**Jar Hell made Easy - Demystifying the classpath with jHades**

Some of the hardest problems a Java Developer will ever have to face are classpath errors: ClassNotFoundException, NoClassDefFoundError,   
Jar Hell, [Xerces Hell](http://stackoverflow.com/questions/11677572/dealing-with-xerces-hell-in-java-maven) and company.

In this post we will go through the root causes of these problems, and see how a minimal tool ([JHades](http://jhades.org/)) can help solving them quickly. We will see why Maven cannot (always) prevent classpath duplicates, and also:

* The only way to deal with Jar Hell
* Class loaders
* The Class loader chain
* Class loader priority: Parent First vs Parent Last
* Debugging server startup problems
* Making sense of Jar Hell with jHades
* Simple strategy for avoiding classpath problems
* The classpath gets fixed in Java 9?

**The only way to deal with Jar Hell**

Classpath problems can be time-consuming to debug, and tend to happen at the worst possible times and places: before releases, and often in environments where there is little to no access by the development team.

They can also happen at the IDE level, and become a source of reduced productivity. We developers tend to find these problems early and often, and this is the usual response:



Let's try to save us some hair and get to the bottom of this. These type of problems are hard to approach via trial and error. The only real way to solve them is to *really understand what is going on*, but where to start?

It turns out that Jar Hell problems are simpler than what they look, and only a few concepts are needed to solve them. In the end, the common root causes for Jar Hell problems are:

* a Jar is missing
* there is one Jar too many
* a class is not visible where it should be

But if it's that simple, then why are classpath problems so hard to debug?

**Jar Hell stack traces are incomplete**

One reason is that the stack traces for classpath problems have a lot of information missing that is needed to troubleshoot the problem. Take for example this stack trace:

[?](http://blog.jhades.org/classnotfoundexception-jhades-jar-hell-made-easy/)

|  |  |
| --- | --- |
| 1  2  3  4 | java.lang.IncompatibleClassChangeError:  Class org.jhades.SomeServiceImpl does not implement  the requested interface org.jhades.SomeService      org.jhades.TestServlet.doGet(TestServlet.java:19) |

It says that a class does not implement a certain interface. But if we look at the class source:

[?](http://blog.jhades.org/classnotfoundexception-jhades-jar-hell-made-easy/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class SomeServiceImpl implements SomeService {      @Override      public void doSomething() {          System.out.println( "Call successful!" );      }  } |

Well, the class clearly implements the missing interface! So what is going on then? The problem is that *the stack trace is missing a lot of information* that is critical to understanding the problem.

The stack trace should have probably contained an error message such as this (we will learn what this means):

*The Class SomeServiceImpl of class loader /path/to/tomcat/lib does not implement the interface SomeService loaded from class loader Tomcat - WebApp - /path/to/tomcat/webapps/test*

This would be at least an indication of where to start:

* Someone new learning Java would at least know that there is this notion of class loader that is essential to understand what is going on
* It would make clear that one class involved was **not** being loaded from a WAR, but somehow from some directory on the server (SomeServiceImpl).

**What is a Class Loader?**

To start, a Class Loader is just a Java class, more exactly an instance of a class at runtime. It is **NOT** an inaccessible internal component of the JVM like for example the garbage collector.

Take for example the WebAppClassLoader of Tomcat, here is it's [javadoc](https://tomcat.apache.org/tomcat-7.0-doc/api/org/apache/catalina/loader/WebappClassLoader.html). As you can see it's just a plain Java class, we can even write our own class loader if needed.

Any subclass of ClassLoader will qualify as a class loader. The main responsibilities of a class loader is to known where class files are located, and then load classes on JVM demand.

**Everything is linked to a class loader**

Each object in the JVM is linked to it's Class via getClass(), and each class is linked to a class loader via getClassLoader(). This means that:

*Every object in the JVM is linked to a class loader!*

Let's see how this fact can be used to troubleshoot a classpath error scenario.

