Class Loader

A typical enterprise Java application may comprise thousands of source and dependency classes. To handle all of them in a proper way JVM introduces a special mechanism called **class loader**.

It is a part of **JRE** responsible for dynamic loading classes into memory. Understanding class loading allows to control this process and helps to avoid some types of exceptions.

First of all, let's recall that java code goes through 2 stages: a compilation from source code to byte code (.java -> .class) and a byte code interpretation.

The task of a class loader is finding the needed class through .class files from a disc and loading a representing object into the RAM. However, classes are not loaded in bulk mode on the application startup.

A class loader loads them on-demand during an interpretation starting with a class containing the main method.

The on-demand approach means that the class will be loaded on its first invocation. It can be a constructor call, e.g. new MyObject() or a static reference to a class, e.g. System.out

Internals

A **class loader** concept is represented by java.lang.ClassLoader abstract class. There are 3 standard ClassLoader implementations:

- **Bootstrap** loads JDK internal classes e.g. java.util package.
- **Extension/Platform** loads classes from JDK extensions.
- **Application/System** loads classes from application **class-path.**

One may ask what comes first if classes are loaded by a class loader and the ClassLoader itself is a class. First, JRE creates the **Bootstrap** ClassLoader which loads core classes.
Then, **Extension** ClassLoader is created with **Bootstrap** as a parent. It loads classes for extensions if such exist.
Finally, the **Application** ClassLoader is created with **Extension** as a parent. It is responsible for loading application classes from a classpath.

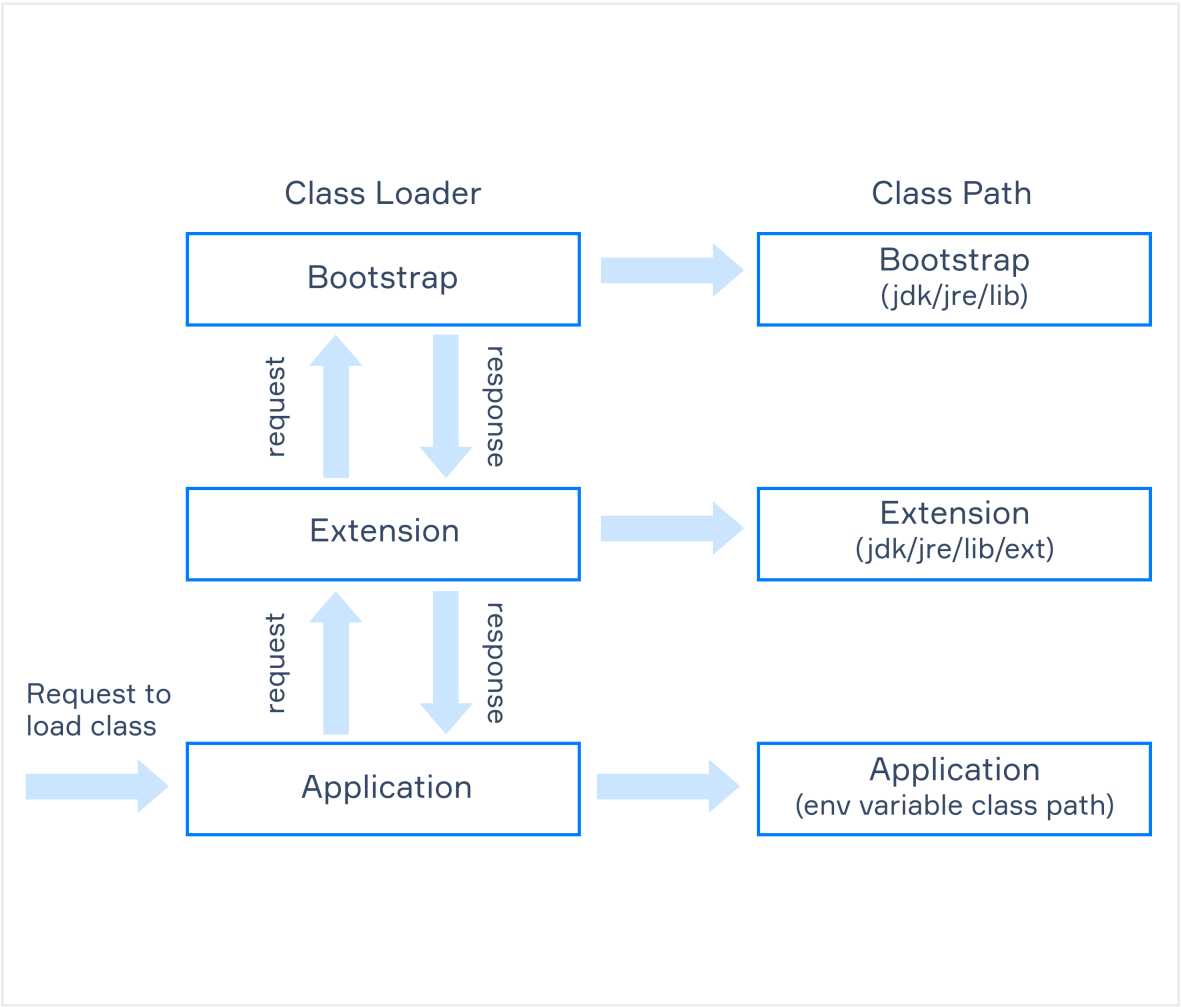
Each class loaded in memory is identified by a fully-qualified class name and ClassLoader that loaded this class. Moreover, Class has a method getClassLoader that returns the class loader which loads the given class.

```
import java.sql.SQLData;
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ClassLoader listClassLoader = ArrayList.class.getClassLoader();
        ClassLoader sqlClassLoader = SQLData.class.getClassLoader();
        ClassLoader mainClassLoader = Main.class.getClassLoader();

        System.out.println(listClassLoader); // null
        System.out.println(sqlClassLoader.getName()); // platform
        System.out.println(mainClassLoader.getName()); // app
    }
}
```

