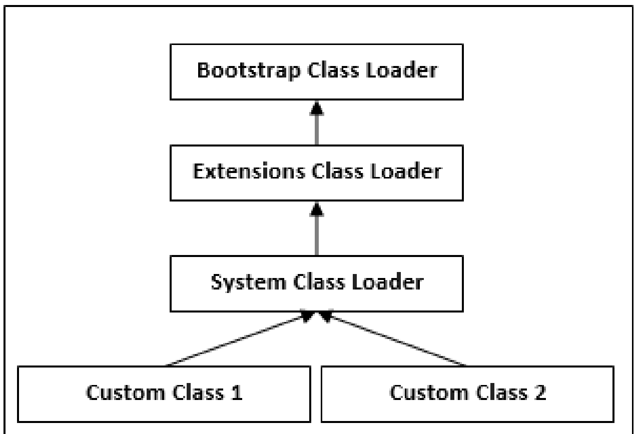


Figure: Hierarchy of Class Loader

Java ClassLoader is based on three principles: **Delegation**, **Visibility**, and **Uniqueness**.

- **Delegation principle:** It forwards the request for class loading to parent class loader. It only loads the class if the parent does not find or load the class.
- **Visibility principle:** It allows child class loader to see all the classes loaded by parent ClassLoader. But the parent class loader cannot see classes loaded by the child class loader.
- **Uniqueness principle:** It allows to load a class once. It is achieved by delegation principle. It ensures that child ClassLoader doesn't reload the class, which is already loaded by the parent.

Types of ClassLoader

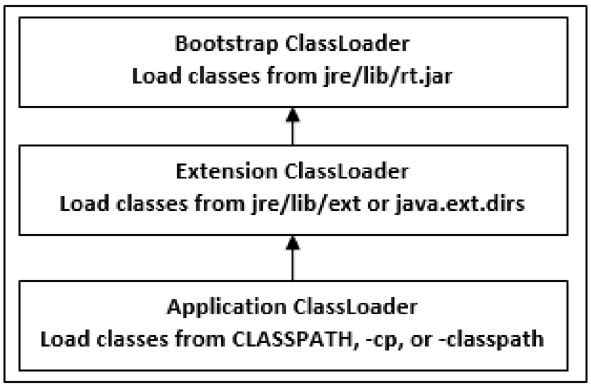
In Java, every ClassLoader has a predefined location from where they load class files. There are following types of ClassLoader in Java:

Bootstrap Class Loader: It loads standard JDK class files from rt.jar and other core classes. It is a parent of all class loaders. It doesn't have any parent. When we call String.class.getClassLoader() it returns null, and any code based on it throws NullPointerException. It is also called Primordial

⇧ SCROLL TO TOP :lass files from jre/lib/rt.jar. For example, java.lang package class.

Extensions Class Loader: It delegates class loading request to its parent. If the loading of a class is unsuccessful, it loads classes from jre/lib/ext directory or any other directory as java.ext.dirs. It is implemented by sun.misc.Launcher$ExtClassLoader in JVM.

System Class Loader: It loads application specific classes from the CLASSPATH environment variable. It can be set while invoking program using -cp or classpath command line options. It is a child of Extension ClassLoader. It is implemented by sun.misc.Launcher$AppClassLoader class. All Java ClassLoader implements java.lang.ClassLoader.


Bootstrap ClassLoader
Load classes from jre/lib/rt.jar
Extension ClassLoader
Load classes from jre/lib/ext or java.ext.dirs
Application ClassLoader
Load classes from CLASSPATH, -cp, or -classpath

How ClassLoader works in Java

When JVM request for a class, it invokes a loadClass() method of the java.lang.ClassLoader class by passing the fully classified name of the class. The loadClass() method calls for findLoadedClass() method to check that the class has been already loaded or not. It is required to avoid loading the class multiple times.

If the class is already loaded, it delegates the request to parent ClassLoader to load the class. If the ClassLoader is not finding the class, it invokes the findClass() method to look for the classes in the file system. The following diagram shows how ClassLoader loads class in Java using delegation.

