WebSphere® software

IBM

Redpaper

Jasper Chui

WebSphere Application Server V6.1: Class loader problem determination

Troubleshooting class loader issues with applications running on WebSphere® Application Server version 6.1 requires a basic understanding of how class loaders work within WebSphere.

This paper begins with an introduction on understanding class loaders and class loaders within WebSphere Application Server. This includes an explanation of configuration options for the class loader and how they can be used with your application to achieve a specific behavior. Next we provide information that will help you recognize when a class loader problem occurs and propose some solutions to help you resolve them. This process includes instructions on collecting class loader specific traces, identifying and gathering exception specific trace information, analyzing the traces, and an explanation of possible root causes and solutions to these common class loader exceptions.

A sample application, ClassloaderSampleEAR.zip, is available for download with this paper. This sample can be used to reproduce some of the common class loader exceptions visited in this document. We encourage you to review it and to follow the analysis described in this document to help you better understand how to recognize and how to resolve class loader issues within WebSphere Application Server V6.1. The instructions to install and to use this application are in "Installing and configuring the sample application" on page 55.

ibm.com/redbooks   **1**

Introduction to class loaders

Understanding how the Java™ and WebSphere class loaders work is critical to packaging and deploying J2EE™ applications. Failure to set up the class loaders properly most likely results in a cascade of the infamous class loading exceptions (such as ClassNotFoundException) when trying to start your application.

This paper explains class loaders and how to customize the behavior of the WebSphere class loaders to suit your particular application's requirements. It includes the following:

► Brief introduction to Java class loaders
► WebSphere class loaders overview
► Configuring WebSphere for class loaders

Brief introduction to Java class loaders

Class loaders enable the Java virtual machine (JVM™) to load classes. Given the name of a class, the class loader locates the definition of this class. Each Java class must be loaded by a class loader.

When you start a JVM, you use three class loaders: the bootstrap class loader, the extensions class loader, and the application class loader.

The bootstrap class loader is responsible for loading only the core Java libraries, which are vm.jar, core.jar, and so on, in the *Java_home*/jre/lib directory, where *Java_home* is the installation directory for the JDK™. This class loader, which is part of the core JVM, is written in native code.

The extensions class loader is responsible for loading the code in the extensions directories (*Java_home*/jre/lib/ext or any other directory specified by the java.ext.dirs system property). This class loader is implemented by the sun.misc.Launcher$ExtClassLoader class.

The application class loader is responsible for loading the code that is found on java.class.path, which ultimately maps to the system CLASSPATH variable. This class loader is implemented by the sun.misc.Launcher$AppClassLoader class.

The parent-delegation model is a key concept to understand when dealing with class loaders. It states that a class loader delegates class loading to its parent before trying to load the class itself. The parent class loader can be either another custom class loader or the bootstrap class loader. But what is very important is that a class loader can only delegate requests to its parent class loader, never to its child class loaders (it can go up the hierarchy but never down).

The extensions class loader is the parent for the application class loader. The bootstrap class loader is the parent for the extensions class loader. The class loaders hierarchy is shown in Figure 1.

If the application class loader needs to load a class, it first delegates to the extensions class loader, which, in turn, delegates to the bootstrap class loader. If the parent class loader cannot load the class, the child class loader tries to find the class in its own repository. In this manner, a class loader is only responsible for loading classes that its ancestors cannot load.
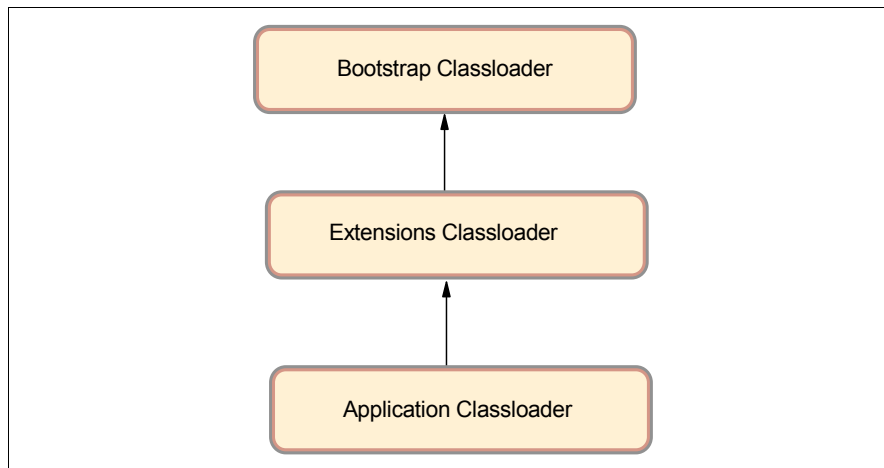

Figure 1   Java class loaders hierarchy

WebSphere class loaders overview

Note: Keep in mind when reading the following discussion that each JVM has its own setup of class loaders. In a WebSphere environment hosting multiple application servers (JVMs), this means the class loaders for the JVMs are completely separated even if they are running on the same physical machine.

Also note that the Java Virtual Machine (JVM) uses class loaders called the extensions and application class loaders. As you will see, the WebSphere run time also uses class loaders called extensions and application class loader, but despite their names they are not the same as the JVM ones.

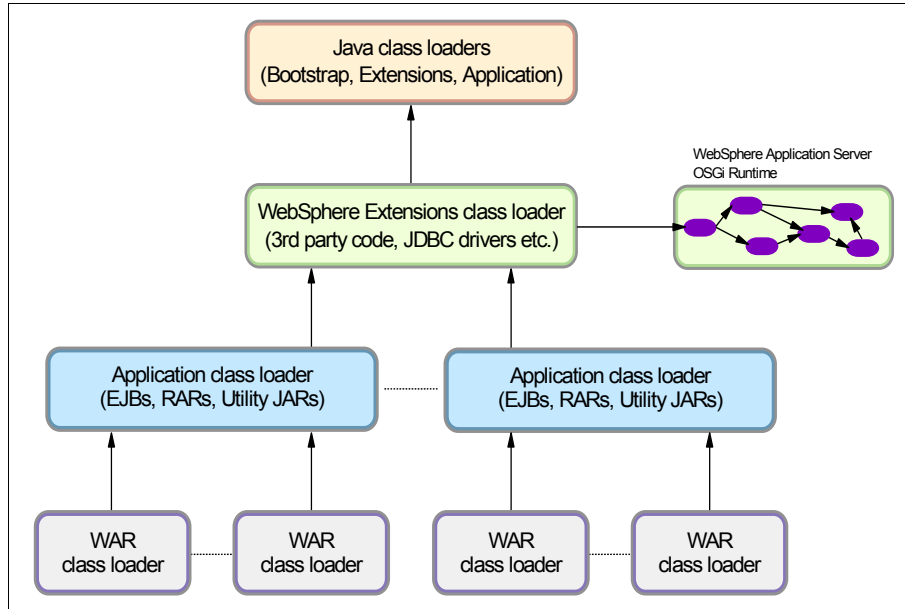WebSphere provides several custom delegated class loaders, as shown in Figure 2 on page 4.

Figure 2   WebSphere class loaders hierarchy

The top box represents the Java (bootstrap, extensions, and application) class loaders. WebSphere loads just enough here to get itself bootstrapped and to initialize the WebSphere extensions class loader.

WebSphere extensions class loader

New in V6.1: The WebSphere extensions class loader is where WebSphere itself is loaded. In previous versions of WebSphere, the run time was loaded by this single class loader. However, beginning with WebSphere Application Server V6.1, WebSphere is now packaged as a set of OSGi bundles. Each OSGi bundle is loaded separately by its own class loader. This network of OSGi class loaders is then connected to the extensions class loader and the rest of the class loader hierarchy through an OSGi gateway class loader.

Despite this architectural change in the internals of how WebSphere loads its own classes, there is no behavioral change as far as your applications are concerned. They still have the same visibility, and the same class loading options still exist for your application.

New in V6.1: In previous versions of WebSphere Application Server the WebSphere run time class files were stored in the classes, lib, lib/ext, and installedChannels directories in the *app_server_root* directory. Because of the OSGi packaging, these directories no longer exist and the run time classes are now stored in the *app_server_root*/plugins directory.

The class path used by the extensions class loader is retrieved from the ws.ext.dirs system property, which is initially derived from the WAS_EXT_DIRS environment variable set in the setupCmdLine script file. The default value of ws.ext.dirs is displayed in Example 1 .

Example 1   Default value of we.ext.dirs

```
SET
WAS_EXT_DIRS=%JAVA_HOME%\lib;%WAS_HOME%\classes;%WAS_HOME%\lib;%WAS_HOME%\insta
lledChannels;%WAS_HOME%\lib\ext;%WAS_HOME%\web\help;%ITP_LOC%\plugins\com.ibm.e
tools.ejbdeploy\runtime
```

Each directory listed in the ws.ext.dirs environment variable is added to the WebSphere extensions class loaders class path, and every JAR file and ZIP file in the directory is added to the class path.

As you can see, even though the classes and installedChannels directories no longer exist in the *app_server_root* directory for V6.1, the setupCmdLine script still adds them to the extensions class path. This means that if you previously added your own JAR files to, for example, the *app_server_root*/lib directory, you could create this directory, and add your JAR files to it and they would still be loaded by the extensions class loader. However, this is not recommended and you should really try to migrate away from such a setup.

On the other hand, if you developed Java applications that rely on the WebSphere JAR files that were previously in the *app_server_root*/lib directory, you will need to modify your application to retain compatibility. WebSphere Application Server V6.1 provides two thin client libraries designed specifically for such applications: one administrative client library and one Web services client library. These thin client libraries can be found in the following *app_server_root*/runtimes directories:

► com.ibm.ws.admin.client_6.1.0.jar

► com.ibm.ws.webservices.thinclient_6.1.0.jar

These libraries provide everything your application might need for connecting to and working with WebSphere.

New in V6.1: WebSphere Application Server V6.1 gives you the ability to restrict access to internal WebSphere classes so that your applications do not make unsupported calls to WebSphere classes not published in the official WebSphere Application Server API. This setting is a per-server (JVM) setting called Access to internal server classes.

The default setting is Allow, meaning that your applications can make unrestricted calls to non-public internal WebSphere classes. This is not recommended and may be prohibited in future releases. Therefore, as an administrator, it is a good idea to switch this setting to *Restrict* to see if your applications still work. If they depend on non-public WebSphere internal classes, you will receive a ClassNotFoundException, and in that case you can switch back to Allow. Your developers should then try to migrate their applications so that they do not make unsupported calls to the WebSphere internal classes in order to retain compatibility with future WebSphere Application Server releases.

Application and Web module class loaders

J2EE applications consist of five primary elements: Web modules, EJB™ modules, application client modules, resource adapters (RAR files), and utility JARs. Utility JARs contain code that can be used by both EJBs and servlets. Utility frameworks such as log4j are good examples of a utility JAR.

EJB modules, utility JARs, resource adapter files, and shared libraries associated with an application are always grouped together into the same class loader. This class loader is called the application class loader. Depending on the class loader policy, this class loader can be shared by multiple applications (EARs) or be unique for each application, which is the default.

By default, Web modules receive their own class loader, a WAR class loader, to load the contents of the WEB-INF/classes and WEB-INF/lib directories. You can modify the default behavior by changing the application's WAR class loader policy. The default is to have a class loader for each WAR file in the application (this setting was called Module in previous releases). If the WAR class loader policy is set to Single class loader for application (called Application in previous releases), the Web module contents are loaded by the application class loader in addition to the EJBs, RARs, utility JARs, and shared libraries. The application class loader is the parent of the WAR class loader.

The application and the WAR class loaders are reloadable class loaders. They monitor changes in the application code to automatically reload modified classes. You can modify this behavior at deployment time.

Handling JNI code

Because a JVM only has a single address space and native code can only be loaded once per address space, the JVM specification states that native code may only be loaded by one class loader in a JVM.

