

Classloading in Oracle9iAS Containers for J2EE

*An Oracle White Paper
January 2003*

Classloading in Oracle9iAS Containers for J2EE

A Familiar Problem.....	3
ClassLoading fundamentals	3
Classloader Namespaces	4
Linking ClassLoaders	5
Searching for Classes.....	6
ClassLoading in a J2SE Environment	9
Other classloader considerations	10
Dependency Declarations.....	10
Dependency Resolution Issues	10
Security	11
ClassLoading in J2EE.....	11
Inter-Module Dependencies.....	12
Cross Application Dependencies	13
Search Order within Web modules	14
Classloading in Oracle9iAS Containers for J2EE (OC4J).....	14
Cross Application Sharing.....	17
Configuration Option Summary.....	18
Common classloading problems in J2EE.....	19
Not Enough Visibility	20
Too Much Visibility.....	20
Mangled Visibility	21
J2EE Classloading Best Practices	22
Conclusion.....	23

Classloading in Oracle9iAS Containers for J2EE

A FAMILIAR PROBLEM

ClassNotFoundException.

NoClassDefFoundError.

ClassCastException.

Sound familiar? If you've ever spent frustrating hours trying to debug any of these exceptions, you're not alone. With the changes introduced in the Java2 and J2EE specifications, class loading is no longer as simple as defining a CLASSPATH environment variable. J2EE application servers such as Oracle9iAS Containers for J2EE (OC4J provided with Oracle9iAS) supplement the CLASSPATH with many different mechanisms to specify where classes live. This additional complexity often leads to classloading errors – errors that can be frustrating and difficult to diagnose.

In this whitepaper, we will dig under the covers to explain how classloading works, both in a standard J2SE environment, and in more complex J2EE environments such as Oracle's OC4J. By understanding some core class loading concepts, you should be able to diagnose problems quickly and resolve those nasty class-loading errors.

CLASSLOADING FUNDAMENTALS

Let's start with the basics. The term “class loading” refers to the process of locating the bytes for a given class name and converting them into a Java *Class* instance. All *java.lang.Class* instances within a Java Virtual Machine (JVM) start life as an array of bytes, structured in the class file format defined by the JVM specification.

Class loading is performed by the JVM during the startup process, and subsequently by subclasses of the *java.lang.ClassLoader* class. These “classloaders” provide an abstraction that provides one of the most powerful features of Java: the ability to choose, load, and execute code from a class that is unknown at compile-time. This “dynamic-loading” feature provides a great deal of extensibility to the Java language and is the basis for Java's much touted “mobile code” capabilities. Using the `Class.forName()` method, for example, developers can create a *java.lang.Class* instance simply by specifying a fully-qualified class name string.

Simply put, a classloader is a subclass of the `java.lang.ClassLoader` class that is responsible for loading classes. Each classloader, in turn, is designed to work with one or more *codesources*. A codesource is a root location from which the JVM searches for classes. Codesources can be defined to represent physical storage of binary class files, java sources that must first be compiled, or even classes generated on the fly.

As an example, most developers are familiar with the CLASSPATH environment variable. In this case, each element in the CLASSPATH is a codesource, whether it is a directory, a zip file, or a jar file. A classloader uses each codesource, along with a class package name, to define a location to search for classes. For example, given a directory code-source such as “c:\classes” and a class name of “com.acme.Dynamite”, a classloader can build a full path through a simple transformation to “c:\classes\com\acme\Dynamite.class”.

In a Java application, there are a number of different classloaders that use any number of different mechanisms to load classes. In addition to the CLASSPATH example above, classloaders can be designed such that classes are retrieved:

- From a database. In this case configuration could consist of all the data needed to point at the correct table(s) in a specific database
- From a remote server running a proprietary communications protocol. Configuration could consist of DNS names, ports, and other network information
- From the file system, but with a special search order specified in a properties file
- From sources defined within an XML file
- From source code (*.java) files that must be compiled

ClassLoader Namespaces

Note that each classloader (or instance of a *ClassLoader* subclass) defines a unique and separate namespace. For a given class with name N, only one instance of class N may exist within a given classloader. But the same class can be loaded by two different classloaders. In this case, the JVM will consider each class to be further qualified by the ClassLoader instance that loaded it. In effect, the full name of a class consists of its fully qualified class name *plus* its classloader.

The loading of duplicate classes can be the source of some subtle and frustrating bugs. For example, if two different classloaders load `com.acme.Dynamite`, there will be two *Class* instances, each with its own separate static data. Any attempt to cast an object from one to the other will result in a *ClassCastException*³. As an example,

³ When testing for type equivalence, the VM walks the source inheritance hierarchy comparing each Class instance to the desired type’s Class instance, using an == (identity) comparison.

consider an application with both an EJB and a Web module, and where the EJBObject interface (“Cart”) from the EJB module is duplicated in the Web module war file. When a servlet from the Web module uses JNDI to lookup the EJB, it gets a ClassCastException when it casts to the Cart interface.

This exception occurs because, unlike all other loaders, the Web module loader must look locally first. In this case, it gets the locally packaged Cart interface, and the EJB returned by JNDI has the Cart interface from the EJB loader. (The reason for this behavior is discussed later in this article in the section “Search order Within Web Modules”).

