Java Interview Questions: Understanding and Extending Java ClassLoader

The **Java ClassLoader** is one of the crucial but rarely used components of Java in Project Development. Personally I have never extended ClassLoader in any of my projects but the idea of having my own ClassLoader that can customize the Java Class Loading thrills me.

This article will provide an overview of Java Class Loading and then move forward to create a custom ClassLoader and use it.

What is a ClassLoader?

We know that Java Program runs on Java Virtual Machine (JVM). When we compile a Java Class, it transforms it in the form of bytecode that is platform and machine independent compiled program and store it as a .class file. After that when we try to use a Class, Java ClassLoader loads that class into memory.

There are three types of built-in Class Loaders in Java:

1. **Bootstrap Class Loader** – It loads JDK internal classes, typically 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** – It loads classes from the current classpath that can be set while invoking a program using -cp or -classpath command line options.

Java Class Loaders are hierarchical and whenever a request is raised to load a class, it delegates it to its parent and in this way 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.

Lets 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 of the above program:

```
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
```

As you can see that java.util.HashMap ClassLoader is coming as null that reflects Bootstrap ClassLoader whereas DNSNameService 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, which in turns delegates it to Bootstrap ClassLoader that found the class and load it in JVM. The same process is followed for DNSNameService class but Bootstrap ClassLoader is not able to locate it since its in $JAVA_HOME/lib/ext/dnsns.jar and hence gets loaded by Extensions Class Loader. Note that Blob class is included in the MySql JDBC Connector jar (mysql-connector-java-5.0.7-bin.jar) that I have included in the build path of the project before executing it and its also getting loaded by System Class Loader.

One more important point to note is that 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 ClassLoader?

Java default ClassLoader can load files from local file system that is good enough for most of the cases. But if you are expecting a class at the runtime or from 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 remote web server.

How does ClassLoader Work?

When JVM requests for a class, it invokes loadClass function of the ClassLoader by passing the fully classified name of the Class.

loadClass function calls for findLoadedClass() method to check that the class has been already loaded or not. It's required to avoid loading the class multiple times.

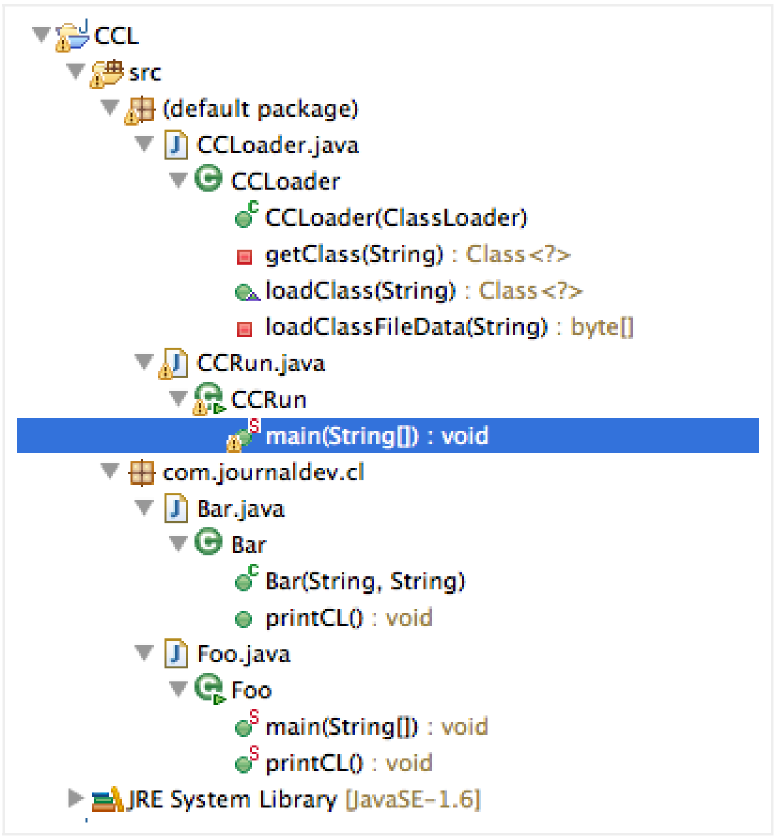
If the Class is not already loaded then it will delegate the request to parent ClassLoader to load the class.

If the parent ClassLoader is not finding the Class then it will invoke findClass() method to look for the classes in the file system.