This may cause a problem if, for example, you have an application (EAR file) with two Web modules that both need to load the same native code through a Java Native Interface (JNI™). Only the Web module that first loads the library will succeed.

To solve this problem, you can break out just the few lines of Java code that load the native code into a class on its own and place this file on WebSphere's application class loader (in a utility JAR). However, if you would deploy multiple such applications (EAR files) to the same application server (JVM), you would have to place the class file on the WebSphere extensions class loader instead to ensure the native code is only loaded once per JVM.

If the native code is placed on a reloadable class loader (such as the application class loader or the WAR class loader), it is important that the native code can properly unload itself should the Java code have to reload. WebSphere has no control over the native code and if it does not unload and load properly the application may fail.

If one native library depends on another one, things become even more complicated. Search for Dependent native library in the Information Center for more details.

Configuring WebSphere for class loaders

In the previous topic, you learned about WebSphere class loaders and how they work together to load classes. There are settings in WebSphere Application Server that allow you to influence WebSphere class loader behavior. This section discusses these options.

Class loader policies

For each application server in the system, the class loader policy can be set to Single or Multiple.

When the application server class loader policy is set to Single, a single application class loader is used to load all EJBs, utility JARs, and shared libraries within the application server (JVM). If the WAR class loader policy then was set

to Single class loader for application (or Application), then the Web module contents for this particular application are also loaded by this single class loader.

When the application server class loader policy is set to Multiple, which is the default, each application will receive its own class loader for loading EJBs, utility JARs, and shared libraries. Depending on whether the WAR class loader loading policy is set to class loader for each WAR file in application (or Module) or Single class loader for application (or Application), the Web module might or might not receive its own class loader.

Following is an example to illustrate. You have two applications, Application1 and Application2, running in the same application server. Each application has one EJB module, one utility JAR, and two Web modules. If the application server has its class loader policy set to Multiple, the default, and the class loader policy for all the Web modules are set to use a class loader for each WAR file in application (Module), also the default, the result is as shown in Figure 3.
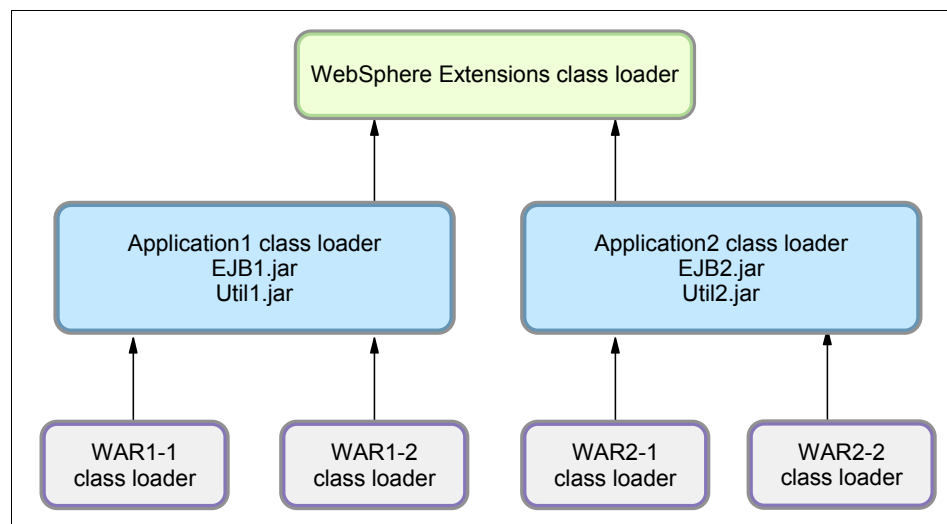

Figure 3   Class loader policies: Example 1

Each application is completely separated from the other, and each Web module is also completely separated from the other one in the same application. WebSphere's default class loader policies result in total isolation between the applications and the modules.

If we now change the class loader policy for the WAR2-2 module from class loader for each WAR file in application (Module) to Single class loader for application (Application), the result is shown in Figure 4.
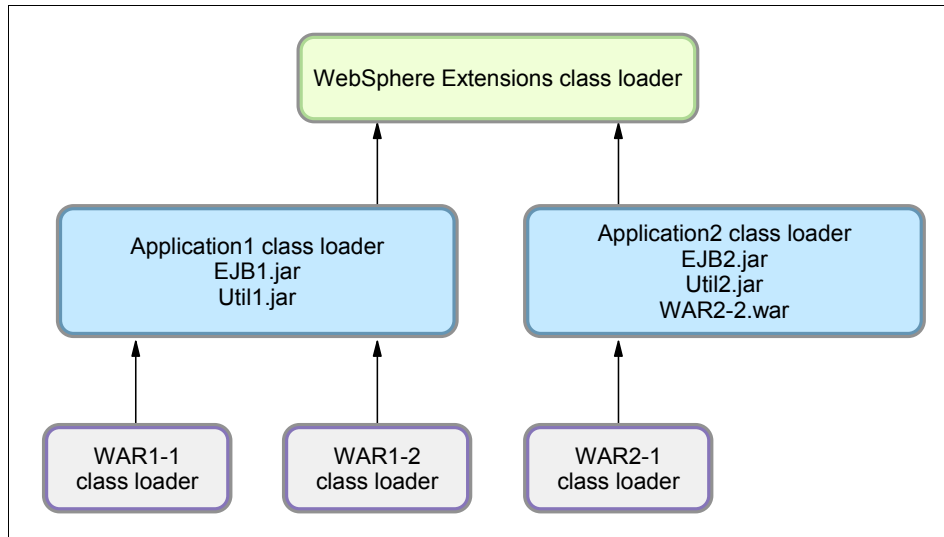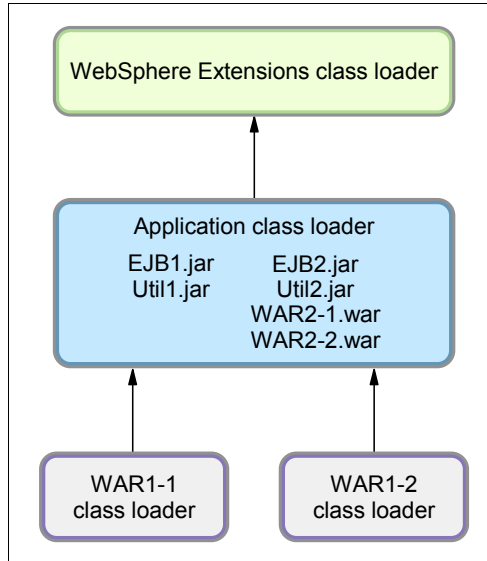

Figure 4   Class loader policies: Example 2

Web module WAR2-2 is loaded by Application2's class loader and classes, for example, in Util2.jar, we can see classes in WAR2-2's /WEB-INF/classes and /WEB-INF/lib directories.

As a last example, if we change the class loader policy for the application server from Multiple to Single and also change the class loader policy for WAR2-1 from class loader for each WAR file in application (Module) to Single class loader for application (Application), the result is as shown in Figure 5 on page 10.

Figure 5   Class loader policies: Example 3

There is now only a single application class loader loading classes for both Application1 and Application2. Classes in Util1.jar can see classes in EJB2.jar, Util2.jar, WAR2-1.war and WAR2-2.war. The classes loaded by the application class loader still cannot, however, see the classes in the WAR1-1 and WAR1-2 modules, because a class loader can only find classes by going up the hierarchy, never down.

Class loading/delegation mode

WebSphere's application class loader and WAR class loader both have a setting called the class loader order. This setting determines whether they should follow the normal Java class loader delegation mechanism or override it.

There are two possible options for the class loading mode:

► Classes loaded with parent class loader first
► Classes loaded with application class loader first

In previous WebSphere releases, these settings were called PARENT_FIRST and PARENT_LAST, respectively.

The default value for class loading mode is Classes loaded with parent class loader first (PARENT_FIRST). This mode causes the class loader to first delegate the loading of classes to its parent class loader before attempting to

load the class from its local class path. This is the default policy for standard Java class loaders.

If the class loading policy is set to Classes loaded with application class loader first (PARENT_LAST), the class loader attempts to load classes from its local class path before delegating the class loading to its parent. This policy allows an application class loader to override and provide its own version of a class that exists in the parent class loader.

> **Note:** The administrative console is a bit confusing at this point. On the settings page for a Web module, the two options for class loader order are Classes loaded with parent class loader first and Classes loaded with application class loader first. However, in this context, the "application class loader" really refers to the WAR class loader, so the option Classes loaded with application class loader first should really be called Classes loaded with WAR class loader first.

Assume you have an application, similar to Application1 in the previous examples, and it uses the popular log4j package to perform logging from both the EJB module and the two Web modules. Also assume that each module has its own, unique, log4j.properties file packaged into the module. It is tempting to configure log4j as a utility JAR so you only have a single copy of it in your EAR file.

However, if you do that, you might be surprised to see that all modules, including the Web modules, load the log4j.properties file from the EJB module. The reason is that when a Web module initializes the log4j package, the log4j classes are loaded by the application class loader. log4j is configured as a utility JAR. log4j then looks for a log4j.properties file on its class path and finds it in the EJB module.

Even if you do not use log4j for logging from the EJB module and the EJB module does not, therefore, contain a log4j.properties file, log4j does not find the log4j.properties file in any of the Web modules anyway. The reason is that a class loader can only find classes by going up the hierarchy, never down.

To solve this problem, you can do one of the following:

▶ Create a separate file, for example, Resource.jar, configure it as a utility JAR, move all log4j.properties files from the modules into this file, and make their names unique (like war1-1_log4j.properties, war1-2_log4j.properties, and ejb1_log4j.properties). When initializing log4j from each module, tell it to load the proper configuration file for the module instead of the default (log4j.properties).

- ▶ Keep the log4j.properties for the Web modules in their original place (/WEB-INF/classes), add log4j.jar to both Web modules (/WEB-INF/lib) and set the class loading mode for the Web modules to Classes loaded with application class loader first (PARENT_LAST). When initializing log4j from a Web module, it loads the log4j.jar from the module itself and log4j would find the log4j.properties on its local classpath, the Web module itself. When the EJB module initializes log4j, it loads from the application class loader and it finds the log4j.properties file on the same class path, the one in the EJB1.jar file.
- ▶ Merge, if possible, all log4j.properties files into one, and place it on the application class loader, in a Resource.jar file, for example.

Singletons: The Singleton pattern is used to ensure that a class is instantiated only once. However, *once* only means *once for each class loader.* If you have a Singleton being instantiated in two separated Web modules, two separate instances of this class are created, one for each WAR class loader. So in a multi-class loader environment, special care must be taken when implementing Singletons.

Shared libraries

Shared libraries are files used by multiple applications. Examples of shared libraries are commonly used frameworks like Apache Struts or log4j. You use shared libraries typically to point to a set of JARs and associate those JARs to an application, a Web module, or the class loader of an application server. Shared libraries are especially useful when you have different versions of the same framework you want to associate to different applications.

Shared libraries are defined using the administration tools. They consist of a symbolic name, a Java class path, and an optional native path for loading JNI libraries. They can be defined at the cell, node, server, or cluster level. However, simply defining a library does not cause the library to be loaded. You must associate the library to an application, a Web module, or the class loader of an application server for the classes represented by the shared library to be loaded. Associating the library to the class loader of an application server makes the library available to all applications on the server.

Note: If you associate a shared library to an application, do not associate the same library to the class loader of an application server.

Associate a shared library with an application

You can associate the shared library to an application in one of two ways:

▶ You can use the administration tools. For information about using this method, see "Associate the shared library with an application" on page 39.

▶ You can use the manifest file of the application and the shared library. The shared library contains a manifest file that identifies it as an extension. The dependency to the library is declared in the application's manifest file by listing the library extension name in an extension list.