Linking ClassLoaders

Classloaders are linked together in such a way so that, for any attempt to locate a class, a specific sequence of classloaders is used to perform the search. Each loader has an associated parent classloader. Using this relationship, a classloader hierarchy can represent a tree structure, ranging in complexity from simple chains to complex multi-branched trees. An important feature of these trees is that they can only be searched one way: upward, through the parent.

The result of this hierarchy is a nested namespace in which class visibility is governed by a classloader’s placement in the tree: nodes farther “down” the tree can see the contents of each of their parents, but parents cannot see the contents of their child nodes.

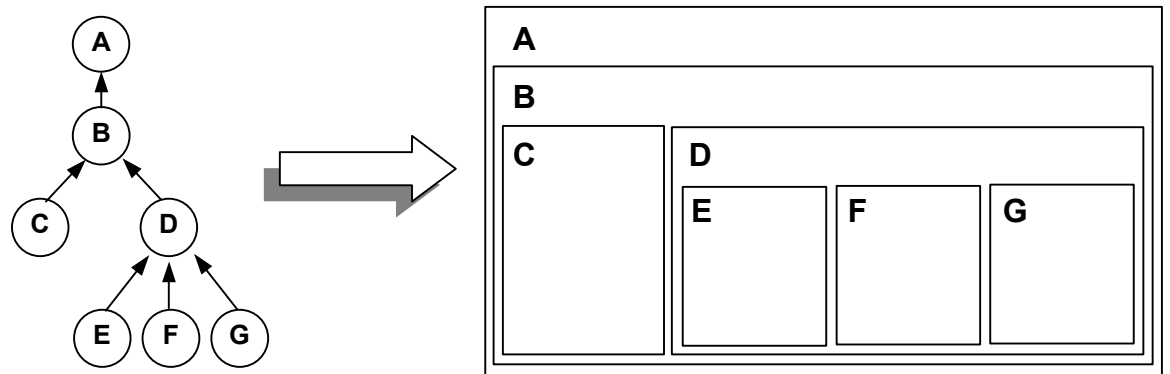


Figure 1: Class Visibility within a ClassLoader tree.

This relationship is shown in Figure 1. Each namespace (box) on the right can “see” only itself and the contents of all the namespaces that *contain* it. Classes that live near the top of the tree have “higher” visibility than those lower down, since more classes can see them.

This partitioning scheme can cause problems at times due to its restrictive nature. For example, consider a framework that dynamically loads implementation code (using the `Class.forName()` method), where the implementation is expected to be added by a third-party. It is very common for the third-party code to reside “lower”

in the tree than the framework itself. In this case, how does the framework extend its reach and gain visibility to this code?

The answer is that the framework must explicitly find a classloader that has the correct visibility, and use it to load the implementation. Java2 provides one simple, but important, mechanism to deal with this case. Each Java Thread can store a single `ClassLoader` instance, known as the “context” `ClassLoader`, that can be retrieved by calling the method `Thread.currentThread().getContextClassLoader()`. This instance can be set by the environment to provide any desired visibility. (By default, the context loader is set to the result of calling `ClassLoader.getSystemClassLoader()`).

Searching for Classes

Now that we’ve discussed relationships among classloaders, let’s discuss how a JVM actually performs searches. Classloaders use a simple delegation model to search for classes. Each `ClassLoader` instance has an associated parent class as well as a cache for previously loaded classes. When called to load a class, a classloader first looks in its cache, and, if no class is found, it will delegate to its parent. If the parent cannot return the class, the loader will attempt to find the class itself. This is a recursive process that goes to the top level in the classloader tree.

The first step in the search process is to find an *initial classloader*. The initial classloader is the first loader used when performing the search described above. Note that the initial loader is not necessarily the actual loader (e.g. a parent loader may return the class first). The selection of the initial loader is a critical first step, and may be either *implicit* or *explicit*, as shown below:

Implicit selection is by far the most common, and occurs in the following cases:

- *Object instantiation*. When class A invokes the *new* operator to create an instance of class B, the VM selects A’s classloader as the initial classloader.
- *Dependency resolution*. If class A depends on B, then the VM will select A’s classloader when it needs to load B. This process is recursive, so that if B depends on C, the VM will select B’s classloader to load C. Note that different classloaders may be involved for each class:

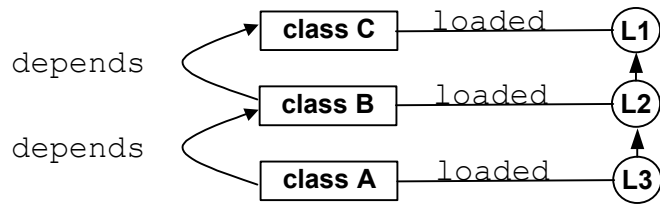


Figure 2: Selection of initial classloaders

- *Dynamic loading.* The `Class.forName()` method is overloaded; one version takes a `ClassLoader` parameter and one does not. When no `ClassLoader` is passed, the JVM searches the call stack for the first object with a non-null `ClassLoader` (i.e. not the “bootstrap” loader – see the next section for details). If none is found, `ClassLoader.getSystemClassLoader()` is used.
- *Object deserialization.* The `ObjectInputStream.resolveClass()` method walks the call stack similarly to `Class.forName()`.

