Java Classloading

Senthalan Kanagalingam
Apr 7 · 3 min read

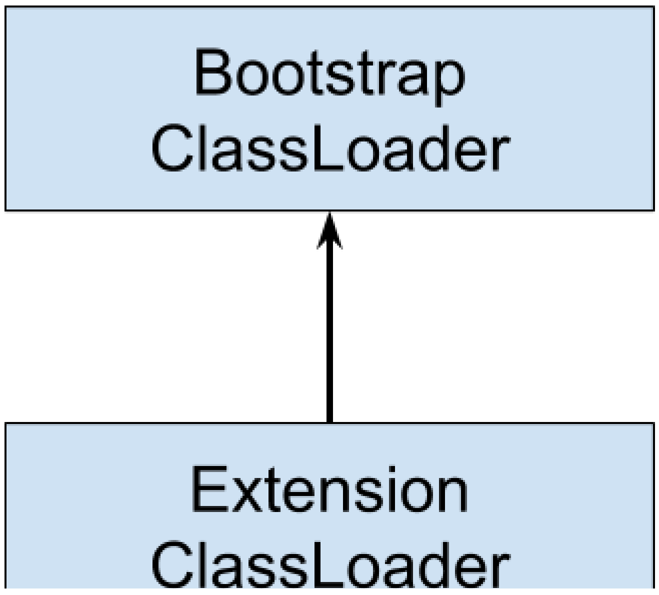
ClassLoader is one of the crucial components of Java. It is rarely used by the developers. If you want to play with it, you must have a deeper understanding of class loaders and class loading hierarchy. Almost all Java-based containers such as EJB or servlet containers implement custom ClassLoaders to support features like hot deployment and runtime platform extensibility. An in-depth understanding of ClassLoaders is important for developers when implementing such Java-based containers.

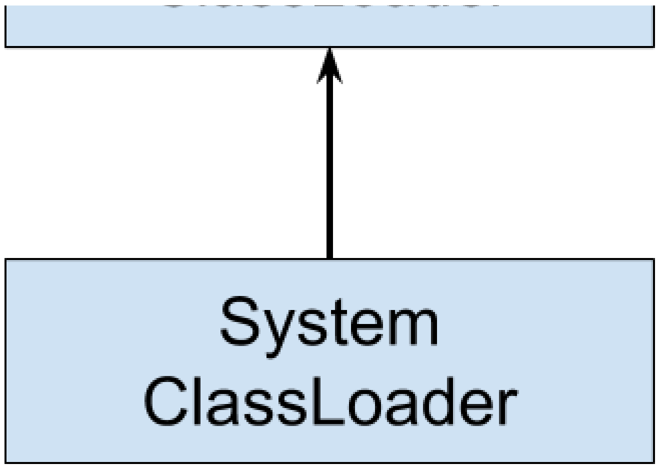
What is Classloading

Class loaders are responsible for **loading and executing classes during runtime dynamically to the JVM** (Java Virtual Machine). Also, they are part of the JRE (Java Runtime Environment). Hence, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders. Also, these Java classes aren't loaded into memory all at once, but when required by an application.

ClassLoaders are architected so that at start-up the JVM doesn't need to know anything about the classes that will be loaded at runtime.

Java class loading hierarchy





Java 8 class loading hierarchy

Java 8 has the following Class Loading hierarchy. It has the following class loaders,

Bootstrap ClassLoader

It loads JDK internal classes, typically loads rt.jar and other core classes for example java.lang.* package classes

Extensions ClassLoader

It loads classes from the JDK extensions directory, usually lib/ext directory of the JRE.

System(Application) ClassLoader

Loads classes from system classpath, that can be set while invoking a program using -cp or -classpath command line options.

How does the ClassLoading hierarchy work?