**How-To find where a class file really is**

Let's take an object and see where it's class file is located in the file system:

[?](http://blog.jhades.org/classnotfoundexception-jhades-jar-hell-made-easy/)

|  |  |
| --- | --- |
| 1  2  3 | System.out.println(service.getClass()      .getClassLoader()      .getResource("org/jhades/SomeServiceImpl.class")); |

This is the full path to the class file:

jar:file:/Users/user1/.m2/repository/org/jhades/jar-2/1.0-SNAPSHOT/jar-2-1.0-SNAPSHOT.jar!/org/jhades/SomeServiceImpl.class

As we can see the class loader is just a runtime component that knowns where in the file system to look for class files and how to load them.

But what happens if the class loader cannot find a given class?

**The Class loader Chain**

By default in the JVM, if a class loader does not find a class, it will then ask it's parent class loader for that same class and so forth.

This continues all the way up until the JVM bootstrap class loader (more on this later). This chain of class loaders is *the class loader delegation chain*.

**Class loader priority: Parent First vs Parent Last**

Some class loaders delegate requests immediately to the parent class loader, without searching first in their own known set of directories for the class file. A class loader operating on this mode is said to be in *Parent First* mode.

If a class loader first looks for a class locally and only after queries the parent if the class is not found, then that class loader is said to be working in *Parent Last* mode.

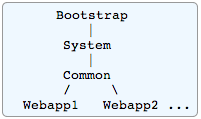
**Do all applications have a class loader chain ?**

Even the most simple Hello World main method has 3 class loaders:

* The Application class loader, responsible for loading the application classes (parent first)
* The Extensions class loader, that loads jars from $JAVA\_HOME/jre/lib/ext (parent first)
* The Bootstrap class loader, that loads any class shipped with the JDK such as java.lang.String (no parent class loader)

**What does the class loader chain of a WAR application look like?**

In the case of application servers like Tomcat or Websphere, the class loader chain is configured differently than a simple Hello World main method program. Take for example the case of the Tomcat class loader chain:



Here we wee that each WAR runs in a WebAppClassLoader, that works in parent last mode (it can be set to parent first as well). The Common class loader loads libraries installed at the level of the server.

**What does the Servlet spec say about class loading?**

Only a small part of the class loader chain behavior is defined by the Servlet container specification:

* The WAR application runs on it's own application class loader, that might be shared with other applications or not
* The files in WEB-INF/classes take precedence over everything else

After that, it's anyones guess! The rest is completely open for interpretation by container providers.

**Why isn't there a common approach for class loading across vendors?**

Usually open source containers like Tomcat or Jetty are configured by default to look for classes in the WAR first, and only then search in server class loaders.

This allows for applications to use their own versions of libraries that override the ones available on the server.

**What about the big iron servers?**

Commercial products like Websphere will try to 'sell' you their own server provided libraries, that by default take precedence over the ones installed on the WAR.

This is done assuming that if you bought the server you want also to use the JEE libraries and versions it provides, which is often **NOT** the case.

This makes deploying to certain commercial products a huge hassle, as they behave differently then the Tomcat or Jetty that developers use to run applications in their workstation. We will see further on a solution for this.

**Common Problem: duplicate class versions**

At this moment you probably have a huge question:

*What if there are two jars inside a WAR that contain the exact same class?*

The answer is that the behavior is undetermined and **only at runtime one of the two classes will be chosen**. Which one gets chosen depends on the internal implementation of the class loader, there is no way to know upfront.

But luckily most projects these days use Maven, and Maven solves this problem by ensuring only one version of a given jar is added to the WAR.

*So a Maven project is immune to this particular type of Jar Hell, right?*

**Why Maven does not prevent classpath duplicates**

Unfortunately Maven cannot help in all Jar Hell situations. In fact, many Maven projects that don't use certain quality control plugins can have*hundreds* of duplicate class files on the classpath (I saw trunks with over 500 duplicates). There are several reasons for that:

* **Library publishers occasionally change the artifact name of a jar:** This happens due to re-branding or other reasons. Take for example the example of the [JAXB jar](http://search.maven.org/#search%7Cga%7C1%7Cjaxb). There is no way Maven can identify those artifacts as being the same jar!
* **Some jars are published with and without dependencies:** Some library providers provide a 'with dependencies' version of a jar, which includes other jars inside. If we have transitive dependencies with the two versions, we will end up with duplicates.
* **Some classes are copied between jars:** Some library creators, when faced with the need for a certain class will just grab it from another project and copy it to a new jar without changing the package name.