Explicit selection, although less common, occurs when application code uses the result from a call to:

- `getClassLoader()` on a `Class` instance.
- `ClassLoader.getSystemClassLoader()`.
- `Thread.currentThread().getContextClassLoader()`.
- An application-specific method to retrieve a class loader.

In either case, the initial loader selection is critical because it defines visibility. For example, if the target class or one of its dependencies is visible only to a loader lower on the chain, or in a different branch, an exception will be the inevitable result.

The default implementation of `loadClass()` encodes the following standard search algorithm (see Figure 3 for reference)⁴:

⁴ While this method can be overridden and the algorithm changed, in the Java2 world this is frowned upon because it can lead to unexpected and difficult to diagnose behavior.

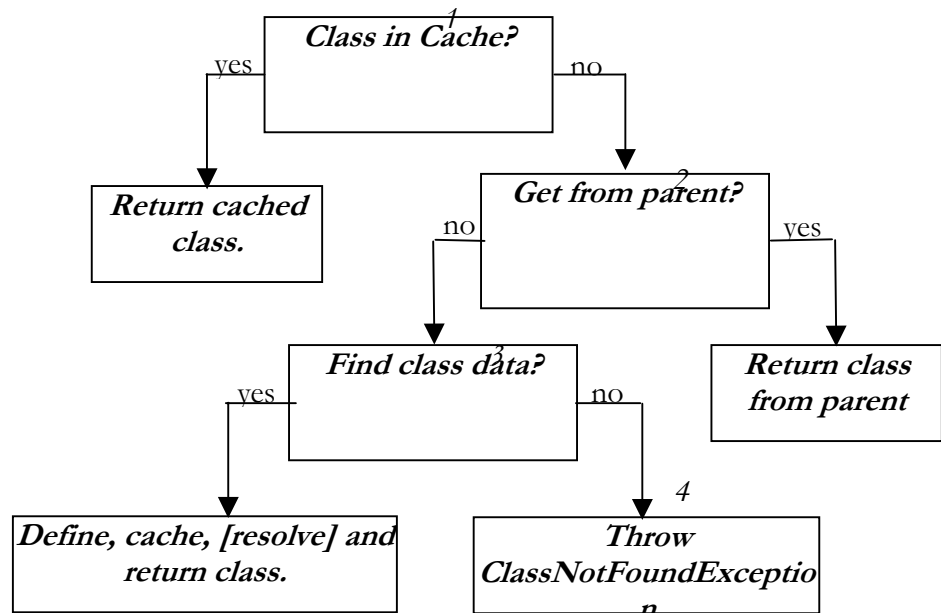


Figure 3: Class loading default algorithm

1. Call `findLoadedClass(name)` to see if the class has been previously loaded and cached. If not found, go to step 2.
2. If parent is non-null, call `parent.loadClass(name)`. (The parent class will execute this same algorithm described here). If parent is null, call `findBootstrapClass(name)`. In either case, if the class is not found, go to step 3.
3. Call `findClass(name)`. This method must be implemented in every `ClassLoader` subclass, and must locate the class bytes and call `defineClass()`, which converts the bytes into a `Class` instance and updates the cache. If the class is still not found, go to step 4.
4. Throw a `ClassNotFoundException`.

If the class was loaded in step 2 or 3, there is a final, optional call to `resolveClass()` which will recursively complete the linking process. This is controlled via the setting of the ‘`resolveClass`’ parameter to `loadClass()`. By default, this parameter is false, thereby enabling a feature called “lazy” loading (see “Dependency Resolution Issues” later in this article).

An important implication of this search order is that if a class is duplicated across classloaders in the same chain, the one with the higher visibility (i.e. higher in the tree) will always be selected. This means that, in general, it is not possible to replace or override a class higher in the chain. In most cases, this is a good thing. Imagine the havoc that could result from introducing a duplicate, but separate, `java.lang.Object` class that is used as the superclass of only *some* objects.

Unless otherwise specified, the search order *within* a `ClassLoader` is defined by the order in which the code-sources are specified.

ClassLoader in a J2SE Environment

When a JVM is started (in all versions since JDK 1.2), it forms an initial classloader hierarchy composed of three loaders:

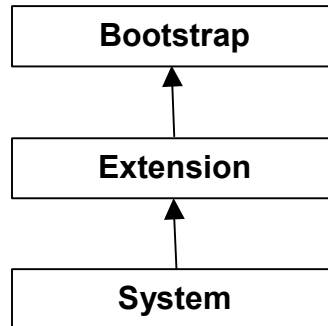


Figure 4: Standard J2SE ClassLoader Chain

The *bootstrap* (also known as “primordial”) loader is responsible for loading the core Java classes (*rt.jar*, *I18n.jar*, etc). In Sun’s JVM, the *-Xbootclasspath* option or the *sun.boot.class.path* system property can be used to specify additional classes. This “loader” is unique in that it is not actually a subclass of *java.lang.ClassLoader* but is implemented by the JVM itself.

