**What is Java ClassLoader?**

We know that Java Program runs on [Java Virtual Machine](https://www.digitalocean.com/community/tutorials/difference-jdk-vs-jre-vs-jvm) (JVM). When we compile a Java Class, JVM creates the bytecode, which is platform and machine-independent. The bytecode is stored in a **.class file**. When we try to use a class, the ClassLoader loads it into the memory.

**Built-in ClassLoader Types**

There are three types of built-in ClassLoader in Java.

1. **Bootstrap Class Loader** – It loads JDK internal classes. It loads rt.jar and other core classes for example java.lang.\* package classes.
2. **Extensions Class Loader** – It loads classes from the JDK extensions directory, usually $JAVA\_HOME/lib/ext directory.
3. **System Class Loader** – This classloader loads classes from the current classpath. We can set classpath while invoking a program using -cp or -classpath command line option.

**ClassLoader Hierarchy**

ClassLoader is hierarchical in loading a class into memory. Whenever a request is raised to load a class, it delegates it to the parent classloader. This is how uniqueness is maintained in the runtime environment. If the parent class loader doesn’t find the class then the class loader itself tries to load the class. Let’s understand this by executing the below java program.

package com.journaldev.classloader;

public class ClassLoaderTest {

public static void main(String[] args) {

System.out.println("class loader for HashMap: "

+ java.util.HashMap.class.getClassLoader());

System.out.println("class loader for DNSNameService: "

+ sun.net.spi.nameservice.dns.DNSNameService.class

.getClassLoader());

System.out.println("class loader for this class: "

+ ClassLoaderTest.class.getClassLoader());

System.out.println(com.mysql.jdbc.Blob.class.getClassLoader());

}

}

Output:

class loader for HashMap: null

class loader for DNSNameService: sun.misc.Launcher$ExtClassLoader@7c354093

class loader for this class: sun.misc.Launcher$AppClassLoader@64cbbe37

sun.misc.Launcher$AppClassLoader@64cbbe37

**How Java ClassLoader Works?**

Let’s understand the working of class loaders from the above program output.

* The [java.util.HashMap](https://www.digitalocean.com/community/tutorials/java-hashmap) ClassLoader is coming as null, which reflects Bootstrap ClassLoader. The DNSNameService class ClassLoader is ExtClassLoader. Since the class itself is in CLASSPATH, System ClassLoader loads it.
* When we are trying to load HashMap, our System ClassLoader delegates it to the Extension ClassLoader. The extension class loader delegates it to the Bootstrap ClassLoader. The bootstrap class loader finds the HashMap class and loads it into the JVM memory.
* The same process is followed for the DNSNameService class. But, the Bootstrap ClassLoader is not able to locate it since it’s in $JAVA\_HOME/lib/ext/dnsns.jar. Hence, it gets loaded by Extensions Classloader.
* The Blob class is included in the MySql [JDBC](https://www.digitalocean.com/community/tutorials/jdbc-tutorial) Connector jar (mysql-connector-java-5.0.7-bin.jar), which is present in the build path of the project. It’s also getting loaded by the System Classloader.
* The classes loaded by a child class loader have visibility into classes loaded by its parent class loaders. So classes loaded by System Classloader have visibility into classes loaded by Extensions and Bootstrap Classloader.
* If there are sibling class loaders then they can’t access classes loaded by each other.

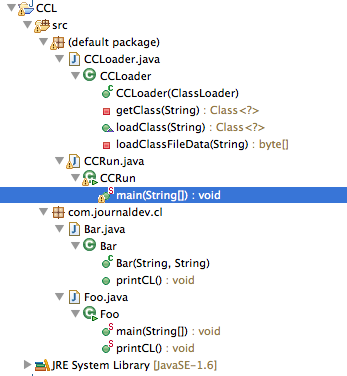
**Why write a Custom ClassLoader in Java?**

Java default ClassLoader can load classes from the local file system, which is good enough for most of the cases. But, if you are expecting a class at the runtime or from the FTP server or via third party web service at the time of loading the class, then you have to extend the existing class loader. For example, AppletViewers load the classes from a remote web server.

**Java ClassLoader Methods**

* When JVM requests for a class, it invokes loadClass() function of the ClassLoader by passing the fully classified name of the Class.
* The loadClass() function calls the findLoadedClass() method to check that the class has been already loaded or not. It’s required to avoid loading the same class multiple times.
* If the Class is not already loaded, then it will delegate the request to parent ClassLoader to load the class.
* If the parent ClassLoader doesn’t find the class then it will invoke findClass() method to look for the classes in the file system.