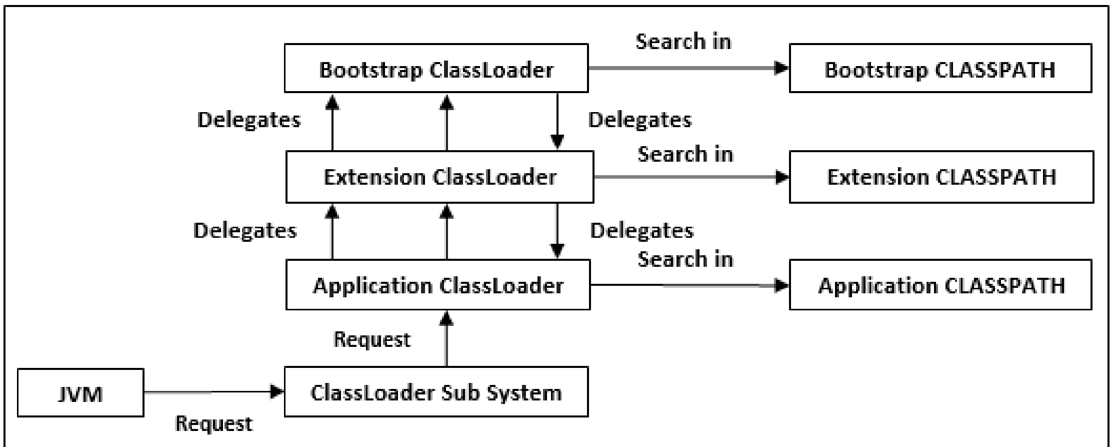


Figure: Working of ClassLoader

⇧ SCROLL TO TOP

Suppose that we have an application specific class Demo.class. The request for loading of this class files transfers to Application ClassLoader. It delegates to its parent Extension ClassLoader. Further, it delegates to Bootstrap ClassLoader. Bootstrap search that class in rt.jar and since that class is not there. Now request transfer to Extension ClassLoader which searches for the directory jre/lib/ext and tries to locate this class there. If the class is found there, Extension ClassLoader loads that class. Application ClassLoader never loads that class. When the extension ClassLoader does not load it, then Application ClaasLoader loads it from CLASSPATH in Java.

Visibility principle states that child ClassLoader can see the class loaded by the parent ClassLoader, but vice versa is not true. It means if Application ClassLoader loads Demo.class, in such case, trying to load Demo.class explicitly using Extension ClassLoader throws java.lang.ClassNotFoundException.

According to the uniqueness principle, a class loaded by the parent should not be loaded by Child ClassLoader again. So, it is possible to write class loader which violates delegation and uniqueness principles and loads class by itself.

In short, class loader follows the following rule:

- It checks if the class is already loaded.
- If the class is not loaded, ask parent class loader to load the class.
- If parent class loader cannot load class, attempt to load it in this class loader.

Consider the following Example:

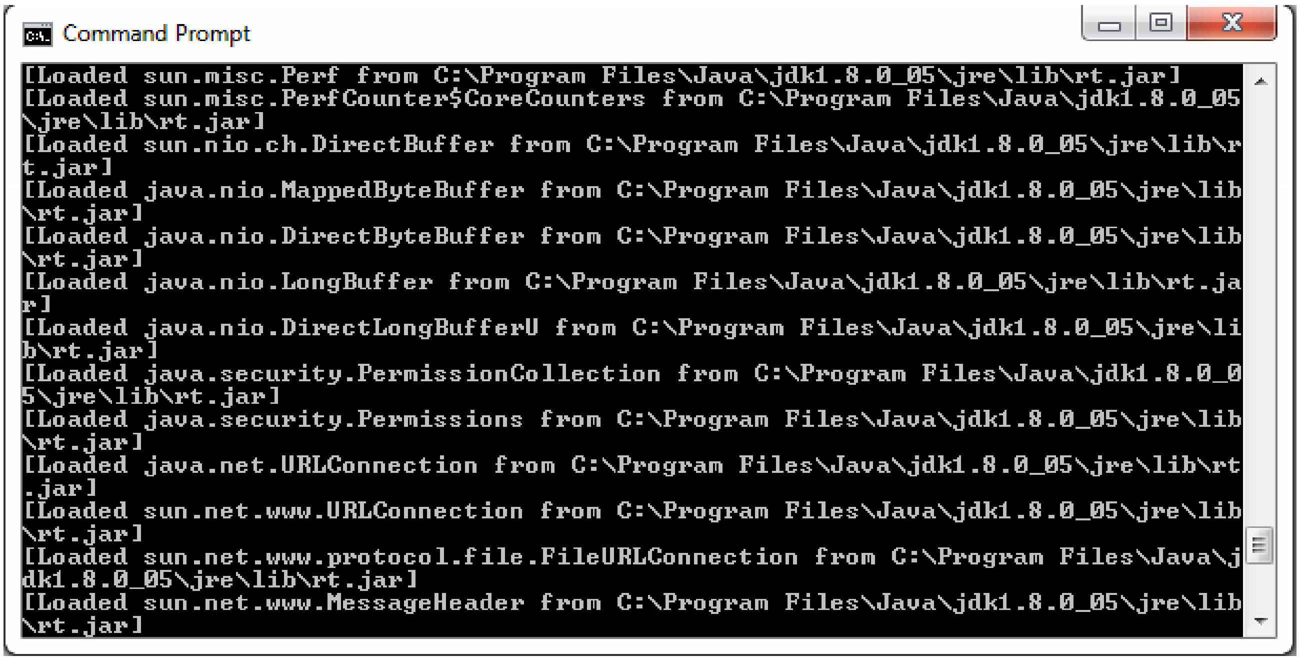
```
public class Demo
{
public static void main(String args[])
{
System.out.println("How are you?");
}
}
```