The *extension* loader is responsible for loading classes from the jars in the JRE’s extension directory (*jre/lib/ext* or as specified by the *java.ext.dirs* system property). This provides a standard mechanism to introduce new functionality beyond the core Java classes. Calling *getParent()* on this instance will always return *null*, since the bootstrap loader is not an actual *ClassLoader* instance. Note that since the default extension directory is common to all JVM’s launched from the same JRE installation, jars put in this directory are visible across *processes*.

The *system* (a.k.a “application”) loader is responsible for loading classes from the directories and jars listed on the command-line and/or the *java.class.path* system property when the JVM is invoked. This loader can always be found via the static method *ClassLoader.getSystemClassLoader()*. If not otherwise specified, any user-instantiated *ClassLoader* will have this loader as its parent.

JDK 1.4 adds a mechanism to override what are called “endorsed standards” – those standards that are not defined by the Java Community Process but are shipped with the JRE, such as CORBA¹. This feature enables new versions of such standards to replace those shipped with the JRE. The machinery that supports this feature appears to be implemented within the VM/bootstrap loader.

¹ See <http://java.sun.com/j2se/1.4/docs/guide/standards/index.html>.

OTHER CLASSLOADER CONSIDERATIONS

Before we move on to J2EE and Oracle OC4J environments, there are a couple other classloading issues related to standard J2SE environments you should be aware of:

Dependency Declarations

Prior to JDK 1.2, there was no mechanism to express the dependencies between classes in one jar file and those in another. The user was expected to know which jars to include and list them in the class path – often causing errors.

Consider an example where an application needs to use classes from a third-party library that creates graphs (“graphs.jar”). Furthermore, that library has a dependency on a low-level image format code in a different library (“compuserve-gif.jar”). In this example, the code might compile correctly with just “graphs.jar” in the classpath, but fail at runtime because “compuserve-gif.jar” is not in the classpath. If this dependency isn’t explicitly documented by the third-party, developers will probably resort to trial-and-error to add the other jar to the classpath.

Java2 added an important new generic configuration mechanism that allows jar files to *declare* their dependencies on other jars. This mechanism is referred to as “*bundled optional packages*”². A new manifest attribute was added, “Class-Path”, whose value is a space-separated list of directory and/or jar file paths – paths that must be relative to the enclosing jar’s location. ClassLoaders that accept jar files are expected to support this feature.

One common and important use of this feature is in enabling application execution using the `-jar <jarfile>` VM option. The manifest attribute allows the jar to define its own class path, eliminating the need for the user to declare it via the `-cp` or `-classpath` options.

JDK 1.3 introduced another mechanism, primarily intended for Applets, which allows a jar to declare dependencies on specific versions of jars – referred to as “*installed optional packages*”. These are extensions to the base JDK functionality, usually installed in the `jre/lib/ext` directory. For Applets executing in the Java Plug-in environment, this mechanism includes the ability to provide a URL from which the jar can be fetched and installed if not already present.

Dependency Resolution Issues

Classloading can be initiated both by the JVM itself (e.g. object instantiation using the “new” operator) or explicitly by application code (e.g. calling `Class.forName()`). One specific case, however, deserves extra attention: dependency resolution.

² This name comes from a later version of the mechanism. See <http://java.sun.com/products/jdk/1.2/docs/guide/extensions/index.html>, or the more recent versions by changing the “1.2” to “1.3” or “1.4”.

Classes contain references to other classes on which they depend, some of which may not yet be loaded. The process of resolving these references (known as “linking”) occurs during class loading, although some references are deferred until first use during method execution. This “lazy” loading improves start-up time, but can lead to errors at unexpected times.

Security

Java2 security is based on the idea that specific permissions can be granted or denied to specific code, where the unit of granularity is the code-source, in the form of a URL. ClassLoaders play a vital role in forming the association between a class instance and its code-source, and therefore have a critical role in security.

While a deeper discussion of this issue is beyond the scope of this article, it is worth mentioning that for any object, the code-source URL can be retrieved by calling:

```
obj.getClass().getProtectionDomain().getCodeSource().getLocation();
```

CLASSLOADING IN J2EE

The J2EE development environment adds complexity that has significant impacts on classloading behavior. Specifically, the J2EE specification defines “applications” in a flexible manner that adds important elements to the classloading picture.

Applications are *component-based, co-resident but separate, and reloadable*. Each of these characteristics is described below:

Component based. Applications are not monolithic; rather they are collections of components (EJBs, Servlets, JSPs, Resource Adapters, etc.) that have pre-defined packaging (using JAR, WAR, and RAR files) and deployment directory structures as well as an umbrella packaging structure (EAR files). Each of these structures has ClassLoader implications.

Within an EAR file, the META-INF/application.xml file contains the relative-paths to each of the contained modules. Each module is or contains pre-defined codesources, depending on the packaging:

- JAR files (EJB and application-client modules). Codesources include:
 1. The jar file itself.
 2. All ClassPath entries in the META-INF/manifest.mf file, if present.
 3. All manifest ClassPath entries in all jars in 2, recursively.
- WAR files (Web modules). Codesources include:
 4. The WEB-INF/classes directory.
 5. All jar files in the WEB-INF/lib directory, if present.

