Class Loaders in Java

1. Introduction to Class Loaders

Class loaders are responsible for **loading Java 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. This is where class loaders come into the picture. They are responsible for loading classes into memory.

In this tutorial, we're going to talk about different types of built-in class loaders, how they work and an introduction to our own custom implementation.

2. Types of Built-in Class Loaders

Let's start by learning how different classes are loaded using various class loaders using a simple example:

```
public void printClassLoaders() throws ClassNotFoundException {
   System.out.println("Classloader of this class:" + PrintClassLoader.class.getClassLoader());
   System.out.println("Classloader of Logging:" + Logging.class.getClassLoader());
   System.out.println("Classloader of ArrayList:"  + ArrayList.class.getClassLoader());
}
```

When executed the above method prints:

```
Class loader of this class:sun.misc.Launcher$AppClassLoader@18b4aac2
Class loader of Logging:sun.misc.Launcher$ExtClassLoader@3caeaf62
Class loader of ArrayList:null
```

As we can see, there are three different class loaders here; **application**, **extension**, and **bootstrap** (displayed as null).

The application class loader loads the class where the example method is contained. An application or system class loader loads our own files in the classpath.

Next, the extension one loads the Logging class. Extension class loaders load classes that are an extension of the standard core Java classes.

Finally, the bootstrap one loads the ArrayList class. A bootstrap or primordial class loader is the parent of all the others.

However, we can see that the last out, for the ArrayList it displays null in the output. This is because the bootstrap class loader is written in native code, not Java – so it doesn't show up as a Java class. Due to this reason, the behavior of the bootstrap class loader will differ across JVMs.

Let's now discuss more in detail about each of these class loaders.

2.1. Bootstrap Class Loader

Java classes are loaded by an instance of java.lang.ClassLoader. However, class loaders are classes themselves. Hence, the question is, who loads the java.lang.ClassLoader itself?

This is where the bootstrap or primordial class loader comes into the picture.

It's mainly responsible for loading JDK internal classes, typically rt.jar and other core libraries located in $JAVA_HOME/jre/lib directory. Additionally, Bootstrap class loader serves as a parent of all the other ClassLoader instances.

This bootstrap class loader is part of the core JVM and is written in native code as pointed out in the above example. Different platforms might have different implementations of this particular class loader.

2.2. Extension Class Loader

The extension class loader is a child of the bootstrap class loader and takes care of loading the extensions of the standard core Java classes so that it's available to all applications running on the platform.

Extension class loader loads from the JDK extensions directory, usually $JAVA_HOME/lib/ext directory or any other directory mentioned in the java.ext.dirs system property.

2.3. System Or Application Class Loader

The system or application class loader, on the other hand, takes care of loading all the application level classes into the JVM. It loads files found in the classpath environment variable, -classpath or -cp command line option. Also, it's a child of Extensions classloader.

3. How do Class Loaders Work?

Class loaders are part of the Java Runtime Environment. When the JVM requests a class, the class loader tries to locate the class and load the class definition into the runtime using the fully qualified class name.

The java.lang.ClassLoader.loadClass() method is responsible for loading the class definition into runtime. It tries to load the class based on a fully qualified name.

If the class isn't already loaded, it delegates the request to the parent class loader. This process happens recursively.

Eventually, if the parent class loader doesn't find the class, then the child class will call java.net.URLClassLoader.findClass() method to look for classes in the file system itself.

If the last child class loader isn't able to load the class either, it throws java.lang.NoClassDefFoundError or java.lang.ClassNotFoundException.

Let's look at an example of output when ClassNotFoundException is thrown.

```
java.lang.ClassNotFoundException: com.baeldung.classloader.SampleClassLoader
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:348)
```

If we go through the sequence of events right from calling java.lang.Class.forName(), we can understand that it first tries to load the class through parent class loader and then java.net.URLClassLoader.findClass() to look for the class itself.

When it still doesn't find the class, it throws a ClassNotFoundException.

