

Overview

Classloaders form a basic component of Java whose function is to load classes at runtime to the Java Virtual Machine (JVM). In other words, a classloader is an object that loads classes into memory, and is responsible for navigating and loading class files at runtime.

Classes are the basic building blocks of Java, outside of which no executable code can be written. A classloader refers to a component that is used to load classes. Classloaders implement `java.lang.ClassLoader` and allow different portions of the container, and the web applications running on the container, to have access to different repositories of available classes and resources.

Classloaders in Pramati Server

The Server classloader is used for loading classes at runtime for each application that has been deployed on Server. There are various application artefacts that are provided and generated by the Server. A comprehensive list of the application artefacts are as follows:

- 1 EJB module
 - Beans
 - Homes
 - Remote interfaces
 - Helper classes packaged within the module
 - Helper classes in ext JARs
- 2 Web module
 - JSPs
 - Servlets
 - Util classes
- 3 Generated during deployment
 - IMPL generated classes – which are executed at the Server side
 - Client.jar – RMI stubs for home and remote interfaces
- 4 Generated at runtime for Java clients
 - Client.jar, which is available only after deployment

Features

The Classloader used by the Deploy Service features:

- *Extension Class Loader* - Parent of all application class loaders and has the capability to pick up classpath on the fly detecting the addition of jars to the classpath.
- *Application Groups* - Ability to share classes among applications using an application group. A group can be defined in *deploy-config.xml*.
- *Application level Class Loader* - One single class loader with preloaded classes for all of the applications which would enable even EJB to pick up web classes.
- *Module level class loader* - Class Loader for each web module, enabling explicit redeployment of a single module without bringing the whole application down.

Application classes precedence can be set by a flag for an application or globally for all applications. You can get the application to load its own classes in the system classpath. This would be available only on demand and not as a default option.

Classloader hierarchy

In the earlier versions of JDK, there were no relationships between classloaders. Java 1.2 introduced classloaders in a parent-child relationship, where a parent classloader branches out to child classloaders. This relationship between parent and child classloaders is akin to the object relationship of super classes and subclasses. This led to increased security as it was possible to assign modification rights attributes to classloaders.

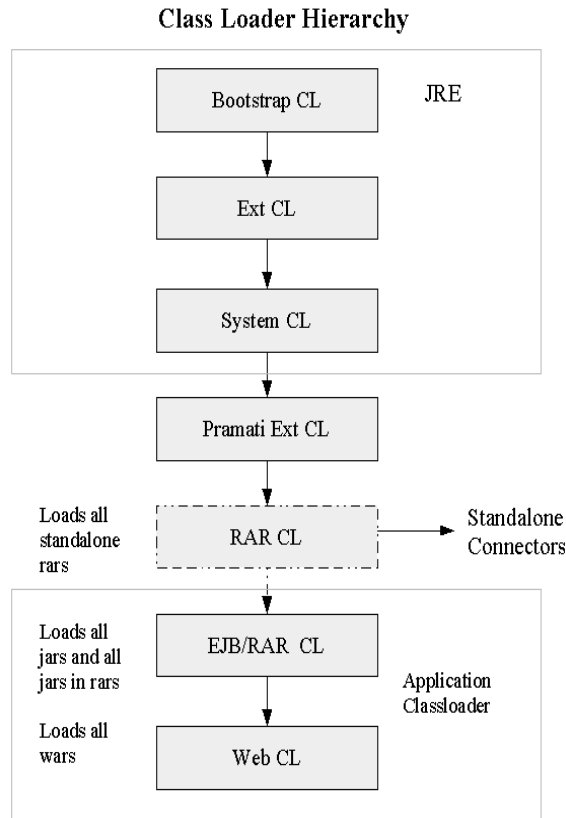


Fig The Classloader Hierarchy

The CLs can be added at any time during code execution. Every CL has a parent, other than the Bootstrap CL.

When a CL is asked to load a particular class or resource, it delegates the request to its parent CL first, which, in turn, sends it to its own parent CL up the hierarchy. If the requested class/resource is not found, the request is sent back to the parent.

Bootstrap

The Bootstrap CL is the parent in the Java classloader hierarchy. It has 'null' parent. The JVM creates the Bootstrap CL, which loads the Java Development Kit (JDK) internal classes and `java.*`

packages included in the JVM. (For example, the Bootstrap CL loads `java.lang.String`.) It loads the basic runtime class of JVM – `'rt.jar'`, along with any classes from JAR files present in the System Extensions directory (`$JAVA_HOME/jre/lib/ext`). Bootstrap serves as the default parent for Extension CL and System CL.