　For more information about this method, search for installed optional packages in the Information Center.

Associate a shared library with a server class loader

Shared files are associated with the class loader of an application server using the administrative tools.

The settings are found in the Server Infrastructure section.

1. Expand the **Java and Process Management**.

2. Select **Class loader**, and then click the **New** button to define a new class loader.

3. After you have defined a new class loader, you can modify it and, using the **Shared library references** link, you can associate it to the shared libraries you need.

Problem determination for class loader exceptions

This section takes you through the process of identifying and resolving common class loader problems. It helps you identify when a class loader problem occurs and helps you to collect class loader specific traces and other diagnostic data. It will help you analyze the diagnostic data and identify possible root causes of the problem and the solution for those root causes.

> **Note:** The examples in this chapter were created using the sample application, ClassloaderSampleEAR, available for download with this paper (see "Installing and configuring the sample application" on page 55).

Identify symptoms

Problems related to class loaders are normally identified during the course of diagnosing general application errors. Depending on how the application handles unexpected exceptions, this can look like any other type of application failure (HTTP 404 Page cannot be displayed, for example).

> **Tip:** A common example of a class loader problem occurring is when **log4j** is not producing logs. The root cause is that the class loader is loading the WebSphere log properties files instead of your own properties files.
>
> For more information about using custom logging, see the following Web site:
>
> ► Jakarta Commons Logging
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?t
> opic=/com.ibm.websphere.base.doc/info/aes/ae/ctrb_classload_jcl.h
> tml

Collect SystemOut

Indications of class loader issues are not readily apparent until you view the JVM SystemOut log. Class loader problems normally appear as an exception and are accompanied with a Java stack trace.

Determine the type of error

Typical class loader exceptions include the following:

► java.lang.ClassCastException
► java.lang.ClassNotFoundException
► java.lang.NoClassDefFoundError
► java.lang.NoSuchMethodError, java.lang.IllegalArgumentException
► java.lang.UnsatisfiedLinkError
► java.lang.VerifyError

Scan the SystemOut log for a class loader exception. When a class loader exception occurs, note the following:

► The accompanying stack trace
► The error message describing the cause of the exception
► The class causing the exception

Example 2 on page 15 illustrates an example of a loader exception.

Example 2   Example class loader exception

```
[12/04/07 11:05:13:468 EDT] 00000027 ServletWrappe I   SRVE0242I:
[ClassloaderSampleEAR] [/ClassCastExceptionWeb] [LoadClass]:
Initialization successful.
[12/04/07 11:05:13:531 EDT] 00000027 SystemErr     R
java.lang.ClassCastException: bean.Root incompatible with bean.Leaf

In this example we see that a ClassCastException is thrown by the
Servlet LoadClass within the application ClassLoaderSampleEAR and web
module ClassCastExceptionWeb complaining that the class bean.Root is
incompatible with bean.Leaf.
```

Recreate the problem and collect traces

After you determine that a class loader issue occurred, you need to collect the following trace to begin diagnostics:

► Class loader trace.

   See "Class loader trace" on page 47 for information on enabling this trace.

In addition, if you think you might need to contact IBM® support at some point and only want to recreate the problem once, collect the following:

► JVM class loader and bootstrap traces.

   See "JVM class loader and bootstrap traces" on page 48 for information on enabling this trace.

The traces contain exception-specific information that will help you understand the problem.

1. Enable the traces.
2. Stop the application server and clear the existing log files. This ensures that the trace information is fresh.
3. Restart the server.
4. Reproduce the problem, and collect all the log files generated.

Tip: By default the application log files and trace enabled log files are located in the directory *profile_home*\logs*server_name*\.

Now that you have the information you need to diagnose the problem, proceed to the appropriate problem area.

Each class loader problem section will help you do the following:

1. Analyze the class loader trace to find information pertinent to the exception or error you are encountering. This information helps you to identify the specific cause of the class loader exception.

2. Evaluate the data to determine the cause for the exception and error.

3. Examine the possible root causes and solutions for these class loader exceptions and errors.

If you do not find one of these exceptions or need further help, go to "The next step" on page 54 for information on performing online searches using your symptom and collecting data to contact IBM technical support.

ClassCastException

This section takes you through the process of diagnosing and resolving ClassCastException errors.

Analyze the class loader trace

First, find the ClassCastException in the trace. Note the following:

► The classes that caused the exception

► The line in the application where the error occurs

Next, search the trace for information about the classes referenced in the exception. Note the following:

► The class loader that loaded the class files

► The location of the class files.

Example

In this Example 3 , the trace tells you that the ClassCastException is caused by an incompatibility of the class bean.Root with bean.Leaf. The error occurs at line 50 in the application.

Example 3   Example ClassCastException

```
[12/04/07 11:05:13:531 EDT] 00000027 SystemErr     R
java.lang.ClassCastException: bean.Root incompatible with bean.Leaf
[12/04/07 11:05:13:531 EDT] 00000027 SystemErr     R    at
servlet.LoadClass.doPost(LoadClass.java:50)
```

```
[12/04/07 11:05:13:531 EDT] 00000027 SystemErr     R   at
servlet.LoadClass.doGet(LoadClass.java:36)
[12/04/07 11:05:13:531 EDT] 00000027 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:743)
[12/04/07 11:05:13:531 EDT] 00000027 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
[12/04/07 11:05:13:531 EDT] 00000027 SystemErr     R   at
com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.j
ava:966)
```

A search for the two classes in the trace, bean.Root and bean.Leaf, provides the
following information.

Example 4 shows the information for bean.Root.

Example 4   bean.Root

```
[12/04/07 11:05:13:437 EDT] 00000027 CompoundClass >  findClass
name=bean.Root this=com.ibm.ws.classloader.CompoundClassLoader@29582958
Entry
[12/04/07 11:05:13:437 EDT] 00000027 ReloadableCla 3   adding path to
reload cache
                                     C:\UtilityJar.jar
[12/04/07 11:05:13:437 EDT] 00000027 CompoundClass 3   class bean.Root
found in SinglePathClassProvider :
com.ibm.ws.classloader.SinglePathClassProvider@14c014c classpath =
C:\UtilityJar.jar
[12/04/07 11:05:13:437 EDT] 00000027 CompoundClass >  loadClass
name=java.lang.Object
this=com.ibm.ws.classloader.CompoundClassLoader@29582958 Entry
[12/04/07 11:05:13:453 EDT] 00000027 CompoundClass 3   loaded bean.Root
from this=
com.ibm.ws.classloader.CompoundClassLoader@29582958
   Local ClassPath:
C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps\jasperch
uiNode02Cell\ClassloaderSampleEAR.ear\ClassCastExceptionWeb.war\WEB-INF
\classes;C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps
\jasperchuiNode02Cell\ClassloaderSampleEAR.ear\ClassCastExceptionWeb.wa
r;C:\UtilityJar.jar
   Delegation Mode: PARENT_FIRST
```

Example 5 on page 18 shows the information for bean.Leaf.

Example 5   bean.Leaf

```
[12/04/07 11:05:13:453 EDT] 00000027 CompoundClass >  findClass
name=bean.Leaf this=com.ibm.ws.classloader.CompoundClassLoader@29582958
Entry
[12/04/07 11:05:13:453 EDT] 00000027 CompoundClass 3   class bean.Leaf
found in SinglePathClassProvider :
com.ibm.ws.classloader.SinglePathClassProvider@14c014c classpath =
C:\UtilityJar.jar
[12/04/07 11:05:13:453 EDT] 00000027 CompoundClass <  findClass Exit
[12/04/07 11:05:13:453 EDT] 00000027 CompoundClass 3   loaded bean.Leaf
from this=
com.ibm.ws.classloader.CompoundClassLoader@29582958
    Local ClassPath:
C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps\jasperch
uiNode02Cell\ClassloaderSampleEAR.ear\ClassCastExceptionWeb.war\WEB-INF
\classes;C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps
\jasperchuiNode02Cell\ClassloaderSampleEAR.ear\ClassCastExceptionWeb.wa
r;C:\UtilityJar.jar
    Delegation Mode: PARENT_FIRST
```

Evaluate the trace data

Do the following:

► Determine if the source and target are loaded by the same class loader. If not, see "Class loader for source and target are different" on page 20.

► Using the information from the trace, examine the source code. The source object should be an instance of the target class. If not, see "Source object is not an instance of the target class" on page 20.

► Check the source to see if the failure occurred while performing a narrow operation on EJBs. Example 6 shows a sample narrow operation being performed for an EJB remote interface. If a narrow operation is being performed, see "Application improperly performs a narrow operation" on page 21.

Example 6   Narrow operation

```
java.lang.Object ejbHome =
initialContext.lookup("java:comp/env/ejb/myEJB");
myEJBHome =
(myEJBHome)javax.rmi.PortableRemoteObject.narrow((org.omg.CORBA.Object)
ejbHome, myEJBHome.class);
```

Example

From the example traces, you know the following:

► The ClassCastException is caused by an incompatibility of the class bean.Root with bean.Leaf.

► The error occurs at line 50 in the application.

► The class loader that loaded the bean.Root class and the bean.Leaf class is the same class loader:

```
SinglePathClassProvider:
com.ibm.ws.classloader.SinglePathClassProvider@14c014c
```

> **Tip:** If the class is located by the Java class loader: extensions class loader, application class loader or the bootstrap class loader, specifying `-Dibm.cl.verbose=<name>` in the generic JVM argument allows you to trace the way these class loaders find and load a given class.
>
> Use the full name of the class, including the package name, for example: `-Dibm.cl.verbose=bean.Root`.
>
> See "Additional class loader diagnostics" on page 52 for more information.

► The files are located in C:\UtilityJar.jar.

You can also reach this conclusion if you search for the classes with the Class Loader Viewer.

The exception fails on line 50 of the servlet class LoadClass. The source for the sample application has the code shown in Example 7 at that location:

Example 7   Source code from ClassCastExceptionWeb sample.

```
49        Root r = new Root();
50        Leaf l = (Leaf)r;
```

The code is trying to cast the object r to the class bean.Leaf, which is not possible since the object r is of the type bean.Root. This problem is addressed in "Source object is not an instance of the target class" on page 20.

ClassCastException root causes

This section lists the possible root causes and possible solutions for a ClassCastException error.

Source object is not an instance of the target class

Ensure that the source class exists within the ancestry of the target class or is the same class. You can determine this by examining the output of the following:

```
System.out.println( source.getClass().getName() + ":" +
target.getClass().getName());
```

Where *source* represents the object being cast and the *target* represents the class being cast to. From the sample application, Example 7 on page 19, r is the source object and Leaf is the target class.

Or use a **javap** command as shown in Example 8 .

Example 8   Using the javap command to investigate ancestry of a class.

```
javap java.util.HashMap
Compiled from "HashMap.java"
public class java.util.HashMap extends java.util.AbstractMap implements
java.util.Map, java.lang.Cloneable, java.io.Serializable {
```

Resolve the problem

After you confirm that this is the cause of the problem, review your code to ensure that it is performing the correct and proper casting for the target class. In the sample, the Leaf class has the ancestry:

```
public class Leaf extends Root
```

It would be more appropriate to have the following:

```
49      Root r = new Root();
50      Leaf l = new Leaf();
51      Root c = (Root) l;
```

Class loader for source and target are different

When the class loader that loaded the source object is different than the class loader that loaded the target class you will have a problem.

Resolve the problem

To resolve this, ensure that the classes are loaded by the same class loader. You should already have this information from reviewing the trace. If they are not, refer to "Using custom JAR files for your application" on page 41 for suggestions on using custom jars in your application and to ensure that the two classes are loaded by the same class loader, for example, by placing the files in the same JAR file if possible.

Application improperly performs a narrow operation

When an application fails to perform, or improperly performs, a narrow operation problem will occur.