Creating our own ClassLoader

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

The project structure will be like the below image:



```
CCL
  src
    (default package)
      CCLoader.java
        CCLoader
          CCLoader(ClassLoader)
          getClass(String) : Class<?>
          loadClass(String) : Class<?>
          loadClassFileData(String) : byte[]
      CCRun.java
        CCRun
          main(String[]) : void
    com.journaldev.cl
      Bar.java
        Bar
          Bar(String, String)
          printCL() : void
      Foo.java
        Foo
          main(String[]) : void
          printCL() : void
  JRE System Library [JavaSE-1.6]
```

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.

CCLoader Source Code

```
import java.io.DataInputStream;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;

/**
 * Our Custom Class Loader 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 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 its not in Classpath to avoid
     * any confusion.
     *
     * @param name
     *            File name
     * @return Byte array read from the file
     * @throws IOException
     *             if any 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;
    }
}
```

CCRun.java:

This is our test class with main function where we are creating object of our ClassLoader and load sample classes using its loadClass method. After loading the Class, we are using Java Reflection API 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]);
    }

}
```

Foo.java and Bar.java:

These are our test classes that is getting loaded by our custom classloader. They also have a printCL() method that is getting invoked to print the ClassLoader that has loaded the Class. Foo class will be loaded by our custom class loader which in turn 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());
    }
}
```

Execution Steps

First of all we will compile all the classes through command line. After that we will run 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 with output will be like below.

```
Pankaj$ javac -cp . com/journaldev/cl/Foo.java
Pankaj$ javac -cp . com/journaldev/cl/Bar.java
Pankaj$ javac CCLoader.java
Pankaj$ javac CCRun.java
CCRun.java:18: warning: non-varargs call of varargs method with inexact argument type for last paramete
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
Pankaj$ 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
ctk-pcs1313512-2:src pk93229$
```

If you look into the output carefully, first its trying to load com.journaldev.cl.Foo class but since its extending java.lang.Object class, its trying to load it first and the request it coming to CCLoader loadClass method that 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 that is getting clear when we invoke their printCL() function.

Note that 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 that just taking it from file system.

Updated from comment by m29

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 java class loader option.

```
Pankaj$ javac -cp .:../lib/mysql-connector-java-5.0.7-bin.jar com/journaldev/classloader/ClassLoaderTes
Pankaj$ java -cp .:../lib/mysql-connector-java-5.0.7-bin.jar -Djava.system.class.loader=CCLoader com.jo
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
Pankaj$
```

As you can see that CCLoader is loading the ClassLoaderTest class because its in com.journaldev package.