Delegation model

When the class loader receives a request for loading a class, it follows certain steps in order to resolve the class. Default behavior defined by JVM specification:

1. Check if the class has already been loaded
2. If not, delegate the request to a parent
3. If the parent class returns nothing, it attempts to find the class in its own classpath



Default logic can be overridden in custom class loaders. So web container class loaders will look for classes in the local **classpath** and only in case the class is not found will delegate resolving to a parent.

```
public class A {
   public static void main(String[] args) {
      B b = new B();
   }
}
public class B {
}
```

Launch the program by command:
java A

Let's go through key points of class loading during code execution:

1. Bootstrap ClassLoader is invoked by JRE on the start java process. It loads java internal packages.
2. Extension ClassLoader is invoked but loads nothing.

3. Application ClassLoader is invoked and loads class A.
4. When the constructor of class B is invoked ClassLoader of class A (Application ClassLoader) is invoked to load class B and delegates loading to Extension ClassLoader.
5. Extension ClassLoader is invoked and delegates loading to Bootstrap ClassLoader.
6. Bootstrap ClassLoader is invoked and tries to resolve the class, but finds nothing and returns control to Extension ClassLoader.
7. Extension ClassLoader finds nothing as well and returns control to Application ClassLoader.
8. Application ClassLoader resolves the class and loads it into memory.

When something goes wrong

The common root cause comes because runtime dependencies may differ from compile-time ones.

For instance,
a project may be compiled successfully, but some classes were not added to the classpath. In that case, a class loader cannot find a class. That leads to ClassNotFoundException or NoClassDefFoundError.

Another kind of exception happens because a project was compiled with one version of a class, but the classpath includes a different one. In that case NoSuchMethodError or NoSuchFieldError are thrown.

The getClassLoader() method returns the class loader for a particular class. If getClassLoader() returns null, it typically means that the class was loaded by the bootstrap class loader.