The narrow operation must be performed on objects returned from a look up for a remote interface of an EJB (Not for the local interface of an EJB).

Resolve the problem

Make sure that the target class submitted to the narrow method is the exact remote interface of the EJB.

> **Where to go from here:** If these suggestions do not resolve your problem, go to "The next step" on page 54 for information on performing online searches of known class loader issues. The referenced section also contains information on what you need to collect in order to engage IBM support.

ClassNotFoundException

This section takes you through the process of diagnosing and resolving ClassNotFoundExceptions errors.

Analyze the class loader trace

First, search for the ClassNotFoundException and note the following:

► The class that could not be found

► The line number in the application where the failure occurs

Next, search the trace for information about the class.

► Note the attempts to find the class by each class loader.

Example

In Example 9 the ClassNotFoundException is being thrown when trying to locate the class `sample.ReferenceClass`. The failure occurs at line 34 in the application.

Example 9   Example ClassNotFoundException

```
[12/04/07 10:56:26:093 EDT] 00000024 SystemErr     R
java.lang.ClassNotFoundException: sample.ReferenceClass
```

```
[12/04/07 10:56:26:093 EDT] 00000024 SystemErr     R   at
java.lang.Class.forNameImpl(Native Method)
[12/04/07 10:56:26:093 EDT] 00000024 SystemErr     R   at
java.lang.Class.forName(Class.java:131)
[12/04/07 10:56:26:093 EDT] 00000024 SystemErr     R   at
servlet.LoadClass.doPost(LoadClass.java:34)
[12/04/07 10:56:26:093 EDT] 00000024 SystemErr     R   at
servlet.LoadClass.doGet(LoadClass.java:26)
[12/04/07 10:56:26:093 EDT] 00000024 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:743)
[12/04/07 10:56:26:093 EDT] 00000024 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

A search for sample.ReferenceClass finds the trace entries shown in Example 10
. You can see from the trace that the class loaders tried to find the class, but it
could not be found.

Example 10   sample.ReferenceClass in the class loader trace files

```
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass > loadClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@1b501b50 Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass > loadClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@5eda5eda Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass > findClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@5eda5eda Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass > findClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@1b501b50 Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass > loadClass
name=java.lang.Throwable
this=com.ibm.ws.classloader.CompoundClassLoader@1b501b50 Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass > loadClass
name=java.lang.Throwable
this=com.ibm.ws.classloader.CompoundClassLoader@5eda5eda Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
java.lang.Throwable from parent
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
java.lang.Throwable using classloader=null
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass < loadClass Exit
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
java.lang.Throwable from parent
```

```
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
java.lang.Throwable using classloader=null
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass <  loadClass Exit
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass >  loadClass
name=javax.servlet.http.HttpServletRequest
this=com.ibm.ws.classloader.CompoundClassLoader@1b501b50 Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass >  loadClass
name=javax.servlet.http.HttpServletRequest
this=com.ibm.ws.classloader.CompoundClassLoader@5eda5eda Entry
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
javax.servlet.http.HttpServletRequest from parent
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
javax.servlet.http.HttpServletRequest using
classloader=sun.misc.Launcher$AppClassLoader@60e060e0
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass <  loadClass Exit
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
javax.servlet.http.HttpServletRequest from parent
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass 3   loaded
javax.servlet.http.HttpServletRequest using
classloader=sun.misc.Launcher$AppClassLoader@60e060e0
[12/04/07 10:56:26:093 EDT] 00000024 CompoundClass <  loadClass Exit
```

Evaluate the trace data

Using the trace data, you can determine the exception, the class throwing the exception, and the line number where the exception occurred.

► If your application makes use of the class loader API to load the missing class, then do the following:

  – Check to make sure you are using the class loader API correctly. See "Application incorrectly uses a class loader API" on page 24.

  – Make sure that the missing class is visible to the class loader that loaded the parent class. A dependent class of the referenced class must be visible to the same class loader. See "A dependent class is not visible" on page 24.

► If the referenced class is an IBM WebSphere class, see "Access to internal server classes" on page 25.

► If none of the class loaders could find the class, see "Class not available on the classpath" on page 24.

Example
Using the example trace data in Example 9 on page 21, you know the following:

► The exception occurred in the LoadClass on line 34. The source for the application contains the following code at that location:

```
34        Class.forName("sample.ReferenceClass");
```

From looking at this line of code, you know the exception occurred when the LoadClass servlet invoked the class loader API to load the sample.ReferenceClass class.

► The class loaders tried to find the class but it could not be found.

In this example, the problem occurred because the class was not on the class path. The problem was resolved by placing the class in a shared library and associating that shared library with the Web module.

ClassNotFoundException root causes

This section lists the possible root causes and possible solutions for ClassNotFoundException errors.

Class not available on the classpath

Ensure that the class is available in the classpath. If the referenced class is in a shared library, ensure that the shared library containing the referenced class is available and associated with the application.

Refer to "Using custom JAR files for your application" on page 41 for suggestions on using custom JAR files in your application.

Application incorrectly uses a class loader API

It is possible to load classes programmatically by obtaining an instance of the class loader and directly loading a class using the loadClass method or by calling Class.forName(class_name, initialize, class_loader) with that class loader. Possible issues for this may be that the name of the class is incorrect, the class is not visible on the classpath of that class loader, or the wrong class loader was engaged.

To correct this, search for the class with the Class Loader Viewer. If it is available, make sure that the correct context is used or the correct API call is used to load the class.

A dependent class is not visible

Dependent classes of the referenced class must be loaded by the same class loader. Search for the class calling the Class.forName method and use the Class Loader Viewer to determine the class loader used to load this class. Search for the dependent classes in the Class Loader Viewer and ensure they are loaded

by the same class loader as the referenced class. Refer to "Using custom JAR files for your application" on page 41 for suggestions on using custom jars in your application.

Access to internal server classes

WebSphere Application Server V6.1 gives you the ability to restrict access to internal WebSphere classes so that your applications do not make unsupported calls to WebSphere classes not published in the official WebSphere Application Server API.

This can be configured with the per-server (JVM) setting called `Access to internal server classes`, found by navigating to **Application Server** → **Server1** in the administrative console.

If this setting is set to `Restrict` and your application makes calls to the WebSphere internal API you can encounter a ClassNotFoundException. Change the setting to `Allow` to correct this. It is, however, recommended that you restrict access to the internal WebSphere API because in the future this restriction may become permanent.

> **Where to go from here:** If these suggestions do not resolve your problem, go to "The next step" on page 54 for information about performing online searches of known class loader issues. This section also contains information on what you will need to collect in order to engage IBM support.

NoClassDefFoundError

This section takes you through the process of diagnosing and resolving NoClassDefFoundError problems.

Analyze the trace

First, find the NoClassDefFoundError in the trace and note the following:

► The name of the missing referenced class

► The line in the application where the error occurred

Next, search for entries relevant to the class in the trace and note what classloaders were used to search for this class.

Example

In the trace in Example 11 you can see the missing referenced class is
sample.ReferenceClass. The error occurred on line 7 in the application.

Example 11   Example NoClassDefFoundError

```
[12/04/07 10:49:19:687 EDT] 00000024 SystemErr     R
java.lang.NoClassDefFoundError: sample.ReferenceClass
[12/04/07 10:49:19:687 EDT] 00000024 SystemErr     R   at
servlet.InvokeAction.performAction(InvokeAction.java:7)
[12/04/07 10:49:19:687 EDT] 00000024 SystemErr     R   at
servlet.LoadClass.doPost(LoadClass.java:45)
[12/04/07 10:49:19:687 EDT] 00000024 SystemErr     R   at
servlet.LoadClass.doGet(LoadClass.java:33)
[12/04/07 10:49:19:687 EDT] 00000024 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:743)
[12/04/07 10:49:19:687 EDT] 00000024 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
```

The entries for sample.ReferenceClass are shown in Example 12 . You can see
that the class loaders tried to find the class, but it could not be found.

Example 12   Trace entries for sample.ReferenceClass

```
[12/04/07 10:49:19:687 EDT] 00000024 CompoundClass >  loadClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@43484348 Entry
[12/04/07 10:49:19:687 EDT] 00000024 CompoundClass >  loadClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@60026002 Entry
[12/04/07 10:49:19:687 EDT] 00000024 CompoundClass >  findClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@60026002 Entry
[12/04/07 10:49:19:687 EDT] 00000024 CompoundClass >  findClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@43484348 Entry
[12/04/07 10:49:19:687 EDT] 00000024 SystemErr     R
java.lang.NoClassDefFoundError: sample.ReferenceClass
```

Evaluate the trace data

Search for the missing class file on the file system. If you can find it, note the
location and the name of the JAR file the class resides in.

Use the Class Loader Viewer (table mode) to determine if the JAR file or the location exists on the class path of the class loaders in WebSphere.

- ▶ If it is on the class path, check the class to ensure it can be loaded. See "Class cannot load" on page 27 for possible solutions.

- ▶ If it is not on the class path, ensure that the missing class is on the class path and available to the class loader to load. See "Class not available on the classpath" on page 27.

Example

From the example trace, you know the following:

- ▶ The NoClassdefFoundError occurs because the class loader cannot locate the class sample.ReferenceClass.

- ▶ The error occurs at line 7 of the InvokeAction.

In the source code for the InvokeAction class, you can see line 7 creates an instance of the class ReferenceClass.

```
7        ReferenceClass root = new ReferenceClass("Value returned");
```

However the class loader in WebSphere cannot locate the ReferenceClass, so it throws the NoClassDefFoundException.

In this example, the problem occurred because the class was not on the class path. The problem was resolved by placing the class in a shared library and associating that shared library with the Web module.

NoClassDefFoundError root causes

This section lists the possible root causes and possible solutions for a NoClassDefFoundError conditions.

Class not available on the classpath

Ensure that the library is available in the classpath. If the referenced class is in a shared library, check that the shared library containing the referenced class is available and associated with the application.

Refer to "Using custom JAR files for your application" on page 41 for suggestions on using custom jars in your application.

Class cannot load

For various reasons the class could not be loaded. Possible reasons include the following:

- ► Failure to load the dependent class
- ► The dependent class has a bad format
- ► The version number of a class

Ensure these are correct.

See "Using custom JAR files for your application" on page 41 for more information.

> **Where to go from here:** If these suggestions do not resolve your problem, go to "The next step" on page 54 for information about performing online searches of known class loader issues. This section also contains information on what you will need to collect in order to engage IBM support.

NoSuchMethodError and IllegalArgumentException

This section takes you through the process of diagnosing and resolving NoSuchMethodErrors and IllegalArgumentException errors.

Analyze the trace

A NoSuchMethodError looks like Example 13 on page 29.

Find the error, and note the following:

- ► The class the exception is referencing
- ► The missing method and parameter
- ► The line in the application where the error occurred

Search for the class in the trace files, and note the following:

- ► The class loader loading this class
- ► The location of the class file

Example

In Example 13 on page 29 the class `sample.ReferenceClass` is missing the method `getText` with parameter `java.lang.String`. The error occurs on line 8 in the application.

Example 13   Example NoSuchMethodError

```
[12/04/07 11:00:42:437 EDT] 00000024 SystemErr     R
java.lang.NoSuchMethodError:
sample/ReferenceClass.getText()Ljava/lang/String;
[12/04/07 11:00:42:437 EDT] 00000024 SystemErr     R   at
servlet.InvokeAction.performAction(InvokeAction.java:8)
[12/04/07 11:00:42:437 EDT] 00000024 SystemErr     R   at
servlet.LoadClass.doPost(LoadClass.java:45)
[12/04/07 11:00:42:437 EDT] 00000024 SystemErr     R   at
servlet.LoadClass.doGet(LoadClass.java:33)
[12/04/07 11:00:42:437 EDT] 00000024 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:743)
[12/04/07 11:00:42:437 EDT] 00000024 SystemErr     R   at
javax.servlet.http.HttpServlet.service(HttpServlet.java:856)
[12/04/07 11:00:42:437 EDT] 00000024 SystemErr     R   at
com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.j
ava:966)
```

An IllegalArgumentException has a similar format. It will reference a class and a particular message related to the exception. An IllegalArgumentException may complain of the incorrect argument being passed to a method.

In Example 14 you can see that sample.ReferenceClass is loaded by the class loader `SinglePathClassProvider : com.ibm.ws.Class loader.SinglePathClassProvider@15761576` and is located in the `C:\MissingMethodUtilityJar.jar` file.

Example 14   Trace entries showing the class being loaded