**Java Custom ClassLoader Example**

We will create our own ClassLoader by extending the ClassLoader class and overriding the loadClass(String name) method. If the class name will start from com.journaldev then we will load it using our custom class loader or else we will invoke the parent ClassLoader loadClass() method to load the class.

**1. CCLoader.java**

This is our custom class loader with below methods.

1. private byte[] loadClassFileData(String name): This method will read the class file from file system to byte array.
2. private Class<?> getClass(String name): This method will call the loadClassFileData() function and by invoking the parent defineClass() method, it will generate the Class and return it.
3. public Class<?> loadClass(String name): This method is responsible for loading the class. If the class name starts with com.journaldev (Our sample classes) then it will load it using getClass() method or else it will invoke the parent loadClass() function to load it.
4. public CCLoader(ClassLoader parent): This is the constructor, which is responsible for setting the parent ClassLoader.

import java.io.DataInputStream;

import java.io.File;

import java.io.IOException;

import java.io.InputStream;

/\*\*

\* Our Custom ClassLoader to load the classes. Any class in the com.journaldev

\* package will be loaded using this ClassLoader. For other classes, it will delegate the request to its Parent ClassLoader.

\*

\*/

public class CCLoader extends ClassLoader {

/\*\*

\* This constructor is used to set the parent ClassLoader

\*/

public CCLoader(ClassLoader parent) {

super(parent);

}

/\*\*

\* Loads the class from the file system. The class file should be located in

\* the file system. The name should be relative to get the file location

\*

\* @param name

\* Fully Classified name of the class, for example, com.journaldev.Foo

\*/

private Class getClass(String name) throws ClassNotFoundException {

String file = name.replace('.', File.separatorChar) + ".class";

byte[] b = null;

try {

// This loads the byte code data from the file

b = loadClassFileData(file);

// defineClass is inherited from the ClassLoader class

// that converts byte array into a Class. defineClass is Final

// so we cannot override it

Class c = defineClass(name, b, 0, b.length);

resolveClass(c);

return c;

} catch (IOException e) {

e.printStackTrace();

return null;

}

}

/\*\*

\* Every request for a class passes through this method. If the class is in

\* com.journaldev package, we will use this classloader or else delegate the

\* request to parent classloader.

\*

\*

\* @param name

\* Full class name

\*/

@Override

public Class loadClass(String name) throws ClassNotFoundException {

System.out.println("Loading Class '" + name + "'");

if (name.startsWith("com.journaldev")) {

System.out.println("Loading Class using CCLoader");

return getClass(name);

}

return super.loadClass(name);

}

/\*\*

\* Reads the file (.class) into a byte array. The file should be

\* accessible as a resource and make sure that it's not in Classpath to avoid

\* any confusion.

\*

\* @param name

\* Filename

\* @return Byte array read from the file

\* @throws IOException

\* if an exception comes in reading the file

\*/

private byte[] loadClassFileData(String name) throws IOException {

InputStream stream = getClass().getClassLoader().getResourceAsStream(

name);

int size = stream.available();

byte buff[] = new byte[size];

DataInputStream in = new DataInputStream(stream);

in.readFully(buff);

in.close();

return buff;

}

}

**2. CCRun.java**

This is our test class with the [main function](https://www.digitalocean.com/community/tutorials/public-static-void-main-string-args-java-main-method). We are creating an instance of our ClassLoader and loading sample classes using its loadClass() method. After loading the class, we are using [Java Reflection API](https://www.digitalocean.com/community/tutorials/java-reflection-example-tutorial) to invoke its methods.

import java.lang.reflect.Method;

public class CCRun {

public static void main(String args[]) throws Exception {

String progClass = args[0];

String progArgs[] = new String[args.length - 1];

System.arraycopy(args, 1, progArgs, 0, progArgs.length);

CCLoader ccl = new CCLoader(CCRun.class.getClassLoader());

Class clas = ccl.loadClass(progClass);

Class mainArgType[] = { (new String[0]).getClass() };

Method main = clas.getMethod("main", mainArgType);

Object argsArray[] = { progArgs };

main.invoke(null, argsArray);

// Below method is used to check that the Foo is getting loaded

// by our custom class loader i.e CCLoader

Method printCL = clas.getMethod("printCL", null);

printCL.invoke(null, new Object[0]);