Mac OS X 10.6.4 Issue with ClassLoader Java Options

If you are working on Mac OS the above execution can throw some exceptions but it will execute successfully.

```
Pankaj$$ java -cp .:../lib/mysql-connector-java-5.0.7-bin.jar -Djava.system.class.loader=CCLoader com.j
Intentionally suppressing recursive invocation exception!
java.lang.IllegalStateException: recursive invocation
    at java.lang.ClassLoader.initSystemClassLoader(ClassLoader.java:1391)
    at java.lang.ClassLoader.getSystemClassLoader(ClassLoader.java:1374)
    at sun.security.jca.ProviderConfig$1.run(ProviderConfig.java:64)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.security.jca.ProviderConfig.getLock(ProviderConfig.java:62)
    at sun.security.jca.ProviderConfig.getProvider(ProviderConfig.java:187)
    at sun.security.jca.ProviderList.getProvider(ProviderList.java:215)
    at sun.security.jca.ProviderList.getService(ProviderList.java:313)
    at sun.security.jca.GetInstance.getInstance(GetInstance.java:140)
    at java.security.cert.CertificateFactory.getInstance(CertificateFactory.java:148)
    at sun.security.pkcs.PKCS7.parseSignedData(PKCS7.java:244)
    at sun.security.pkcs.PKCS7.parse(PKCS7.java:141)
    at sun.security.pkcs.PKCS7.parse(PKCS7.java:110)
    at sun.security.pkcs.PKCS7.<init>(PKCS7.java:92)
    at sun.security.util.SignatureFileVerifier.<init>(SignatureFileVerifier.java:80)
    at java.util.jar.JarVerifier.processEntry(JarVerifier.java:256)
    at java.util.jar.JarVerifier.update(JarVerifier.java:188)
    at java.util.jar.JarFile.initializeVerifier(JarFile.java:321)
    at java.util.jar.JarFile.getInputStream(JarFile.java:386)
    at sun.misc.JarIndex.getJarIndex(JarIndex.java:99)
    at sun.misc.URLClassPath$JarLoader$1.run(URLClassPath.java:606)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.misc.URLClassPath$JarLoader.ensureOpen(URLClassPath.java:597)
    at sun.misc.URLClassPath$JarLoader.<init>(URLClassPath.java:581)
    at sun.misc.URLClassPath$3.run(URLClassPath.java:331)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.misc.URLClassPath.getLoader(URLClassPath.java:320)
    at sun.misc.URLClassPath.getLoader(URLClassPath.java:297)
    at sun.misc.URLClassPath.getResource(URLClassPath.java:167)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:192)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:244)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:319)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:309)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:330)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:254)
    at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:399)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:247)
    at java.lang.SystemClassLoaderAction.run(ClassLoader.java:2147)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.lang.ClassLoader.initSystemClassLoader(ClassLoader.java:1404)
    at java.lang.ClassLoader.getSystemClassLoader(ClassLoader.java:1374)
Intentionally suppressing recursive invocation exception!
java.lang.IllegalStateException: recursive invocation
    at java.lang.ClassLoader.initSystemClassLoader(ClassLoader.java:1391)
    at java.lang.ClassLoader.getSystemClassLoader(ClassLoader.java:1374)
    at sun.security.jca.ProviderConfig$3.run(ProviderConfig.java:231)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.security.jca.ProviderConfig.doLoadProvider(ProviderConfig.java:225)
    at sun.security.jca.ProviderConfig.getProvider(ProviderConfig.java:205)
    at sun.security.jca.ProviderList.getProvider(ProviderList.java:215)
    at sun.security.jca.ProviderList.getService(ProviderList.java:313)
    at sun.security.jca.GetInstance.getInstance(GetInstance.java:140)
    at java.security.cert.CertificateFactory.getInstance(CertificateFactory.java:148)
    at sun.security.pkcs.PKCS7.parseSignedData(PKCS7.java:244)
    at sun.security.pkcs.PKCS7.parse(PKCS7.java:141)
    at sun.security.pkcs.PKCS7.parse(PKCS7.java:110)
    at sun.security.pkcs.PKCS7.<init>(PKCS7.java:92)
    at sun.security.util.SignatureFileVerifier.<init>(SignatureFileVerifier.java:80)
    at java.util.jar.JarVerifier.processEntry(JarVerifier.java:256)
    at java.util.jar.JarVerifier.update(JarVerifier.java:188)
    at java.util.jar.JarFile.initializeVerifier(JarFile.java:321)
    at java.util.jar.JarFile.getInputStream(JarFile.java:386)
    at sun.misc.JarIndex.getJarIndex(JarIndex.java:99)
    at sun.misc.URLClassPath$JarLoader$1.run(URLClassPath.java:606)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.misc.URLClassPath$JarLoader.ensureOpen(URLClassPath.java:597)
    at sun.misc.URLClassPath$JarLoader.<init>(URLClassPath.java:581)
    at sun.misc.URLClassPath$3.run(URLClassPath.java:331)
    at java.security.AccessController.doPrivileged(Native Method)
    at sun.misc.URLClassPath.getLoader(URLClassPath.java:320)
    at sun.misc.URLClassPath.getLoader(URLClassPath.java:297)
    at sun.misc.URLClassPath.getResource(URLClassPath.java:167)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:192)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at sun.misc.Launcher$ExtClassLoader.findClass(Launcher.java:244)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:319)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:309)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:330)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:254)
    at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:399)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:247)
    at java.lang.SystemClassLoaderAction.run(ClassLoader.java:2147)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.lang.ClassLoader.initSystemClassLoader(ClassLoader.java:1404)
    at java.lang.ClassLoader.getSystemClassLoader(ClassLoader.java:1374)
Loading Class 'com.journaldev.classloader.ClassLoaderTest'
Loading Class using CCLoader
Loading Class 'java.lang.Object'
Loading Class 'java.lang.String'
Loading Class 'java.lang.System'
Loading Class 'java.lang.StringBuilder'
Loading Class 'java.util.HashMap'
Loading Class 'java.lang.Class'
Loading Class 'java.io.PrintStream'
class loader for HashMap: null
Loading Class 'sun.net.spi.nameservice.dns.DNSNameService'
class loader for DNSNameService: sun.misc.Launcher$ExtClassLoader@24480457
class loader for this class: CCLoader@38503429
Loading Class 'com.mysql.jdbc.Blob'
sun.misc.Launcher$AppClassLoader@2f94ca6c
Pankaj$
```

Refer this email archive where Tim Quinn is also facing the same issue.