```
[12/04/07 11:00:42:421 EDT] 00000024 CompoundClass >  loadClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@22462246 Entry
[12/04/07 11:00:42:421 EDT] 00000024 CompoundClass >  loadClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@48c248c2 Entry
[12/04/07 11:00:42:421 EDT] 00000024 CompoundClass >  findClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@48c248c2 Entry
[12/04/07 11:00:42:421 EDT] 00000024 CompoundClass >  findClass
name=sample.ReferenceClass
this=com.ibm.ws.classloader.CompoundClassLoader@22462246 Entry
[12/04/07 11:00:42:421 EDT] 00000024 ReloadableCla 3   adding path to
reload cache
                                c:\MissingMethodUtilityJar.jar
```

```
[12/04/07 11:00:42:437 EDT] 00000024 CompoundClass 3   class
sample.ReferenceClass found in SinglePathClassProvider :
com.ibm.ws.classloader.SinglePathClassProvider@15761576 classpath =
c:\MissingMethodUtilityJar.jar
[12/04/07 11:00:42:437 EDT] 00000024 CompoundClass <  findClass Exit
[12/04/07 11:00:42:437 EDT] 00000024 CompoundClass 3   loaded
sample.ReferenceClass from this=
com.ibm.ws.classloader.CompoundClassLoader@22462246
    Local ClassPath:
C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps\jasperch
uiNode02Cell\ClassloaderSampleEAR.ear\NoSuchMethodErrorWeb.war\WEB-INF\
classes;C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps\
jasperchuiNode02Cell\ClassloaderSampleEAR.ear\NoSuchMethodErrorWeb.war;
C:\MissingMethodUtilityJar.jar
    Delegation Mode: PARENT_FIRST
[12/04/07 11:00:42:437 EDT] 00000024 CompoundClass 3   loaded
sample.ReferenceClass using classloader=
com.ibm.ws.classloader.CompoundClassLoader@22462246
    Local ClassPath:
C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps\jasperch
uiNode02Cell\ClassloaderSampleEAR.ear\NoSuchMethodErrorWeb.war\WEB-INF\
classes;C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\installedApps\
jasperchuiNode02Cell\ClassloaderSampleEAR.ear\NoSuchMethodErrorWeb.war;
C:\MissingMethodUtilityJar.jar
    Delegation Mode: PARENT_FIRST
[12/04/07 11:00:42:437 EDT] 00000024 CompoundClass <  loadClass Exit
```

Evaluate the trace data

Using the information gathered in the trace, determine if the class loaded by the class loader contains the expected version of the class.

Use the Class Loader Viewer to determine the location of the class referenced and to make sure that the class loaded is the correct one.

If the class loaded by the class loader in WebSphere does not contain the method or the argument referenced by the source code, see "Different version of classes" on page 31.

Example
From the trace entries in the example, you know the following:

► The class sample.ReferenceClass is missing the getText method with parameter java.lang.String.

- ► The class sample.ReferenceClass is located in the following file, C:\MissingMethodUtilityJar.jar.
- ► The method is being called from the InvokeAction class on line 8 and is invoking the method sample/ReferenceClass.getText(String text).

This tells you that `MissingMethodUtilityJar.jar` contains the ReferenceClass class file that does not have the method getText(String). The problem was resolved by placing the correct version of the class with the correct method in a shared library and associating that shared library with the Web module.

NoSuchMethodError, IllegalArgumentException root causes

This section lists the possible root causes and possible solutions for a NoSuchMethodError or IllegalArgumentException conditions.

Different version of classes

An incorrect version of the class is being referenced.

It may be possible that there are multiple versions of the class or JAR files available on the class path of the class loaders in WebSphere. If they are not needed for other applications deployed to the server you can remove the incorrect versions of the class. If they are needed, use the class loader delegation mode to ensure the correct version is loaded. See "Class loading/delegation mode" on page 10 for more information.

If the class is in a shared library, ensure that the correct files are associated with the shared library and with the application.

Refer to "Using custom JAR files for your application" on page 41 for suggestions on using custom jars in your application.

Example scenario

For example if there are two copies of the referenced class file, one loaded by the WebSphere extensions classloader and one available to the application classloader, ensure that the class loader delegation mode of `Parent Last` is set to pick up the class within your application instead of the one on the server.

Where to go from here: If these suggestions do not resolve your problem, go to "The next step" on page 54 for information about performing online searches of known class loader issues. This section also contains information on what you will need to collect in order to engage IBM support.

UnsatisfiedLinkError

This section takes you through the process of diagnosing and resolving UnsatisfiedLinkError problems.

Analyze the trace

An UnsatisfiedLinkError message will resemble Example 15 . The error message of the UnsatisfiedLinkError will have the name of the missing native library.

► Note the native library referenced.

Example 15   Example UnsatisfiedLinkError message.

```
java.lang.UnsatisfiedLinkError mynativelibrary.so
```

In the example, the library is `mynativelibrary`.

Evaluate the trace data

Other than the library name, the trace does not provide much information. There are several possibilities for the root cause of the problem, but you need to check each possibility until you find your problem.

The possible root causes for this problem are as follows:

► The native library is not referenced properly. See "Configuration error" on page 33.

► The library is not visible to the application. See "Library is not visible" on page 33 for suggestions on how to make it visible.

► The native library is loaded more than once. See "Native library is already loaded" on page 33.

► The dependent native libraries for the referenced native library are not successfully loaded. See "Dependent native library configuration" on page 35.

► There are problems with the JVM itself in loading the native libraries. The method System.mapLibraryName is used to return the proper extension for referencing a native library file in the various operating systems.

See "System.mapLibraryName returns the wrong library file" on page 36 for possible actions to resolve this.

UnsatisfiedLinkError root causes

This section lists the possible root causes and possible solutions for an UnsatisfiedLinkError condition.

Configuration error

A configuration error is usually one of the following:

- ▶ Incorrect library extension:
  - – Windows®: A library has the dynamic link library name library_name.dll.
  - – AIX®, HP-UX, Solaris™, Linux®: A library has the name library_name.so or library_name.a.
- ▶ System.loadLibrary is passed an incorrect parameter:
  - – Windows: To load Name.dll, Name is passed to the loadLibrary method.
  - – AIX, HP-UX, Solaris, Linux: To load libName.so or libName.a, libName is passed to the loadLibrary method.

Correct the native library reference in your application or the shared library reference to resolve this.

Library is not visible

Ensure that the native library is available on the Java library path (java.library.path). During startup of the WebSphere Application Server, the SystemOut log prints the Java library path (java.library.path) variable and lists the directories and libraries that it will load. Check this to make sure your native library is there.

You can also configure a shared library for the native library and associate it with the application. Specify the path to the native library in the Native Library Path field and the class or JAR file referencing the native library in the Classpath field.

Refer to "Using custom JAR files for your application" on page 41 for suggestions on using custom JAR files in your application.

Native library is already loaded

If the native library was already loaded by an application and the same application tries to load it again, this can cause this error. One possible scenario for this to occur is if the application is restarted and it tries to load the native library again. WebSphere has no control over the native code and if it does not unload and load properly the application may fail.

To determine if the application was restarted. Check the SystemOut log for the following sequence of logs. See Example 16 for sample log output.

Example 16   Restarting an application

```
[03/05/07 13:47:50:781 EDT] 0000002f ApplicationMg A  WSVR0220I:
Application stopped: ClassloaderSampleEAR
[03/05/07 13:47:57:031 EDT] 00000030 ApplicationMg A  WSVR0200I:
Starting application: ClassloaderSampleEAR
[03/05/07 13:47:57:046 EDT] 00000030 ApplicationMg A  WSVR0204I:
Application: ClassloaderSampleEAR  Application build level: Unknown
[03/05/07 13:47:58:078 EDT] 00000030 WebGroup      A  SRVE0169I:
Loading Web Module: ClassNotFoundExceptionWeb.
[03/05/07 13:47:58:140 EDT] 00000030 VirtualHost   I  SRVE0250I: Web
Module ClassNotFoundExceptionWeb has been bound to
default_host[*:9081,*:80,*:9444,*:5063,*:5062,*:443].
[03/05/07 13:47:58:218 EDT] 00000030 WebGroup      A  SRVE0169I:
Loading Web Module: ClassCastExceptionWeb.
[03/05/07 13:47:58:343 EDT] 00000030 VirtualHost   I  SRVE0250I: Web
Module ClassCastExceptionWeb has been bound to
default_host[*:9081,*:80,*:9444,*:5063,*:5062,*:443].
[03/05/07 13:47:58:390 EDT] 00000030 WebGroup      A  SRVE0169I:
Loading Web Module: NoSuchMethodErrorWeb.
[03/05/07 13:47:58:468 EDT] 00000030 VirtualHost   I  SRVE0250I: Web
Module NoSuchMethodErrorWeb has been bound to
default_host[*:9081,*:80,*:9444,*:5063,*:5062,*:443].
[03/05/07 13:47:58:546 EDT] 00000030 WebGroup      A  SRVE0169I:
Loading Web Module: NoClassDefFoundErrorWeb.
[03/05/07 13:47:58:640 EDT] 00000030 VirtualHost   I  SRVE0250I: Web
Module NoClassDefFoundErrorWeb has been bound to
default_host[*:9081,*:80,*:9444,*:5063,*:5062,*:443].
[03/05/07 13:47:58:687 EDT] 00000030 ApplicationMg A  WSVR0221I:
Application started: ClassloaderSampleEAR
```

To resolve this, ensure that the library is only loaded once. To achieve this you can package a class containing a static call to this native library in a utility JAR file and package it in a shared library to be associated with a server class loader. This ensures that the native library is loaded only once for each Java virtual machine regardless of the application life cycle.

Example 17   Sample native library loader

```
Sample Class:
public class LibLoader {
   static {System.loadLibrary(MyNativeLib);}
```

```
    public LibLoader();
}
```

After you create a native library loader, you can create and configure a shared library for this class and your native library. Specify the JAR file containing the native library loader in the Classpath field, and specify the path to your native library in the Native Library Path field.

Now create a class loader with the server and associate the shared library to this class loader.

1. In the administrative console, navigate to **Servers** → **Application servers** → *server_name* → **Server Infrastructure/Java and Process Management** select **Class loader**.

2. Click **New**, and select the class loader order from the drop-down according to the needs of your application: `Classes loaded with parent class loader first` or `Classes loaded with application class loader first`.

3. Refer to "Class loading/delegation mode" on page 10 for more information about these options.

4. Under Additional Properties, select **Shared library references**, and create a new reference.

5. Select the shared library from the drop down, and click **OK**.

6. Save the server configuration.

Modify your application code to use the native library loader to load your native libraries.

Dependent native library configuration

Dependent native libraries of the native library must be loaded by the JVM class loader. Therefore any dependent native libraries must be on the Java library path (LIBPATH). This is because when the JVM loads the native library it can only call the JVM class loader to resolve the dependency and the JVM class loader can only reference the Java library path to find referenced dependent native libraries.

To determine if your native library is on the Java library path, you can review the SystemOut log from the WebSphere Application Server. During startup it prints the Java library path (java.library.path) and lists the directories and libraries that it will load.

If it is not available, you can append the path to the native library to the platform-specific native library environment variable or to the java.library.path system property of the server process definition.

The native library environment variable is as follows:

► Windows: PATH

► Linux: LD_LIBRARY_PATH

► AIX: LIBPATH

► HP-UX: SHLIB_PATH

► Solaris: LD_LIBRARY_PATH

Use the following steps to set the java.library.path JVM system property:

1. Navigate to **Server → Application server → *server_name* → Java and Process Management → Process Definition → Java Virtual Machine → Custom Properties**.

2. Create a new custom property.

3. Specify the java.library.path as the name and the path to your native library as the value.

4. Save your server configuration.

System.mapLibraryName returns the wrong library file

When loading a shared library, the JVM calls mapLibraryName(libName) to convert libName to a platform specific name. This method may return a file name with an improper extension. For example, in a UNIX® environment it may return libName.so rather than libName.a.

Write a program to call System.mapLibraryName() to verify that it returns the correct value. If it does not return the correct value engage IBM Support.

Where to go from here: If these suggestions do not resolve your problem, go to "The next step" on page 54 for information on performing online searches of known class loader issues. This section also contains information on what you will need to collect in order to engage IBM support.

VerifyError

This section takes you through the process of diagnosing and resolving VerifyError problems.

Analyze the trace

Search for the VerifyError in the trace and note the following:

► The class

► The method

► The error message

Example
A VerifyError will resemble Example 18 . In this example, the class is `MyClass`, the method is `MyMethod` and the error message is `incompatible object argument for method call`.

Example 18   Example VerifyError.