There are three important features of class loaders.

3.1. Delegation Model

Class loaders follow the delegation model where on request to find a class or resource, a ClassLoader instance will delegate the search of the class or resource to the parent class loader.

Let's say we have a request to load an application class into the JVM. The system class loader first delegates the loading of that class to its parent extension class loader which in turn delegates it to the bootstrap class loader.

Only if the bootstrap and then the extension class loader is unsuccessful in loading the class, the system class loader tries to load the class itself.

3.2. Unique Classes

As a consequence of the delegation model, it's easy to ensure unique classes as we always try to delegate upwards.

If the parent class loader isn't able to find the class, only then the current instance would attempt to do so itself.

3.3. Visibility

In addition, children class loaders are visible to classes loaded by its parent class loaders.

For instance, classes loaded by the system class loader have visibility into classes loaded by the extension and Bootstrap class loaders but not vice-versa.

To illustrate this, if Class A is loaded by an application class loader and class B is loaded by the extensions class loader, then both A and B classes are visible as far as other classes loaded by Application class loader are concerned.

Class B, nonetheless, is the only class visible as far as other classes loaded by the extension class loader are concerned.

4. Custom ClassLoader

The built-in class loader would suffice in most of the cases where the files are already in the file system.

However, in scenarios where we need to load classes out of the local hard drive or a network, we may need to make use of custom class loaders.

In this section, we'll cover some other uses cases for custom class loaders and we'll demonstrate how to create one.

4.1. Custom Class Loaders Use-Cases

Custom class loaders are helpful for more than just loading the class during runtime, a few use cases might include:

1. Helping in modifying the existing bytecode, e.g. weaving agents
2. Creating classes dynamically suited to the user's needs. e.g in JDBC, switching between different driver implementations is done through dynamic class loading.
3. Implementing a class versioning mechanism while loading different bytecodes for classes with same names and packages. This can be done either through URL class loader (load jars via URLs) or custom class loaders.

There are more concrete examples where custom class loaders might come in handy.

Browsers, for instance, use a custom class loader to load executable content from a website. A browser can load applets from different web pages using separate class loaders. The applet viewer which is used to run applets contains a *ClassLoader* that accesses a website on a remote server instead of looking in the local file system.

And then loads the raw bytecode files via HTTP, and turns them into classes inside the JVM. Even if these **applets have the same name, they are considered as different components if loaded by different class loaders**.

Now that we understand why custom class loaders are relevant, let's implement a subclass of *ClassLoader* to extend and summarise the functionality of how the JVM loads classes.

4.2. Creating our Custom Class Loader

For illustration purposes, let's say we need to load classes through FTP, the loading of the class isn't possible through built-in class loaders since it isn't present in the classpath at the time:

```
public class CustomClassLoader extends ClassLoader {
   public CustomClassLoader(ClassLoader parent) {
      super(parent);
   }
   public Class getClass(String name) throws ClassNotFoundException {
      byte[] b = loadClassFromFTP(name);
      return defineClass(name, b, 0, b.length);
   }

   @Override
   public Class loadClass(String name) throws ClassNotFoundException {

      if (name.startsWith("com.baeldung")) {
         System.out.println("Loading Class from Custom Class Loader");
```

```
            return getClass(name);
        }
        return super.loadClass(name);
    }

    private byte[] loadClassFromFTP(String fileName) {
        // Returns a byte array from specified file.
    }
}
```

In the above example, we defined a custom class loader that extends the default class loader and loads files from the com.baeldung package.

We set the parent class loader in the constructor. Then we load the class using FTP specifying a fully qualified class name as an input.

5. Understanding java.lang.ClassLoader

Let's discuss a few essential methods from the java.lang.ClassLoader class to get a clearer picture of how it works.

5.1. The loadClass() Method

```
public Class<?> loadClass(String name, boolean resolve) throws ClassNotFoundException {
```

This method is responsible for loading the class given a name parameter. The name parameter refers to the fully qualified class name.

