

(/)

Class Loaders in Java

Last modified: February 5, 2022

by baeldung (<https://www.baeldung.com/author/baeldung/>)

Java (<https://www.baeldung.com/category/java/>) +

Core Java (<https://www.baeldung.com/tag/core-java/>)

Get started with Spring 5 and Spring Boot 2, through the *Learn Spring* course:

> CHECK OUT THE COURSE (</ls-course-start>)

1. Introduction to Class Loaders

Class loaders are responsible for **loading Java classes dynamically to the JVM (Java Virtual Machine) during runtime**. They're also part of the JRE (Java Runtime Environment). Therefore, the JVM doesn't need to know about the underlying files or file systems in order to run Java programs thanks to class loaders.

Furthermore, these Java classes aren't loaded into memory all at once, but rather when they're required by an application. This is where class loaders come into the picture. They're responsible for loading classes into memory.

In this tutorial, we'll talk about different types of built-in class loaders and how they work. Then we'll introduce our own custom implementation.

Further reading:

Understanding Memory Leaks in Java (/java-memory-leaks)

Learn what memory leaks are in Java, how to recognize them at runtime, what causes them, and strategies for preventing them.

Read more (/java-memory-leaks) →

ClassNotFoundException vs NoClassDefFoundError (/java-classnotfoundexception-and-noclassdeffounderror)

Learn about the differences between ClassNotFoundException and NoClassDefFoundError.

Read more (/java-classnotfoundexception-and-noclassdeffounderror) →

2. Types of Built-in Class Loaders

Let's start by learning how we can load different classes using various class loaders:

```
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:

(<https://freestar.com/?>

```
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 class loader loads the *Logging* class. **Extension class loaders load classes that are an extension of the standard core Java classes.**

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

However, we can see that 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.** As a result, the behavior of the bootstrap class loader will differ across JVMs.

 (https://freestar.com/?

Now let's discuss each of these class loaders in more detail.

2.1. Bootstrap Class Loader

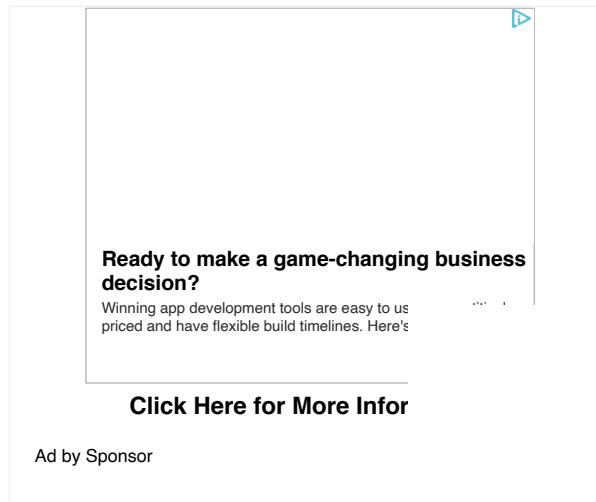
Java classes are loaded by an instance of *java.lang.ClassLoader*. However, class loaders are classes themselves. So the question is, who loads the *java.lang.ClassLoader* itself?

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

It's mainly responsible for loading JDK internal classes, typically *rt.jar* and other core libraries located in the *\$JAVA_HOME/jre/lib* directory.

Additionally, the **Bootstrap class loader serves as the 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.



(<https://freestar.com/?>

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 they're available to all applications running on the platform.

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

2.3. System 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**. It's also a child of the extensions class loader.

3. How Do Class Loaders Work?

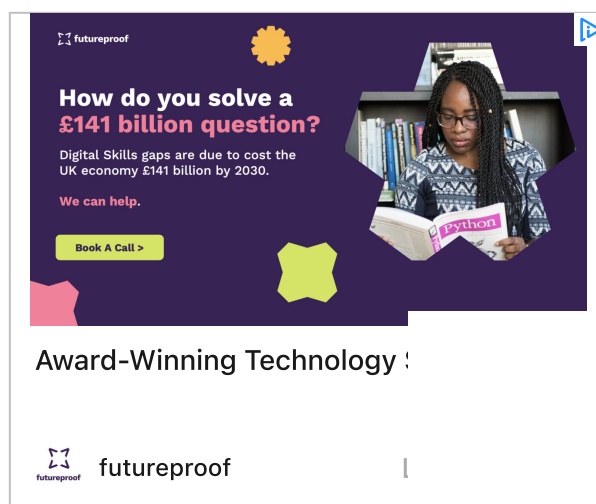
Class loaders are part of the Java Runtime Environment. When the JVM requests a class, the class loader tries to locate the class and load the class definition into the runtime using the fully qualified class name.

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 the *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*.
(/java-classnotfoundexception-and-noclassdeffoundererror)



(<https://freestar.com/?>

Let's look at an example of the 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 see that it first tries to load the class through the 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*.

Now let's examine 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 are 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 will the current instance attempt to do so itself.

3.3. Visibility

In addition, **children class loaders are visible to classes loaded by their 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 the 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 the application class loader are concerned.