```
java.lang.VerifyError: (class: MyClass, method: MyMethod signature:
(I)V) incompatible object argument for method call
```

Collect Class Loader Viewer information
Use the Class Loader Viewer to determine how the class is loaded.

VerifyError root causes

VerifyError occurs when the byte code verification process, during class loading, fails due to internal inconsistency or security problems. The error message associated with the VerifyError will help you identify the class and the cause of the error.

Different version of class
A VerifyError is usually the result of loading an incorrect version of the class or library. The information collected from the Class Loader Viewer can help determine the location of the suspect class or library referenced.

If the class is in a shared library, ensure that the correct files are associated with the shared library and with the application.

Refer to "Using custom JAR files for your application" on page 41 for suggestions on using custom jars in your application.

Example scenario

For example if there are two copies of the referenced class file, one loaded by the WebSphere extensions classloader and one available to the application classloader, ensure that the class loader delegation mode of **Class loaded with application class loader first** picks up the class within your application instead of the one on the server.

Where to go from here: If these suggestions do not help, collect the class loader mustgather and engage IBM support. Refer to "Contact IBM" on page 55 for instructions.

Configuring shared libraries

Following are instructions to configure a shared library and to associate it with an application.

Create the shared library

To create a shared library, do the following:

1. In the administrative console, navigate to **Environment** → **Shared Libraries**.

2. Select the scope of the shared library. You have the option to choose All, Cell, Node, and Node/Server. This allows the shared library to be visible to the applications and servers residing in these scopes.

3. Click **New**, and specify the name of the shared library, the classpath, or the native classpath to the location of the utility JAR file or utility library. See Figure 6 on page 39 for an example.

Figure 6   Configuring shared library options

Associate the shared library with an application

To associate the shared library with an application, do the following:

1. In the administrative console, navigate to **Enterprise Application** →
   Enterprise Applications.

2. Click the application name to open the configuration view.

3. Select **Shared library references** under the References section.

   You will see the option to configure a shared library and associate it with a
   particular application or a Web module.

4. Enable the check box located beside the application or the Web module, and
   click **Reference shared libraries**.

5. Select the shared library you created, and click the arrow to associate it with
   this Web module.

   See Figure 7 on page 40 for an example**.**
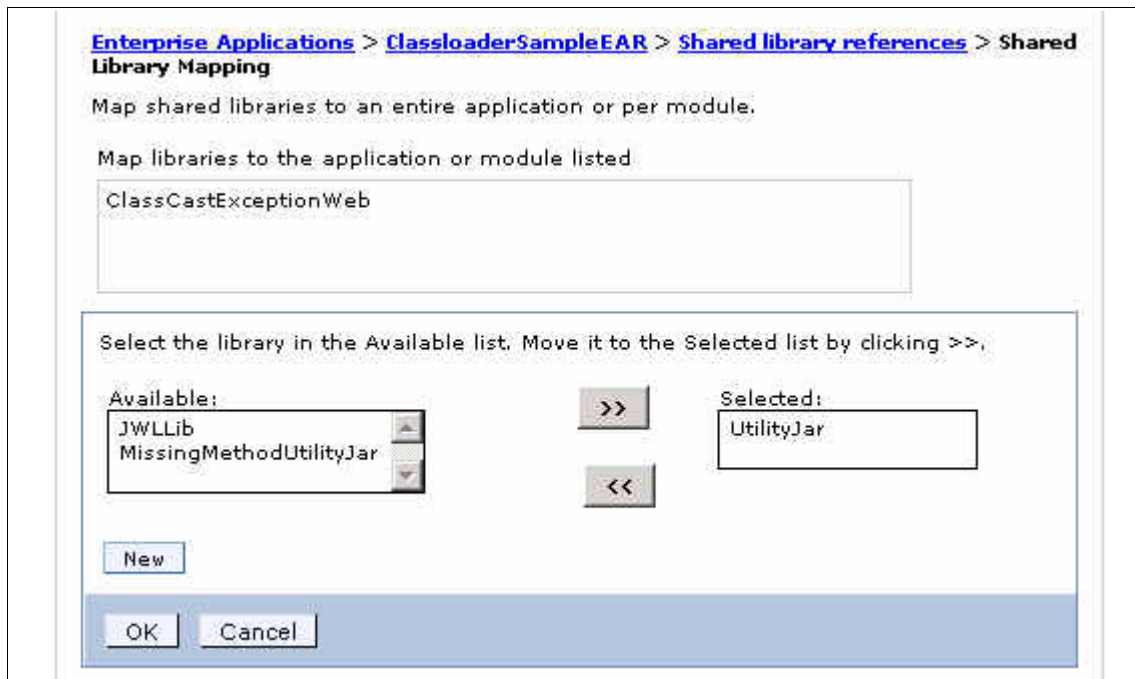
Figure 7   Associating shared library with a Web module of an application

6. Click **OK**.

7. Click **OK** again to complete the action.

8. Save the configuration.

Using custom JAR files for your application

Class loader exceptions often occur due to the incorrect location of a common utility class or utility JAR file or the configuration of the server that uses it. Because every application is unique and its requirements are different, it is not simple to determine where best to place a utility file.

This section discusses some possible scenarios. It will help you determine the best place for your utility file and how to configure the WebSphere Application Server to use it.

First, you should have an understanding of what is considered a utility file and where *not* to place these files.

Requirements for a utility file

The basic requirement for a class or JAR file to be a utility file is that *it cannot depend on any application that may be using it*. With this caveat, it is easier to maintain and to allow modifications to the utility files without having to modify the application itself. In addition this condition allows for utility files to be overridden and allows for multiple versions of the utility file to exist in your environment. A utility file can depend on other utility files, but this causes additional complications to the administration of your environment.

Where you should not place utility files

In deciding where best to place your utility files, it is important to recognize that these files should *not* be included in the WebSphere Application Server's environment.

For example: *app_server_root*/lib, *app_server_root*/lib,/ext* , *app_server_root*/java (including any subdirectories), or the JVM classpath.

Adding utility files to those directories can cause problems with the WebSphere runtime environment and can cause unexpected results, including the overwriting of WebSphere classes that can be detrimental to the overall functionality of the server.

Class loader policy options

Now that you understand where not to place these files you can look at options for where you can place them. Start by examining the class loader policy options in WebSphere Application Server V6.1.

Server class loader policy settings

To configure the server-specific application settings, navigate to **Application Server → *server_name*** (Figure 8).

By default the WebSphere server class loader policy is set to `Multiple`, meaning each application will receive its own class loader for loading EJBs, utility JARs, and shared libraries. The alternative is to set the policy to `Single`, meaning there is a single class loader for all the applications on the server.

You also set the class loading mode here, selecting either `Parent first` to load parent classloader files first or `Parent last` to load application classloader files first.

Figure 8   Configuring Server-specific Application Setting

Application class loader settings

To configure the application class loader, navigate to **Enterprise Application** → *application_name* (Figure 9). You can update both the class loader order and the WAR class loader policy.

The default configuration for the application class loader is `Class loaded with parent class loader first.` The alternative option is `Class loaded with application class loader first.` These options are similar to server class loading mode options.

In addition, you have the option to select the WAR class loader policy. You can choose to have a class loader for each WAR file in the application or to have one class loader for the entire application.

Figure 9   Configuring Application class loader

Web module class loader settings

You can also set the class loader policy at the WAR file. The options are to load classes from the parent class loader first or last.

To configure the WAR class loader, navigate to **Enterprise Application** → *application_name* → **Manage Modules** → *Web_module* → **Class loader order**. (Figure 10).

Figure 10   Configuring Web module class loader

Refer to the following article in the WebSphere Information Center for more information about class loader options:

► *Class loaders*

   http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.
   websphere.nd.doc/info/ae/ae/crun_classload.html

Where you should place utility files

Following is a decision tree to help you identify how best to use common application files.

Will you need to deploy different versions of this utility file?

1. **If no**, will this utility file be shared by more than one enterprise application?

    a. **If yes**, will visibility to this utility file cause any other applications installed to the same server to malfunction?

        i. **If yes**, configure a shared library in the server scope, and associate it with the application.

        ii. **If no**, configure a shared library at the server, node, or cell level scope, depending on where the applications using these shared libraries are located, and associate it with all the applications that will use it.

            For example, if you have two applications using the shared library, in a cell on two different servers, create the shared library at the cell level scope so that the shared library is visible to those applications.

    b. **If no**, will multiple modules within the application need this utility file?

        i. **If yes**, place the utility file in the root directory of the EAR, and modify the manifest.mf file to reference the utility JAR.

            You can modify the `manifest.mf` file using the Application Server Toolkit, Rational® Application Developer or by modifying the file directly. We recommend that you use an editor.

            To use an editor, import the EAR into the work space and import the utility file to the root of the EAR. From the context of the reference module (Web or EJB), select **Properties** → **J2EE Module Dependencies** → **J2EE Modules** and check the UtilityJar.jar option.

            To modify the manifest.mf manually, open the *reference_module*\META-INF\manifest.mf in a text editor, and add the reference JAR file to the classpath.

            Refer to the following Application Server Toolkit Information Center article for additional information:

            Specify dependent JAR files or modules at the following Web site

            http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.js
            p?topic=/com.ibm.etools.j2ee.doc/topics/tjdepend.html

            Example 19 shows a sample manifest.mf file. In this example, UtilityJar.jar is located in the root directory of the EAR project, and UtilityJar2.jar is located in the folder *mypath* under the root directory of the EAR.

Example 19   Example manifest.mf to add UtilityJar.jar to the application classpath.