6. All ClassPath entries in the META-INF/manifest.mf file, if present.
 7. All manifest ClassPath entries in all jars in 2 and 3, recursively.
- RAR (Resource-adapter modules). Codesources include:
 8. All jar files (at any level).
 9. All manifest ClassPath entries in all jars in 1, recursively.

Note that any manifest file (and any ClassPath entries) at the root of an EAR file itself is *ignored*.

Application server vendors are free to add enhancements that introduce additional (though non-standard) code-sources. (Oracle OC4J enhancements are described in the next section.)

Co-resident but separate. Many applications may exist side-by-side within an application server, but each application must live in its own separate namespace. In practice, this means that each application must be loaded by (one or more) separate ClassLoader instances.

Reloadable. Applications may be reloaded without stopping the server. For servers that support this feature (such as Oracle OC4J), the ClassLoader(s) used for a given application may be thrown out and re-instantiated³. This behavior inherently causes the “same class, different loader” scenario, which can lead to subtle class loading problems.

Inter-Module Dependencies

Inherent in the design of J2EE is the ability of Servlets and JSPs to call EJBs, and for all three to be able to use resource adapters. Web-modules within an application are required to be isolated from one another. In EJB 1.x, only the client elements needed to be visible (i.e. home/remote interfaces and their dependencies). Now, in EJB 2.x, local interfaces add the requirement that implementation classes are also visible.

J2EE servers must arrange the classloaders for a given application to ensure that each module has the correct visibility to meet the above requirements. A typical classloader structure for an EAR file containing multiple web modules, EJBs and resource-adapters might be:

³ Since JDK1.2, classes and class loaders may be garbage collected, so when all references to the original application classes go away, these are collected.

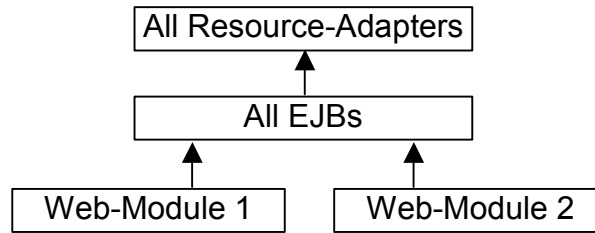


Figure 5: Classloader structure across modules in a J2EE application

Each loader would include all of the codesources defined in the respective packages, as described earlier⁴. The structure above would ensure that classes loaded in each web module are isolated, while providing each module visibility to EJB and resource adapter classes.

Cross Application Dependencies

Generally speaking, dependencies across applications should be minimized, and J2EE 1.3 makes no explicit mechanism available to satisfy them. When it *is* needed, there are two scopes to consider, depending on whether you want the shared code to be visible to:

- *All applications.* To do this, either put the libraries in the JRE extension directory, include the libraries via the `-classpath` argument when launching the server, or use a vendor-specific mechanism. Changing code at this level may require the server to be re-started.
- *Some applications.* Achieving this requires either duplicating the libraries in each application, or using a vendor-specific configuration option. One important consideration here is whether or not an *object* of a shared class must be visible across the applications. If so, duplicating the libraries will not work because the class is duplicated, and, as explained earlier, the VM considers them to be different classes.

The choices for “some applications” visibility are less than ideal. Duplication increases both disk and memory footprint and can lead to version problems. Relying on vendor-specific mechanisms reduces portability. Deploying at the

⁴ Some vendors have chosen to export the code-sources defined in the WAR manifest Class-Path to the top-most loader in the application’s structure, citing an ambiguity in the J2EE 1.3 specifications. OC4J does not do so. This issue should be resolved in the J2EE 1.4 specifications. You can use Manifest Class-Path in WAR files in OC4J 9.0.3 using the following tag in the `orion-web.xml`:

```
<web-app-class-loader include-war-manifest-class-path="true" />
```

container level is always an option, but may result in too much visibility for those libraries and may also limit application reloading.

A better solution (which we will hopefully see in J2EE 1.4) would allow sharing of code across applications that:

1. Is standardized.
2. Eliminates duplication.
3. Is visible only to the desired subset of applications.
4. Properly supports application re-loading.

Search Order within Web modules

The Servlet 2.3 specification requires that the search order within a web-module is WEB-INF/classes/ first, then WEB-INF/lib/. In addition, it introduces an interesting twist to the standard ClassLoader search order:

“It is recommended also that the application class loader be implemented so that classes and resources packaged within the WAR are loaded in preference to classes and resources residing in container-wide library JARs.”

This means that, rather than searching up the parent chain first, a web-application loader should first look locally. This allows classes packaged within the WAR to *override* code installed higher up the loader-chain.

Note that this is only a recommendation, so not all vendors will implement it. Further, it applies only to web-applications, even though the capabilities it provides may be useful to other J2EE components. Perhaps this issue will also be addressed more comprehensively in J2EE 1.4.