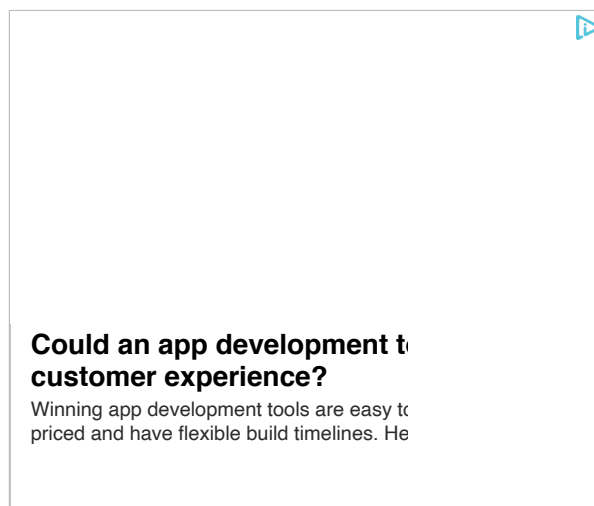
Class B, however, is the only class visible to other classes loaded by the extension class loader.

4. Custom ClassLoader

The built-in class loader is sufficient for most 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 use cases for custom class loaders and demonstrate how to create one.



(<https://freestar.com/?>

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 to modify 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 the same names and packages. This can be done either through a URL class loader (load jars via URLs) or custom class loaders.

Below 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.

It 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're considered 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 from a file using a custom class loader.

We need to extend the *ClassLoader* class and override the *findClass()* method:

```

public class CustomClassLoader extends ClassLoader {

    @Override
    public Class findClass(String name) throws ClassNotFoundException {
        byte[] b = loadClassFromFile(name);
        return defineClass(name, b, 0, b.length);
    }

    private byte[] loadClassFromFile(String fileName) {
        InputStream inputStream =
getClass().getClassLoader().getResourceAsStream(
        fileName.replace('.', File.separatorChar) + ".class");
        byte[] buffer;
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        int nextValue = 0;
        try {
            while ( (nextValue = inputStream.read()) != -1 ) {
                byteStream.write(nextValue);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        buffer = byteStream.toByteArray();
        return buffer;
    }
}

```

In the above example, we defined a custom class loader that extends the default class loader, and loads a byte array from the specified file.



(<https://freestar.com/?>

anding&utm_medium=banner&utm_source=baeldung.com&utm_content=ba

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 the *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 the *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. Before we use the class, we need to resolve it.

If the data doesn'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.

In addition, *loadClass()* invokes this method if the parent class loader can'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 can't be found or 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 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 the 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 the application loader (child class loader).



(<https://freestar.com/?>

anding&utm_medium=banner&utm_source=baeldung.com&utm_content=ba

J2SE delegation doesn't work here, and to get around this problem, we need to find alternative ways of class loading. This 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. In this article, we provided a good introduction to them.

We discussed the 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, load classes from the Java extensions directory and classpath, respectively.

We also learned how class loaders work and examined some features, such as delegation, visibility, and uniqueness. Then we briefly explained how to create a custom class loader. Finally, we provided an introduction to Context class loaders.

As always, the source code for these examples can be found over on GitHub (<https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-jvm>).

Get started with Spring 5 and Spring Boot 2, through the *Learn Spring* course:

>> CHECK OUT THE COURSE (/ls-course-end)



Learning to build your API
with Spring?

Download the E-book (/rest-api-spring-guide)

Comments are closed on this article!



(<https://freestar.com/?>

COURSES

[ALL COURSES \(/ALL-COURSES\)](#)

[ALL BULK COURSES \(/ALL-BULK-COURSES\)](#)

[THE COURSES PLATFORM \(HTTPS://COURSES.BAELDUNG.COM\)](https://courses.baeldung.com)

SERIES

[JAVA "BACK TO BASICS" TUTORIAL \(/JAVA-TUTORIAL\)](#)

[JACKSON JSON TUTORIAL \(/JACKSON\)](#)

[APACHE HTTPCLIENT TUTORIAL \(/HTTPCLIENT-GUIDE\)](#)

[REST WITH SPRING TUTORIAL \(/REST-WITH-SPRING-SERIES\)](#)

[SPRING PERSISTENCE TUTORIAL \(/PERSISTENCE-WITH-SPRING-SERIES\)](#)

[SECURITY WITH SPRING \(/SECURITY-SPRING\)](#)

[SPRING REACTIVE TUTORIALS \(/SPRING-REACTIVE-GUIDE\)](#)

ABOUT

[ABOUT BAELDUNG \(/ABOUT\)](#)

[THE FULL ARCHIVE \(/FULL_ARCHIVE\)](#)

[EDITORS \(/EDITORS\)](#)

[JOBS \(/TAG/ACTIVE-JOB/\)](#)

[OUR PARTNERS \(/PARTNERS\)](#)

[PARTNER WITH BAELDUNG \(/ADVERTISE\)](#)

[TERMS OF SERVICE \(/TERMS-OF-SERVICE\)](#)

[PRIVACY POLICY \(/PRIVACY-POLICY\)](#)

[COMPANY INFO \(/BAELDUNG-COMPANY-INFO\)](#)

[CONTACT \(/CONTACT\)](#)