ClassLoader in Java

Difficulty Level : Easy • Last Updated : 07 Sep, 2020

The **Java ClassLoader** is a part of the **Java Runtime Environment** that dynamically loads Java classes into the **Java Virtual Machine**. The Java run time system does not need to know about files and file systems because of classloaders.

Java classes aren't loaded into memory all at once, but when required by an application. At this point, the **Java ClassLoader** is called by the **JRE** and these ClassLoaders load classes into memory dynamically.

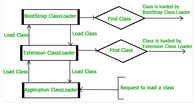
Types of ClassLoaders in Java

Not all classes are loaded by a single ClassLoader. Depending on the type of class and the path of class, the ClassLoader that loads that particular class is decided. To know the ClassLoader that loads a class the *getClassLoader()* method is used. All classes are loaded based on their names and if any of these classes are not found then it returns a **NoClassDefFoundError** or **ClassNotFoundException**.

A Java ClassLoader is of **three types**:

1. **BootStrap ClassLoader**: A Bootstrap ClassLoader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location ***rt.jar***. Bootstrap ClassLoader doesn't have any parent ClassLoaders. It is also called as the **Primodial ClassLoader**.
2. **Extension ClassLoader**: The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from ***jrv/lib/ext*** directory or any other directory pointed by the system property ***java.ext.dirs***.
3. **System ClassLoader**: An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable ***CLASSPATH, -classpath or -cp command line option***. The Application ClassLoader is a child class of Extension ClassLoader.

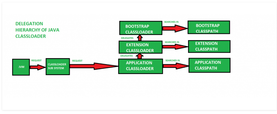
Note: The ClassLoader Delegation Hierarchy Model always functions in the order Application ClassLoader->Extension ClassLoader->Bootstrap ClassLoader. The Bootstrap ClassLoader is always given the higher priority, next is Extension ClassLoader and then Application ClassLoader.



Principles of functionality of a Java ClassLoader

Principles of functionality are the **set of rules** or features on which a Java ClassLoader works. There are **three principles of functionality**, they are:

1. **Delegation Model**: The Java Virtual Machine and the Java ClassLoader use an algorithm called the **Delegation Hierarchy Algorithm** to Load the classes into the Java file.
   The ClassLoader works based on a set of operations given by the delegation model. They are:
   - ClassLoader always follows the **Delegation Hierarchy Principle**.
   - Whenever JVM comes across a class, it checks whether that class is already loaded or not.
   - If the Class is already loaded in the method area then the JVM proceeds with execution.
   - If the class is not present in the method area then the JVM asks the Java ClassLoader Sub-System to load that particular class, then ClassLoader sub-system hands over the control to **Application ClassLoader**.
   - Application ClassLoader then delegates the request to Extension ClassLoader and the **Extension ClassLoader** in turn delegates the request to **Bootstrap ClassLoader**.
   - Bootstrap ClassLoader will search in the Bootstrap classpath (JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.
   - Extension ClassLoader searches for the class in the Extension Classpath (JDK/JRE/LIB/EXT) . If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.
   - Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a **ClassNotFoundException** exception is generated.



2. **Visibility Principle**: The **Visibility Principle** states that a class loaded by a parent ClassLoader is visible to the child ClassLoaders but a class loaded by a child ClassLoader is not visible to the parent ClassLoaders. Suppose a class GEEKS. class has been loaded by the Extension ClassLoader, then that class is only visible to the Extension ClassLoader and Application ClassLoader but not to the Bootstrap ClassLoader. If that class is again tried to load using Bootstrap ClassLoader it gives an exception ***Java.lang.ClassNotFoundException***.
3. **Uniqueness Property**: The **Uniqueness Property** ensures that the classes are unique and there is no repetition of classes. This also ensures that the classes loaded by parent classloaders are not loaded by the child classloaders. If the parent class loader isn't able to find the class, only then the current instance would attempt to do so itself.

Methods of Java.lang.ClassLoader

After the JVM requests for the class, a few steps are to be followed in order to load a class. The Classes are loaded as per the delegation model but there are a few important Methods or Functions that play a vital role in loading a Class.

1. **loadClass(String name, boolean resolve)**: This method is used to load the classes which are referenced by the JVM. It takes the name of the class as a parameter. This is of type loadClass(String, boolean).
2. **defineClass()**: The defineClass() method is a *final* method and cannot be overriden. This method is used to define a array of bytes as an instance of class. If the class is invalid then it throws **ClassFormatError**.
3. **findClass(String name)**: This method is used to find a specified class. This method only finds but doesn't load the class.
4. **findLoadedClass(String name)**: This method is used to verify whether the Class referenced by the JVM was previously loaded or not.
5. **Class.forName(String name, boolean initialize, ClassLoader loader)**: This method is used to load the class as well as initialize the class. This method also gives the option to choose any one of the ClassLoaders. If the ClassLoader parameter is NULL then Bootstrap ClassLoader is used.

Example: The following code is executed before a class is loaded:

```
protected synchronized Class<?>
loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    Class c = findLoadedClass(name);
    try {
        if (c == NULL) {
            if (parent != NULL) {
                c = parent.loadClass(name, false);
            }
            else {
                c = findBootstrapClass0(name);
            }
        }
        catch (ClassNotFoundException e)
        {
            System.out.println(e);
        }
    }
}
```

Note: If a class has already been loaded, it returns it. Otherwise, it delegates the search for the new class to the parent class loader. If the parent class loader doesn't find the class, **loadClass()** calls the method **findClass()** to find and load the class. The **findClass()** method searches for the class in the current **ClassLoader** if the class wasn't found by the parent **ClassLoader**.

Attention reader! Don't stop learning now. Get hold of all the important **Java Foundation** and Collections concepts with the **Fundamentals of Java and Java Collections Course** at a student-friendly price and become industry ready. To complete your preparation from learning a language to DS Algo and many more, please refer **Complete Interview Preparation Course**.

Next

How JVM Works - JVM Architecture?

Article Contributed By :

DonnanaManoj
@DonnanaManoj