**Are all class files duplicates dangerous?**

If the duplicate class files exist inside the same class loader, and the two duplicate class files are exactly identical then it does not matter which one gets chosen first - this situation is not dangerous.

If the two class files are inside the same class loader and they are not identical, then there is no way which one will be chosen at runtime - this is problematic and can manifest itself when deploying to different environments.

If the class files are in two different class loaders, then they are never considered identical (see the class identity crisis section further on).

**How can WAR classpath duplicates be avoided?**

This problem can be avoided for example by using the Maven Enforcer Plugin, with the extra rule of [Ban Duplicate Classes](http://mojo.codehaus.org/extra-enforcer-rules/banDuplicateClasses.html) turned on.

You can quickly check if your WAR is clean using the [JHades WAR duplicate classes report](http://jhades.org/documentation.html" \l "warreport) as well. This tool has an option to filter 'harmless' duplicates (same class file size).

But even a clean WAR might have deployment problems: Classes missing, classes taken from the server instead of the WAR and thus with the wrong version, class cast exceptions, etc.

**Debugging the classpath with JHades**

Classpath problems often show up when the application server is starting up, which is a particularly bad moment specially when deploying to an environment where there is limited access.

JHades is a tool to help deal it with Jar Hell (disclaimer: I wrote it). It's a single Jar with no dependencies other than the JDK7 itself. This is an example of how to use it:

[?](http://blog.jhades.org/classnotfoundexception-jhades-jar-hell-made-easy/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | new JHades()     .printClassLoaders()     .printClasspath()     .overlappingJarsReport()     .multipleClassVersionsReport()     .findClassByName("org.jhades.SomeServiceImpl") |

This prints to the screen the class loader chain, jars, duplicate classes, etc.

**Debugging server startup problems**

JHades works works well in scenarios where the server does not start properly. A [servlet listener](http://jhades.org/documentation.html#troubleshooting) is provided that allows to print classpath debugging information even before any other component of the application starts running.

**ClassCastException and the Class Identity Crisis**

When troubleshooting Jar Hell, beware of ClassCastExceptions. A class is identified in the JVM not only by it's fully qualified class name, but **also**by it's class loader.

This is counterintuitive but in hindsight makes sense: We can create two different classes with the same package and name, ship them in two jars and put them in two different class loaders. One let's say extends   
ArrayList and the other is a Map.

The classes are therefore completely different (despite the same name) and cannot be cast to each other! The runtime will throw a CCE to prevent this potential error case, because there is no guarantee that the classes are castable.

Adding the class loader to the class identifier was the outcome of the[Class Identity Crisis](http://www.ibm.com/developerworks/library/j-dyn0429/) that occurred in earlier Java days.

**A Strategy for Avoiding Classpath Problems**

This is easier said then done, but the best way to avoid classpath related deployment problems is to run the production server in P*arent Last*mode.

This way the class versions of the WAR take precedence over the ones on the server, and the same classes are used in production and in a developer workstation where it's likely that Tomcat, Jetty or other open source *Parent Last* server is being used.

In certain servers like Websphere, this is not sufficient and you also have to provide special properties on the manifest file to explicitly turn off certain libraries like for example JAX-WS.

**Fixing the classpath in Java 9**

In Java 9 the classpath gets completely revamped with the new [Jigsaw](http://openjdk.java.net/projects/jigsaw/)modularity system. In Java 9 a jar can be declared as a module and it will run in it's own isolated class loader, that reads class files from other similar module class loaders in an OSGI sort of way.

This will allow multiple versions of the same Jar to coexist in the same application if needed.

**Conclusions**

In the end, Jar Hell problems are not that low level or unapproachable as they might seem at first. It's all about zip files (jars) being present/ not being present in certain directories, how to find those directories, and how to debug the classpath in environments with limited access.

By knowing a limited set of concepts such as Class Loaders, the Class Loader Chain and Parent First / Parent Last modes, these problems can be tackled effectively.

**External links**

This presentation [Do you really get class loaders](http://parleys.com/play/514892260364bc17fc56be0e/chapter1/about) from Jevgeni Kabanov of ZeroTurnaround ([JRebel](http://zeroturnaround.com/software/jrebel/) company) is a great resource about Jar Hell and the different type of classpath related exceptions.