Starting with OC4J 9.0.3 you can achieve this functionality on a per web-application basis by having the following tag in the orion-web.xml:

```
<web-app-class-loader search-local-classes-  
first="true"/>
```

CLASSLOADING IN ORACLE9IAS CONTAINERS FOR J2EE (OC4J)

Containers for J2EE (OC4J), like all application servers, must extend the class loading capabilities of the J2SE to support the requirements of J2EE. In this section, we will look at the specifics of class loading in OC4J.

As in all application servers, OC4J relies on various configuration files that are not part of the J2EE specification. In addition to controlling other features of OC4J, there are two options made available in some of these files that are relevant to class loading, both of which allow specification of additional code-sources. Since the files are formatted in XML, the class loading options take the form of XML tags:

- **<classpath path="...">** A semicolon-separated list of directories, jar or zip files. Paths can be either relative or absolute.

- **<library path="..." />** Same as **<classpath>**, but directories are treated specially: any jar or zip files contained in a directory listed in the path attribute are also included. The directory search is not recursive.

There are two categories of configuration files that support these tags:

- **Server-wide.** These files usually live in the “config” directory and affect all applications:
 1. *application.xml*⁵. Supports the **<library>** tag. Code-sources listed here are visible to all applications.
 2. *global-web-application.xml*. Supports the **<classpath>** tag. Code-sources listed here are added to all web-applications, and are not visible above that layer. Note that even though each web application will use the same path to reference these code-sources, using this tag causes class duplication since each web application has its own class loader.
- **Application-specific.** These files live alongside the standard deployment descriptor files within the EAR and WAR files and extend their functionality:
 3. *orion-application.xml*. Extends *application.xml* and supports the **<library>** tag. Code-sources listed here are visible to all modules within an application.
 4. *orion-web.xml*. Extends *web.xml* and supports the **<classpath>** tag. Code-sources listed here are visible only to the web-application.

(For more detailed configuration details, refer to the OC4J Users Guide on the Oracle Technology Network at

http://otn.oracle.com/tech/java/oc4j/pdf/oc4j_so_usersguide_r2.pdf).

A typical class loader tree in an OC4J instance might be as follows:

⁵ This file would be better named “global-application.xml” to avoid confusion with the standard EAR deployment descriptor of the same name.

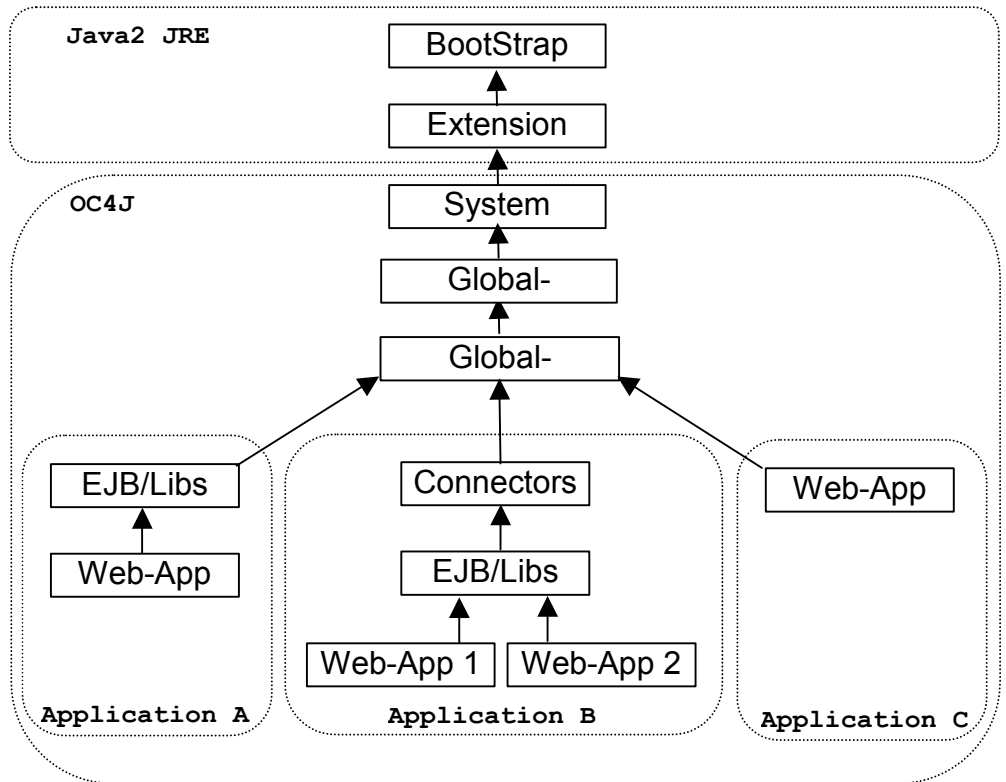


Figure 6: Typical OC4J class loader tree

Each of the OC4J loaders is described briefly below:

The **system** loader is the standard J2SE system loader, configured to contain all of the J2EE API and OC4J classes.

The **global-connectors** loader contains all code-sources from RAR files referenced in the global oc4j-connectors.xml file.

The **global-application** loader contains all code-sources from any <library> tags in the global application.xml file. Note that by default, the “home/lib” directory inside the OC4J installation directory is listed here, so any jars placed in that directory will be added.

The **connectors** loader contains all code-sources from any application RAR files.