```
Manifest-Version: 1.0
Class-Path: UtilityJar.jar mypath/UtilityJar2.jar
```

ii. **If no**, place the utility JAR file in the following location:

EJB module: *EAR_file/EJB_jar*/

WAR module: *EAR_file/WAR_file*/WEB-INF/lib/

2. **If yes**, configure a shared library for each version of the utility JAR file, and associate them with the application or Web module.

After you place the utility file in the correct location, you can alter the class loader settings to use a different version of a utility file that is already included with WebSphere Application Server. For example, if you want to use a newer version of a library located in your application, you can configure the class loader policy to `Class loaded with application class loader first` so that it picks up the utility files in your application first rather than the one on the server.

Class loader options also provide flexibility in your ability to override existing libraries included with WebSphere.

Tip: When using your own version of Xerces and Xalan, you can also use the `Parent Last` class loader policy to load the version included with your application instead of the one included with the SDK.

Class loaders for custom logging: It is common for users to want to use their own logging mechanism for their applications. Refer to the following documentation in the WebSphere Information Center for more information on configuring the class loaders to use custom logging.

► Jakarta Commons Logging

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/ctrb_classload_jcl.html`

► Configurations for the WebSphere Application Server logger

  `http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.base.doc/info/aes/ae/ctrb_classload_jcl_conf.html`

Collecting diagnostic data

This section discusses how to collect traces relevant to class loader problems. It also includes instructions for using the Class Loader Viewer.

Class loader-specific traces

This section tells you how to collect traces for class loader problems.

Class loader trace

Use the following steps to collect a class loader trace for WebSphere Application server Version 6.1:

1. Log on to the administrative console.

2. In the left navigational panel, expand **Troubleshooting**. Click **Logs and Trace**.

3. Select the application server to be traced, and then on the next page click the **Diagnostic Trace** link.

4. Select the **Configuration tab**.

5. Select the Enable Log property.

6. Under **Trace Output**, select the **File** radio button, and specify the file name. Also Increase the Maximum file size to 100 MB, and increase the Maximum number of historical files to 10.

> **Note:** The name and location of the output file, by default, is ${SERVER_LOG_ROOT}/trace.log. It outputs to *profile_home*\logs*server_name*\trace.log.

7. Select **Basic (Compatible) Trace Output Format**.

8. Navigate to **Logging and Tracing** > *application_server* > **Change Log Detail Levels**.

9. Under the **Configuration tab**, specify the trace string suggested.

> **Tip:** For class loaders the tracestring is `com.ibm.ws.classloader.*=all`.

10. Click **Apply** and **OK**. Save your configuration. Select the **Synchronize changes with Nodes** option if you are using a distributed environment.

JVM class loader and bootstrap traces

To enable the Java Virtual Machine (JVM) class loader and bootstrap traces for Version 6.1, do the following:

1. From the administrative console, select **Servers > Application servers**, and select the problem server.

2. On the right side, under Server Infrastructure, expand **Java and Process Management**.

3. Click **Process Definition**.

4. Under **Additional Properties**, select **Java Virtual Machine**.

5. Enable the following Properties by selecting the check boxes located on the left of the property.

   – `Verbose class loading`
   – `Verbose JNI`

6. Click **Apply**.

7. On right side under **Additional Properties**, click **Custom Properties**.

8. Click **New**.

9. Enter the following custom properties:

   – Name: `ws.ext.debug`
   – Value: `true`

10. Click **Apply** and **OK**. Save your configuration. Select the **Synchronize changes with Nodes** option if you are using a distributed environment.

Using the Class Loader Viewer

The Class Loader Viewer in the administrative console can be used to determine how a class is loaded by the class loaders in the WebSphere Application Server.

Enable the Class Loader Viewer service:

If the Class Loader Viewer service is not enabled, the Class Loader Viewer only displays the hierarchy of class loaders and their classpaths, but not the classes actually loaded by each of the class loaders. This also means that the search capability of the Class Loader Viewer is lost.

To enable the Class Loader Viewer Service, select **Servers** → **Application Servers** → *server_name*, and then click the **Class Loader Viewer Service** under the **Additional Properties** link. Next, select **Enable service at server startup**. Restart the application server for the setting to take effect.

From the administrative console, navigate to **Troubleshooting** → **Class Loader Viewe**r, and expand the <server_name> tree until you see the application and its modules.

With the Class Loader Viewer you have two options to view the classes associated with the Web module, the default tree view (Figure 11 on page 50) or the Table View (Figure 12 on page 51). In both options the viewer lists the class loader class, the JAR files on each of the class loader's classpath, if the Class Loader Viewer Service is enabled, and the classes loaded by the class loaders.

As an example, search for the classes bean.Leaf and bean.Root used to illustrate the ClassCastException error.

From the administrative console, navigate to **Troubleshooting** → **Class Loader Viewer**, and expand the topology tree until you see the ClassloaderSampleEAR and its associated Web modules. Next, select the ClassCastExceptionWeb.war.

Figure 11   Default Tree mode for the Class Loader Viewer

Figure 12   Table mode for the Class Loader Viewer

To use the search function, ensure that you enabled the Class Loader Viewer service, and select the **Search** option. This allows you to search for loaded classes.

> **Tip:** Ensure that the class is loaded by invoking the application that will reference the class or else the Search option will not work.

Figure 13 on page 52 shows an example using `*Leaf` as the search parameter. The Class Loader Viewer finds classes with this pattern if they are loaded.
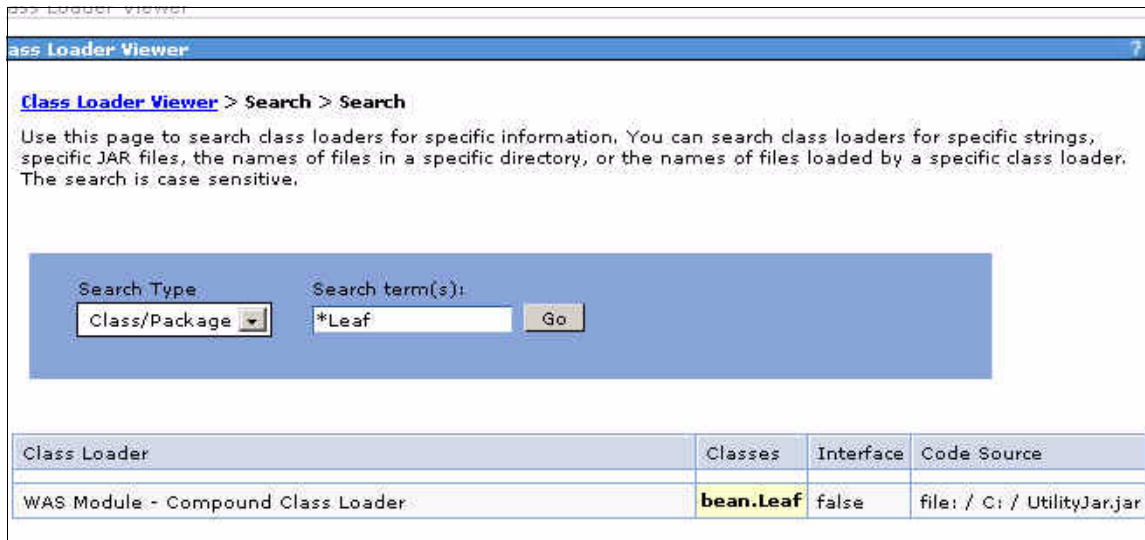
Figure 13   Using the search functionality in Class Loader Viewer

Additional class loader diagnostics

JVM Version 5.0 provides some additional options that can be useful when troubleshooting class loading problems. These options are set as command line arguments for the JVM.

Select **Servers** → **Application Servers** → *server_name* and then expand the **Java and process management** section under the Server Infrastructure heading. Click the **Process Definition** link and then the **Java Virtual Machine** link. Enter the following options in the **Generic JVM arguments** field:

► -verbose:dynload

This option provides detailed information about classes being loaded. Information includes the class name and package, the JAR file name if the class is packaged in a JAR file, the size of the class, and the time it takes to load the class. The information is written to the native_stderr.log file. An example output looks similar to the following:

```
<Loaded servlet/LoadClass>
<  Class size 1508; ROM size 1800; debug size 0>
<  Read time 0 usec; Load time 40 usec; Translate time 32 usec>
```

► -Dibm.cl.verbose=*name*

This option allows you to trace the way the class loaders find and load a given class. The name is the full name of the class, including the package name. By

specifying -Dibm.cl.verbose=servlet.LoadClass, the following output is printed to the SystemOut log file when the LoadClass class is loaded.

Example 20   Output printed to SystemOutlogfile when LoadClass is loaded

```
[23/04/07 15:25:19:984 EDT] 00000024 SystemOut     O ExtClassLoader
attempting to find servlet.LoadClass
[23/04/07 15:25:19:984 EDT] 00000024 SystemOut     O ExtClassLoader
using classpath
C:\IBM\WebSphere\AppServer_v61\java\jre\lib\ext\CmpCrmf.jar;...
[23/04/07 15:25:19:984 EDT] 00000024 SystemOut     O ExtClassLoader
could not find servlet/LoadClass.class in
C:\IBM\WebSphere\AppServer_v61\java\jre\lib\ext\CmpCrmf.jar
...
...
[23/04/07 15:25:19:984 EDT] 00000024 SystemOut     O ExtClassLoader
could not find servlet.LoadClass

[23/04/07 15:25:19:984 EDT] 00000024 SystemOut     O AppClassLoader
attempting to find servlet.LoadClass
[23/04/07 15:25:19:984 EDT] 00000024 SystemOut     O AppClassLoader
using classpath
C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\properties;...
[23/04/07 15:25:20:000 EDT] 00000024 SystemOut     O AppClassLoader
could not find servlet/LoadClass.class in
C:\IBM\WebSphere\AppServer_v61\profiles\AppSrv01\properties
...
...
[23/04/07 15:25:20:000 EDT] 00000024 SystemOut     O AppClassLoader
could not find servlet.LoadClass