}

}

**3. Foo.java and Bar.java**

These are our test classes that are getting loaded by our custom classloader. They have a printCL() method, which is getting invoked to print the ClassLoader information. Foo class will be loaded by our custom class loader. Foo uses Bar class, so Bar class will also be loaded by our custom class loader.

package com.journaldev.cl;

public class Foo {

static public void main(String args[]) throws Exception {

System.out.println("Foo Constructor >>> " + args[0] + " " + args[1]);

Bar bar = new Bar(args[0], args[1]);

bar.printCL();

}

public static void printCL() {

System.out.println("Foo ClassLoader: "+Foo.class.getClassLoader());

}

}

package com.journaldev.cl;

public class Bar {

public Bar(String a, String b) {

System.out.println("Bar Constructor >>> " + a + " " + b);

}

public void printCL() {

System.out.println("Bar ClassLoader: "+Bar.class.getClassLoader());

}

}

**4. Java Custom ClassLoader Execution Steps**

First of all, we will compile all the classes through the command line. After that, we will run the CCRun class by passing three arguments. The first argument is the fully classified name for Foo class that will get loaded by our class loader. Other two arguments are passed along to the Foo class main function and Bar constructor. The execution steps and the output will be like below.

$ javac -cp . com/journaldev/cl/Foo.java

$ javac -cp . com/journaldev/cl/Bar.java

$ javac CCLoader.java

$ javac CCRun.java

CCRun.java:18: warning: non-varargs call of varargs method with inexact argument type for last parameter;

cast to java.lang.Class<?> for a varargs call

cast to java.lang.Class<?>[] for a non-varargs call and to suppress this warning

Method printCL = clas.getMethod("printCL", null);

^

1 warning

$ java CCRun com.journaldev.cl.Foo 1212 1313

Loading Class 'com.journaldev.cl.Foo'

Loading Class using CCLoader

Loading Class 'java.lang.Object'

Loading Class 'java.lang.String'

Loading Class 'java.lang.Exception'

Loading Class 'java.lang.System'

Loading Class 'java.lang.StringBuilder'

Loading Class 'java.io.PrintStream'

Foo Constructor >>> 1212 1313

Loading Class 'com.journaldev.cl.Bar'

Loading Class using CCLoader

Bar Constructor >>> 1212 1313

Loading Class 'java.lang.Class'

Bar ClassLoader: CCLoader@71f6f0bf

Foo ClassLoader: CCLoader@71f6f0bf

$

If you look at the output, it’s trying to load com.journaldev.cl.Foo class. Since it’s extending java.lang.Object class, it’s trying to load Object class first. So the request is coming to CCLoader loadClass method, which is delegating it to the parent class. So the parent class loaders are loading the Object, String, and other java classes. Our ClassLoader is only loading Foo and Bar class from the file system. It’s clear from the output of the printCL() function. We can change the loadClassFileData() functionality to read the byte array from FTP Server or by invoking any third party service to get the class byte array on the fly. I hope that the article will be useful in understanding Java ClassLoader working and how we can extend it to do a lot more than just taking it from the file system.

**Making Custom ClassLoader as Default ClassLoader**

We can make our custom class loader as the default one when JVM starts by using Java Options. For example, I will run the ClassLoaderTest program once again after providing the java classloader option.

$ javac -cp .:../lib/mysql-connector-java-5.0.7-bin.jar com/journaldev/classloader/ClassLoaderTest.java

$ java -cp .:../lib/mysql-connector-java-5.0.7-bin.jar -Djava.system.class.loader=CCLoader com.journaldev.classloader.ClassLoaderTest

Loading Class 'com.journaldev.classloader.ClassLoaderTest'

Loading Class using CCLoader

Loading Class 'java.lang.Object'

Loading Class 'java.lang.String'

Loading Class 'java.lang.System'

Loading Class 'java.lang.StringBuilder'

Loading Class 'java.util.HashMap'

Loading Class 'java.lang.Class'

Loading Class 'java.io.PrintStream'

class loader for HashMap: null

Loading Class 'sun.net.spi.nameservice.dns.DNSNameService'

class loader for DNSNameService: sun.misc.Launcher$ExtClassLoader@24480457

class loader for this class: CCLoader@38503429

Loading Class 'com.mysql.jdbc.Blob'

sun.misc.Launcher$AppClassLoader@2f94ca6c

$

The CCLoader is loading the ClassLoaderTest class because its in com.journaldev package.

You can download the ClassLoader example code from our [GitHub Repository](https://github.com/journaldev/journaldev/tree/master/CoreJavaProjects/JavaPrograms/src).