The **ejb/libraries** loader contains all code-sources from any <ejb> tags in application.xml, and from any <library> tags in the orion-application.xml file.

The **web-application** loader contains all code-sources from any WAR files, any <classpath> tags from orion-web.xml and any <library> tags from the global-web-application.xml file.

The tree is constructed based on the contents of the configuration files, as well as the contents of each application. For example, application “C” above contains only a single web-module, and does not specify a <library> tag in an orion-application.xml file, so there is only a web-application loader. Similarly, if there are no <library> tags in the global application.xml file, the “GlobalApplication” loader is not created and the “System” loader will be the common application parent.

Cross Application Sharing

OC4J provides a mechanism to support the “some applications” visibility discussed earlier, in which classes need to be shared between two or more applications without duplication. The mechanism is simple and flexible: any application can declare that another application is its parent. OC4J then arranges for one of the parent application’s ClassLoaders to be the parent of the highest loader in the child applications.

For example, by making an “application” that contains no modules, only the shared jars, and listing them in <library> tags in the orion-application.xml file, the following arrangement can be created:

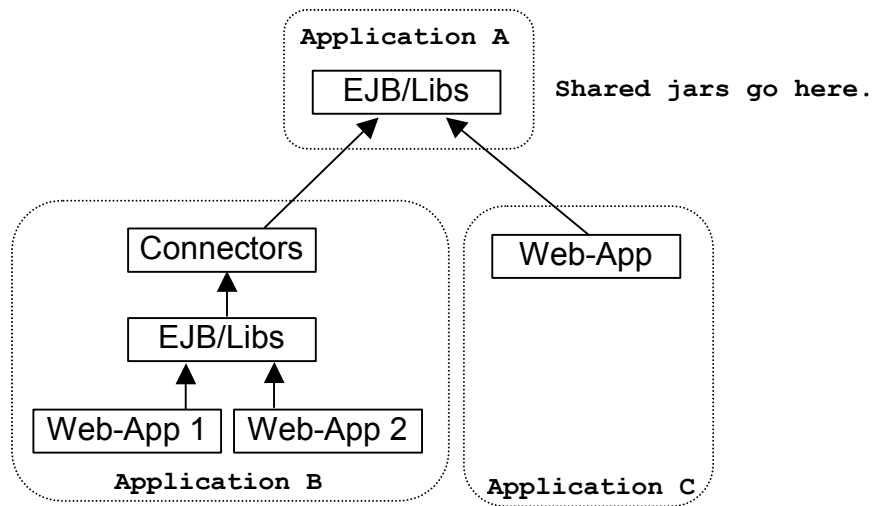


Figure 7: A parent application with shared jars

The “parent” declaration is made in the <application> tag within server.xml. For the above example, these might appear as follows:

```

<application name="A" path="a.ear" />
<application name="B" path="b.ear" parent="A" />
  
```

```
<application name="C" path="c.ear" parent="A" />
```

Configuration Option Summary

Throughout this article, we have described a number of configuration options available to OC4J application developers, ranging from basic J2SE options, through generic J2EE options, to OC4J specific options. The table below summarizes all of these for convenient reference:

ClassLoader	Configuration Options	Type
Bootstrap	Command-line: -Xbootclasspath (Sun VM only)	JVM
	System property: sun.boot.class.path (Sun VM only)	JVM
	META-INF/manifest.mf Class-Path of above jars	JRE
Extensions	System property: java.ext.dir. Default: \$JAVA_HOME/jre/lib/ext	JRE
	META-INF/manifest.mf Class-Path of above jars	JRE
System	Command-line: -classpath	JRE
	Command-line: -cp	JRE
	Command-line: -jar	JRE
	System property: java.class.path	JRE
	META-INF/manifest.mf Class-Path of above jars	JRE
global-application	<library> tags in global application.xml	OC4J
	META-INF/manifest.mf Class-Path of above jars	JRE
"cross application"	parent attribute in <application> tag in server.xml	OC4J
Connectors	RAR file: all jars at the root.	J2EE
	META-INF/manifest.mf Class-Path of above jars	JRE
ejb/libraries	<ejb> tags from application.xml	J2EE
	<library> tags from orion-application.xml	OC4J
	META-INF/manifest.mf Class-Path of above jars	JRE
web-application	WAR file: META-INF/manifest.mf Class-Path	J2EE
	WAR file: META-INF/classes	J2EE
	WAR file: META-INF/lib	J2EE
	<classpath> tags from orion-web.xml	OC4J
	<library> tags from global-web-application.xml	OC4J
	META-INF/manifest.mf Class-Path of above jars	JRE

The shaded options are all available within an EAR file.

Note that support for environment variables (e.g CLASSPATH) as configuration options varies depending on the JVM and OS and are therefore non-portable.

COMMON CLASSLOADING PROBLEMS IN J2EE

Next, let's turn our attention to some of the most common problems developers will find when dealing with classloading issues. Most classloading problems in J2EE are related to *visibility* and surface as one of a small set of exceptions.