[23/04/07 15:25:20:000 EDT] 00000024 SystemOut     O
com.ibm.ws.bootstrap.ExtClassLoader attempting to find
servlet.LoadClass
[23/04/07 15:25:20:000 EDT] 00000024 SystemOut     O
com.ibm.ws.bootstrap.ExtClassLoader using classpath
C:\IBM\WebSphere\AppServer_v61\java\lib;....
...
...
[23/04/07 15:25:20:015 EDT] 00000024 SystemOut     O
com.ibm.ws.bootstrap.ExtClassLoader could not find
servlet/LoadClass.class in C:\IBM\WebSphere\AppServer_v61\java\lib
...
...
```

Note that the output in Example 20 on page 53 was truncated to fit.

The next step

The symptoms and problem areas included in this activity are some that you are more likely to experience. However, there are other class loader-related problems that you can experience.

Search online support

If you are sure the problem is with the class loader, there are things that you can do before contacting IBM support. First, you should review the documentation that you gathered for errors that were not addressed in this paper, and search support sites for information or fixes. Look for current information available from IBM support on known issues and resolutions on the following IBM support page:

http://www-1.ibm.com/support/search.wss?rs=180&tc=SSEQTP&tc1=SSCVS24

Look also at the WebSphere Information Center documentation for additional resources for diagnosing and fixing class loader issues:

▶ *Troubleshooting Class loaders*

  – Network Deployment on distributed platforms:

    http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.nd.doc/info/ae/ae/ttrb_classload_viewer.html

  – Network Deployment on i5/OS®

    http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.nd.iseries.doc/info/iseriesnd/ae/ttrb_classload_viewer.html

On-line resources

If, after going through this process, you still have an undiagnosed problem, we recommend that you return to *Approach to Problem Determination in WebSphere Application Server V6* at the following Web location:

http://www.redbooks.ibm.com/redpapers/pdfs/redp4073.pdf

Review the problem classifications to see if there are any other components that might be causing the problem.

You can also review the troubleshooting process for class loaders available at the following Web location:

http://www-1.ibm.com/support/docview.wss?rs=180&uid=swg21242692

For more information about class loaders please refer to Chapter 12. Understanding class loaders in the IBM Redbooks® publication *WebSphere Application Server V6.1: System Management and Configuration* at the following Web site:

http://www.redbooks.ibm.com/redbooks/pdfs/sg247304.pdf

Contact IBM

If these steps do not resolve your problem, then gather additional information as specified in the following MustGather document and raise a problem record with IBM. The following Web site contains a list of the MustGather documentation for Classloaders.

http://www-1.ibm.com/support/docview.wss?uid=swg21196187

Installing and configuring the sample application

The sample used in this paper is available for download from the following Web address:

ftp://www.redbooks.ibm.com/redbooks/REDP4307/

This is a sample class loader application that demonstrates some of the common class loader problems. Use the following instructions to install the application and to duplicate the exceptions.

The ClassloaderSample.zip will contain the following three files.

► ClassloaderSampleEAR.ear

  The enterprise application to be deployed to WebSphere Application Server V6.1.

► UtilityJar.jar

  Contains the utility classes for the ClassCastExceptionWeb module.

► MissingMethodUtilityJar.jar

  Contains the utility classes with a missing method for the NoSuchMethodError module.

Install the application

To install the application, do the following:

1. Unzip the zip file to a directory on your local system. For example C:\ for Windows.

2. Start the application server.

3. In the administrative console, expand **Enterprise Application** → **Install New Application**. Specify the location of the EAR as in Figure 14.

Figure 14   Installing the sample class loader application

4. Step through the rest of the wizard, taking the default settings.

5. Click **Finish** to complete the install and to save the configuration. Figure 15 on page 57 shows the results of the installation.

Figure 15   Save the server configuration

Create the shared libraries

Create two shared libraries using the following steps.

1. In the administrative console, navigate to **Environment** → **Shared Libraries**.

2. Select **New** and specify the following:

- Name: `UtilityJar`
- Classpath: The location of the file (for example, `C:\UtilityJar.jar`).

3. Create another shared library for the MissingMethodUtilityJar.jar. Specify the following:
   - Name: `MissingMethodUtilityJar`
   - Classpath: The location of the file (for example,. `C:\MissingMethodUtilityJar.jar`).

   See Figure 16.

Figure 16   Configuring MissingMethodUtilityJar.jar as a shared library

Start the application

To start the application, do the following:

1. In the administrative console navigate to **Enterprise Application** →
   Enterprise Application.

2. Ensure that the ClassloaderSampleEAR application is started. If it is not,
   select the check box located beside the application, and select **Start.**
   (Figure 17).
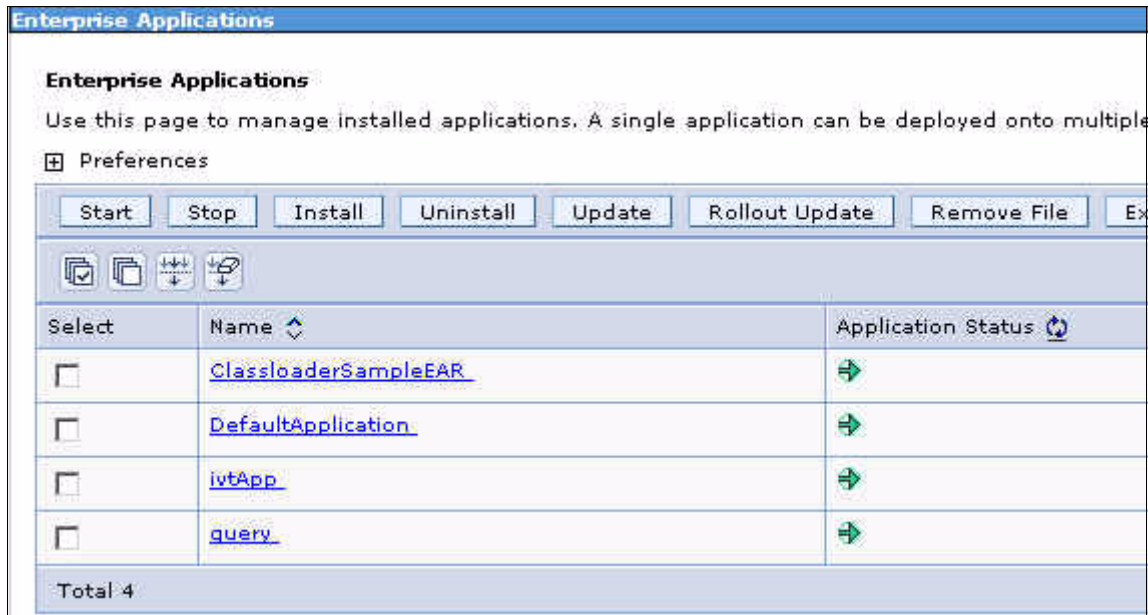
Figure 17   Starting the application

Configure the shared libraries for the application

To configure the shared libraries, do the following:

1. Click the ClassloaderSampleEAR application.

2. Select **Shared library references** under the References section.

   Figure 18 on page 60 shows the options to configure shared libraries and to
   associate them with a particular application or Web module. In this case, you
   will configure the shared library with the individual Web modules.

   Refer to for image of the initial configuration**.**

Figure 18   Associate shared libraries with the Web modules or the application

3. Select **ClassCastExceptionWeb**, and click **Reference shared libraries**.

4. Select the **UtilityJar** shared library, and click the arrow to associate it with this Web module. See Figure 19 on page 61 for the final configuration.
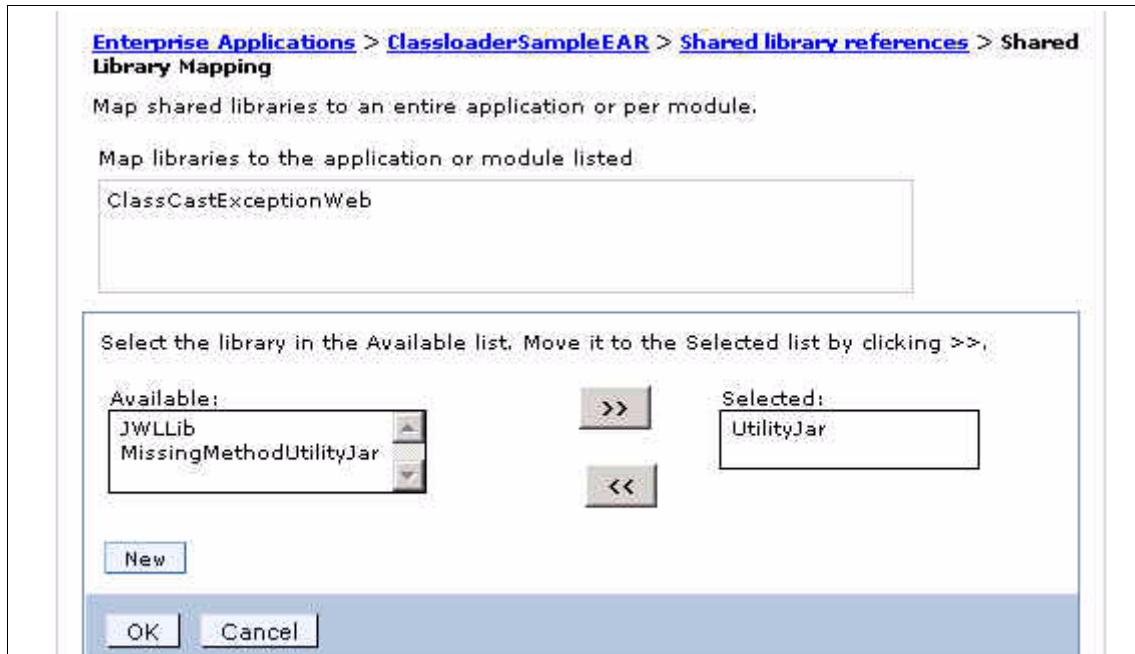
Figure 19   Associating UtillityJar shared library with ClassCastExceptionWeb web module.

5. Click **OK**.

6. Select **NoSuchMethodErrorWeb**, and click the **Reference shared libraries**.

7. Select the **MissingMethodUtilityJar** shared library, and click the arrow to associate it with this Web module. See Figure 20 on page 62 for the final configuration.
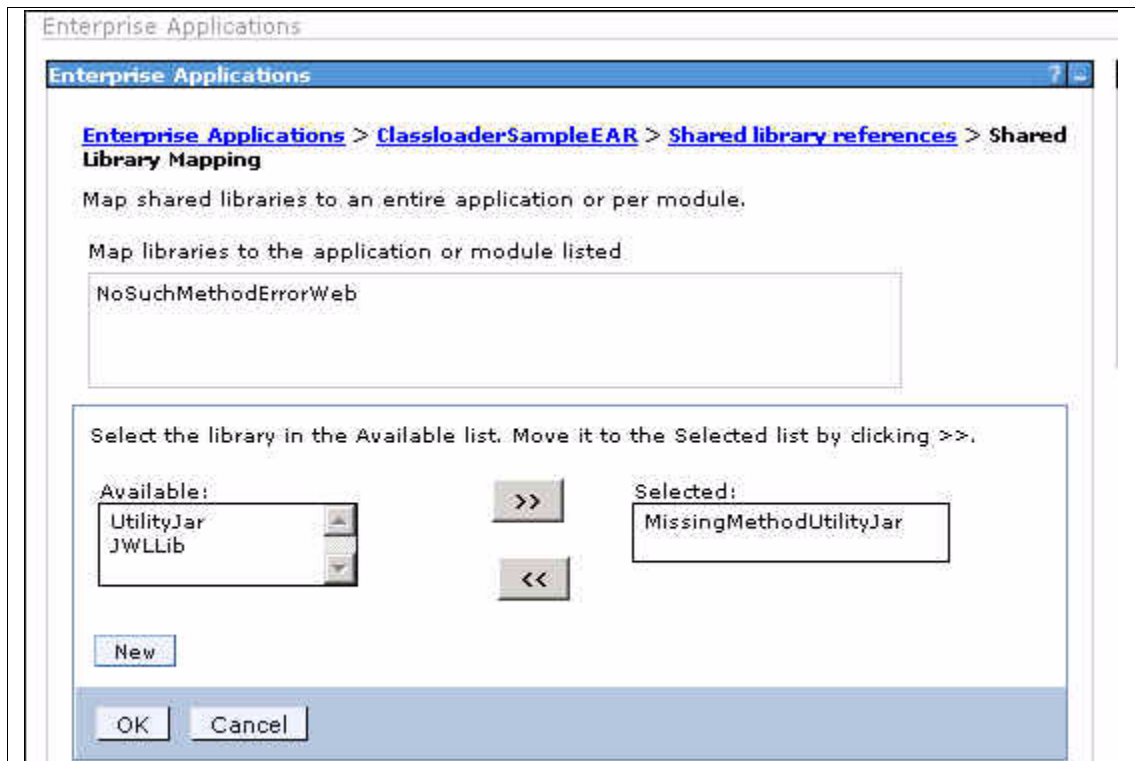
Figure 20   Associating MissingMethodUtilityJar shared library with NoSuchMethodErrorWeb web module.

8. Select **OK**.

The shared library references should be similar to Figure 21 on page 63**.**

Figure 21   Final shared library configuration

9. Click **OK** to complete this action.

10. Select the option to **Save to the master configuration** of the server.

Produce the exceptions

Ensure that the application is started. From a Web browser, invoke the servlet to reproduce each exception.

► NoClassDefFoundError:

– http://*hostname:port*/NoClassDefFoundErrorWeb/LoadClass

► ClassCastException

– http://*hostname:port*/ClassCastExceptionWeb/LoadClass

► NoSuchMethodError

– http://*hostname:port*/NoSuchMethodErrorWeb/LoadClass

► ClassNotFoundException

– http://*hostname:port*/ClassNotFoundExceptionWeb/LoadClass

Where the *hostname* is the hostname of the machine and *port* is the defaulthost port for the application server profile.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Send us your comments in one of the following ways:
- ► Use the online **Contact us** review Redbooks form found at:
  `ibm.com`/redbooks
- ► Send your comments in an email to:
  redbook@us.ibm.com
- ► Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099, 2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	Rational®	WebSphere®
IBM®	Redbooks (logo) ®	
i5/OS®	Redbooks®	

The following terms are trademarks of other companies:

EJB, Java, JDK, JNI, JVM, J2EE, Solaris, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.