Compile and run the above code by using the following command:

```
javac Demo.java
java -verbose:class Demo
```

-verbose:class: It is used to display the information about classes being loaded by JVM. It is useful when using class loader for loading classes dynamically. The following figure shows the

⇧ SCROLL TO TOP



```
[Loaded sun.misc.Perf from C:\Program Files\Java\jdk1.8.0_05\jre\lib\rt.jar]
[Loaded sun.misc.PerfCounter$CoreCounters from C:\Program Files\Java\jdk1.8.0_05
\jre\lib\rt.jar]
[Loaded sun.nio.ch.DirectBuffer from C:\Program Files\Java\jdk1.8.0_05\jre\lib\r
t.jar]
[Loaded java.nio.MappedByteBuffer from C:\Program Files\Java\jdk1.8.0_05\jre\lib
\rt.jar]
[Loaded java.nio.DirectByteBuffer from C:\Program Files\Java\jdk1.8.0_05\jre\lib
\rt.jar]
[Loaded java.nio.LongBuffer from C:\Program Files\Java\jdk1.8.0_05\jre\lib\rt.ja
r]
[Loaded java.nio.DirectLongBufferU from C:\Program Files\Java\jdk1.8.0_05\jre\li
b\rt.jar]
[Loaded java.security.PermissionCollection from C:\Program Files\Java\jdk1.8.0_0
5\jre\lib\rt.jar]
[Loaded java.security.Permissions from C:\Program Files\Java\jdk1.8.0_05\jre\lib
\rt.jar]
[Loaded java.net.URLConnection from C:\Program Files\Java\jdk1.8.0_05\jre\lib\rt
.jar]
[Loaded sun.net.www.URLConnection from C:\Program Files\Java\jdk1.8.0_05\jre\lib
\rt.jar]
[Loaded sun.net.www.protocol.file.FileURLConnection from C:\Program Files\Java\j
dk1.8.0_05\jre\lib\rt.jar]
[Loaded sun.net.www.MessageHeader from C:\Program Files\Java\jdk1.8.0_05\jre\lib
\rt.jar]
```

We can observe that runtime classes required by the application class (Demo) are loaded first.

When classes are loaded

There are only two cases:

- When the new byte code is executed.
- When the byte code makes a static reference to a class. For example, **System.out**.

Static vs. Dynamic Class Loading

Classes are statically loaded with "new" operator. Dynamic class loading invokes the functions of a class loader at run time by using Class.forName() method.

Difference between loadClass() and Class.forName()

The loadClass() method loads only the class but does not initialize the object. While Class.forName() method initialize the object after loading it. For example, if you are using ClassLoader.loadClass() to load the JDBC driver, class loader does not allow to load JDBC driver.

The java.lang.Class.forName() method returns the Class Object coupled with the class or interfaces with the given string name. It throws ClassNotFoundException if the class is not found.

Example

In this example, java.lang.String class is loaded. It prints the class name, package name, and the names of all available methods of String class. We are using Class.forName() in the following

⇧ SCROLL TO TOP

Class<?>: Represents a Class object which can be of any type (? is a wildcard). The Class type contains meta-information about a class. For example, type of String.class is Class<String>. Use Class<?> if the class being modeled is unknown.

getDeclaredMethod(): Returns an array containing Method objects reflecting all the declared methods of the class or interface represented by this Class object, including public, protected, default (package) access, and private methods, but excluding inherited methods.

getName(): It returns the method name represented by this Method object, as a String.

```
import java.lang.reflect.Method;
public class ClassForNameExample
{
public static void main(String[] args)
{
try
{
Class<?> cls = Class.forName("java.lang.String");
System.out.println("Class Name: " + cls.getName());
System.out.println("Package Name: " + cls.getPackage());
Method[] methods = cls.getDeclaredMethods();
System.out.println("-----Methods of String class -------------");
for (Method method : methods)
{
System.out.println(method.getName());
}
}
catch (ClassNotFoundException e)
{
e.printStackTrace();
}
}
}
```

Output

```
Class Name: java.lang.String
Package Name: package java.lang
             String class -------------
```

⇧ SCROLL TO TOP

```
value
coder
equals
length
toString
hashCode
getChars
------
------
------
intern
isLatin1
checkOffset
checkBoundsOffCount
checkBoundsBeginEnd
access$100
access$200
```

← Prev Next →

Feedback

⇧ SCROLL TO TOP

ack to feedback@javatpoint.com

Help Others, Please Share

Learn Latest Tutorials

Digital Marketing	Elasticsearch	Entity Framework	Firewall
Functional Programming	Google Colab	Graph Theory	Groovy
Group Discussion	Informatica	Ionic	ITIL
IOS with Swift	Angular Material	Deep Learning	

Preparation

Aptitude	Reasoning	Verbal Ability	Interview Questions

⇧ SCROLL TO TOP