Not Enough Visibility

This situation occurs when a required class is not visible from within the current scope. The problem can surface as a:

- ***ClassNotFoundException***. This exception occurs during dynamic loading, via any method that explicitly loads a class (e.g. `Class.forName()` or `ClassLoader.loadClass()`). The “framework” scenario, described in the “Linking Classloaders” section earlier, is a typical example of this exception.
- ***NoClassDefFoundException***. This exception is thrown when code tries to instantiate an object using the *new* operator or when dependencies of a previously loaded class cannot be resolved. The latter case is often obscured by lazy-loading and hidden dependencies. Consider the following situation:
 1. Your code creates an instance of class `Foo`.
 2. Buried in the code of the `Foo.doIt()` method, `Foo` instantiates and uses class `XYZ`.
 3. Class `XYZ` lives in a different package and *a different jar* than either `Foo` or your code.
 4. You were unaware of the dependency on `XYZ`, so you did not include its jar in your application.
 5. Because of lazy-loading, `XYZ` is not loaded in step 1.
 6. Your code invokes `aFoo.doIt()` and a `NoClassDefFoundException` is thrown for `XYZ`. Surprise!

This situation is fairly common, and is often due to a much longer chain of dependencies, thus further separating the cause and effect.

Longer dependency chains are also more likely to span multiple namespaces (i.e. `ClassLoaders`), increasing the likelihood of a visibility error.

Too Much Visibility

This problem occurs when a class is duplicated, and surfaces as a:

Cross-application dependencies that are managed by copying the dependent jars into each application.

Application reloading. When an application is reloaded, the container creates a new `ClassLoader` and uses it to reload the application classes. In this scenario, however, the original classes may still be accessible. Normally, the existence of duplicate classes is not a problem (other than footprint). In order for an exception to occur, an instance created in one namespace (classloader) must be passed into the

namespace of the other. This can be achieved through any storage facility visible to both namespaces, including:

ClassCastException. Often the cause of this exception is simple and obvious. In the “same class, different loader” situation, however, the fact that the source and target types have the same class name can be both surprising and frustrating. Class duplication usually occurs in two scenarios:

- **Cross-application dependencies.** Dependencies between application modules that are managed by copying the dependent jars into each application
- **Application reloading.** When an application is reloaded, the container creates a new `ClassLoader` and uses it to reload the application classes. In this scenario, however, the original classes may still be accessible. Normally, the existence of duplicate classes is not a problem (other than footprint). In order for an exception to occur, an instance created in one namespace (classloader) must be passed into the namespace of the other. This can be achieved through any storage facility visible to both namespaces, including:
 - Static fields
 - Global collections
 - JNDI

The latter case normally does not result in a problem. The same *instance* is not shared across namespaces because a copy is created through serialization. The copy resulting from the de-serialization process will use the class from the local namespace.

Some JNDI implementations do not serialize objects that are locally bound and retrieved. Thus, when applications in the same JVM store and retrieve an object, that instance *is* shared, and the duplicate class problem can occur.

Mangled Visibility

This problem occurs when something is seriously wrong. Fortunately, these problems rarely occur and are all classified as errors:

- ***IncompatibleClassChangeError.*** This error indicates that a superclass or interface has changed.
- ***ClassCircularityError.*** This error indicates that the current class exists as one of its own superclasses or superinterfaces.
- ***UnsupportedClassVersionError.*** This error usually occurs when loading a class compiled in a more recent JDK version than the one in which it is running.

- **VerifyError.** This error occurs when the code in the class violates one of the constraints imposed by the JVM.
- **ClassFormatError.** This error indicates that the class file format is invalid, often due to corruption.

J2EE CLASSLOADING BEST PRACTICES

If you've stuck with us this far, congratulations! It's a lot of detail to absorb. Let's conclude with a set of guidelines that distill many of the ideas presented earlier:

- **Declare dependencies.** Make dependencies explicit. Hidden or unknown dependencies will be left behind when you move your application to another environment.
- **Group dependencies.** Ensure that all dependencies are visible at the same level or above. If you must move a library, make sure all dependencies are still visible.
- **Minimize visibility.** Dependency libraries should be placed at the lowest visibility level that satisfies all dependencies. For example, if a library is only used by a single web application, it should be included in the WEB-INF/lib directory of the WAR file.
- **Share libraries.** Avoid duplicating libraries if you can. Use the "parent" attribute to share classes across a set of applications. Use the <library> tag in the global application.xml file to share classes across all applications.
- **Keep configurations portable.** In general, create configurations that are as portable as possible. Specify configuration options in the following order:
 1. Standard J2EE options.
 2. Options that can be expressed *within* your EAR file.
 3. Server-level options.
 4. J2SE extension options.
- **Use the correct loader.**

If you call `Class.forName()`, always explicitly pass the loader returned by:

```
Thread.currentThread().getContextClassLoader()
```

If you are loading a properties file, use :

```
Thread.currentThread().getContextClassLoader().getResourceAsStream()
```

CONCLUSION

We hope that you have had some “ahah!” moments while reading this article, and that you now find some of the mysteries of classloading a bit more clear. Now it’s time to productively apply these insights in your J2EE applications!



Classloading in Oracle9iAS Containers for J2EE
January 2003
Authors Bryan Atsatt, Debu Panda

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

Copyright © 2003 Oracle Corporation
All rights reserved.