The Java Virtual Machine invokes loadClass() method to resolve class references setting resolve to true. However, it isn't always necessary to resolve a class. If we only need to determine if the class exists or not, then resolve parameter is set to false.

This method serves as an entry point for the class loader.
We can try to understand the internal working of the loadClass() method from the source code of java.lang.ClassLoader:

```
protected Class<?> loadClass(String name, boolean resolve)
  throws ClassNotFoundException {

    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }

            if (c == null) {
                // If still not found, then invoke findClass in order
                // to find the class.
                c = findClass(name);
            }
        }
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
}
```

The default implementation of the method searches for classes in the following order:
1. Invokes the findLoadedClass(String) method to see if the class is already loaded.
2. Invokes the loadClass(String) method on the parent class loader.
3. Invoke the findClass(String) method to find the class.

5.2. The defineClass() Method

protected final Class<?> defineClass(String name, byte[] b, int off, int len) throws ClassFormatError

This method is responsible for the conversion of an array of bytes into an instance of a class. And before we use the class, we need to resolve it.

In case data didn’t contain a valid class, it throws a ClassFormatError.

Also, we can’t override this method since it’s marked as final.

5.3. The findClass() Method

protected Class<?> findClass(String name) throws ClassNotFoundException

This method finds the class with the fully qualified name as a parameter. We need to override this method in custom class loader implementations that follow the delegation model for loading classes.

Also, loadClass() invokes this method if the parent class loader couldn’t find the requested class.

The default implementation throws a ClassNotFoundException if no parent of the class loader finds the class.

5.4. The getParent() Method

public final ClassLoader getParent()

This method returns the parent class loader for delegation.
Some implementations like the one seen before in Section 2. use null to represent the bootstrap class loader.

5.5. The getResource() Method

public URL getResource(String name)
This method tries to find a resource with the given name.

It will first delegate to the parent class loader for the resource. If the parent is null, the path of the class loader built into the virtual machine is searched.

If that fails, then the method will invoke findResource(String) to find the resource. The resource name specified as an input can be relative or absolute to the classpath.

It returns an URL object for reading the resource, or null if the resource could not be found or if the invoker doesn't have adequate privileges to return the resource.

It's important to note that Java loads resources from the classpath.

Finally, resource loading in Java is considered location-independent as it doesn't matter where the code is running as long as the environment is set to find the resources.

6. Context Classloaders

In general, context class loaders provide an alternative method to the class-loading delegation scheme introduced in J2SE.

Like we've learned before, classloaders in a JVM follow a hierarchical model such that every class loader has a single parent with the exception of the bootstrap class loader.

However, sometimes when JVM core classes need to dynamically load classes or resources provided by application developers, we might encounter a problem.

For example, in JNDI the core functionality is implemented by bootstrap classes in rt.jar. But these JNDI classes may load JNDI providers implemented by independent vendors (deployed in the application classpath). This scenario calls for the bootstrap class loader (parent class loader) to load a class visible to application loader (child class loader).

J2SE delegation doesn't work here and to get around this problem, we need to find alternative ways of class loading. And it can be achieved using thread context loaders.

The java.lang.Thread class has a method getContextClassLoader() that returns the ContextClassLoader for the particular thread. The ContextClassLoader is provided by the creator of the thread when loading resources and classes.

If the value isn't set, then it defaults to the class loader context of the parent thread.

7. Conclusion

Class loaders are essential to execute a Java program. We've provided a good introduction as part of this article.

We talked about different types of class loaders namely – Bootstrap, Extensions and System class loaders. Bootstrap serves as a parent for all of them and is responsible for loading the JDK internal classes. Extensions and system, on the other hand, loads classes from the Java extensions directory and classpath respectively.

Then we talked about how class loaders work and we discussed some features such as delegation, visibility, and uniqueness followed by a brief explanation of how to create a custom one. Finally, we provided an introduction to Context class loaders.