There are exactly two cases when the classes are statically loaded,

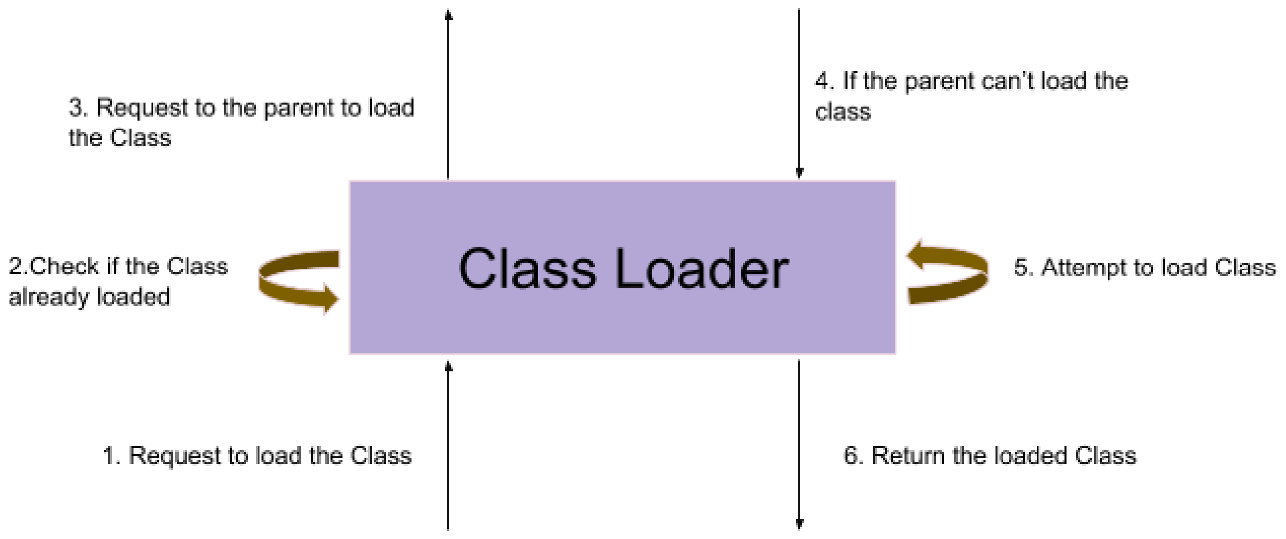
- When the new bytecode is executed (for example, **MyClass** *mc* = new **MyClass**())
- When the bytecodes make a static reference to a class (for example, **MyClass.**doS*omthing()*).

The very first class is loaded with the help of the static main() method declared in your class. All the subsequently loaded classes are loaded by the classes, which are already loaded and running.

Further classloaders follow these rules when loading classes,

- Check if the class was already loaded, then return the loaded class

- If not loaded, ask parent class loader(Not necessarily needed to be the parent) to load the class
- If parent class loader can't load the class, then that particular class loader attempt to load it



3. Request to the parent to load the Class
4. If the parent can't load the class
2.Check if the Class already loaded
Class Loader
5. Attempt to load Class
1. Request to load the Class
6. Return the loaded Class

Class loading rules

For example, suppose system classloader starts to load a **MyClass**. So first it will ask the extension class loader to find that class and extension class loader will ask the bootstrap classloader to search the class. If bootstrap classloader finds the class in bootstrap classes then that class is loaded into JVM. Otherwise, the responsibility for loading the given class is returned to extension classloader. Extension classloader tries to find the class in ext directory if found the class is loaded into JVM. Otherwise, the responsibility of loading the given class returns to system classloader and system class loader tries to find the class in classpath and loads into JVM. If the system class loader also can't load the class, **ClassNotFoundException** will be thrown.

We are not finished yet,

Java 9 Classloading

From Java 9 onward the following changes are made for the Java class loaders,

- **The bootstrap class loader** is still built-in to the Java Virtual Machine and represented by null in the ClassLoader API. It defines the classes in a handful of critical modules, such as java.base

- **The Extension classloader** has been renamed to **Platform class loader**. All classes in the Java SE Platform are guaranteed to be visible through the platform class loader. In addition, the classes in modules that are standardized under the Java Community Process but not part of the Java SE Platform are guaranteed to be visible through the platform class loader. The Platform class loader is not an instance of URLClassLoader, but an internal class.
- **The System(Application) classloader** is no longer an instance of URLClassLoader, but an internal class. It loads the classes in modules that are neither Java SE nor JDK modules.

In the next blog, we can walk through the class loading patterns of popular Java Application Servers and frameworks

Java Class Loader Java9 Heirarchy