Note: Some JVMs may implement this as more than one classloader, or it may not be visible as a CL.

JDK Extension

The Extension CL is a child of the Bootstrap CL. The Extension CL is the parent of all Application CLs and has the capability to pick up classpath, and detect the addition of jars to the classpath. It loads any JAR files placed in the Extension directory of the JDK. This is a convenient means of extending the JDK without adding entries to the classpath. However, anything in the Extension directory must be self-contained and can only refer to classes in the Extension directory or JDK classes. It loads the classes present in the directory path `'jre/lib/ext'`.

System

The System CL is the child of the Extension CL. It loads the classes from the “classpath” environment variable.

Pramati Extension CL

The CL `.jre/lib/ext` directory can be changed by pointing to a node specific directory. The Extension CL is also used during loading of external classes like DataSources and startup hooks etc. Jar files, classes directories or zip files can be dropped without the server being stopped in the `.jre/lib/ext` dir.

Application

The Application classloaders are created by pramati server to manage application class loading. Different strategies are handled to suite specific requirements.

Note: The behavior of the above structure is not related to `'jre/lib/ext'`. The JARs doesn't get picked up, when you copy them manually. The classpaths have to be set for the JARs to be picked.

Application classes precedence

By setting a flag for the whole application or a web module, you can get the application to load its own classes over the same class in the system class path. This would be available only on demand and not as a default option. This is usually required for applications which need to use their own version of classes rather than the ones which the server uses.

Application Groups

Ability to share classes among applications using an application group. A group can be defined in `deploy-config.xml`. Grouping enables an application to access the classes packaged in another application in the same group. When an application belonging to a group is restarted all started applications of that group have to be restarted because the applications use a common classloader.

When the Deploy Service tries to deploy an application, it loads the EJB/RAR classloader and web classloader. After which the application loads the classes through these classloaders. The EJB/RAR classloader loads all jars and all jars inside rars. The web classloader loads all jars required for the web module. The Bootstrap classloader, Ext classloader and the System classloader are loaded by the Java runtime environment.

The Pramati Ext classloader loads all other external classes from the directory specified in the server-config.xml. The default directory is <install_root>/server/libext. The RAR classloader loads all the standalone RARs.

A single class loader can be used to load multiple applications, if the applications are setup in a logical group. Class loader for multiple applications can be set in deploy-config.xml.

```
<classloader-for-multiple-apps enabled="false">
  <application-group apps="abcd.ear,xyz.jar" />
</classloader-for-multiple-apps>
```

In the above case if 'classloader-for-multiple-apps' is false, a single classloader is loader for each application. In the above case, if no group is defined, a universal group for all the applications is assumed.

Any application not mentioned in the apps list as shown above will get its own dedicated class loader. If no group is specified, then a single classloader will be used.

Single classloader for an application

The Deploy Service can be instructed to use a single classloader at the time of deployment of web, connector and bean components within an application from the pramati-j2ee-server.xml file.

```
<use-single-classloader-for-app>
  true
</use-single-classloader-for-app>
```

CLs have preloaded classes for all applications which would enable even an EJB to pick up web classes, once this feature is enabled. The web loader would be the same instance and not use the EJB CL as a parent to enable web classes access by EJB.

Preferring Deployed Classes

This option is used to specify the preference of the classes for the classloader at the time of deployment of the application.

```
<prefer-deployed-classes for="application"/>
```

In this case if the application has classes which are also present in the system classpath, the application classes are picked up by the classloader. In this case the application includes web modules also.

```
<prefer-deployed-classes for="web-module"/>
```

If the web module classes are to be preferred over the other classes, the above option can be specified.

```
<prefer-deployed-classes for="none"/>
```

In this case there is no preference for the classloader and the classes are loaded normally.

Setting up the Ext Directory

By default the external classes are loaded from `<install_root>/server/libext`. But a different ext directory can be specified, from where the classes can be auto loaded. This can be done by using `<ext-classloader-dir>` from `server-config.xml`. Since `server-config.xml` is specific to a server instance, multiple ext directory can be set for multiple server instances.

Classloader Lifecycle

When the Server starts,

- 1 Bootstrap CL loads all the JDK related classes
- 2 Extension CL loads all the classes present in the JRE/LIB/ext folder
- 3 System CL loads all the classes set in the Classpath

All classes that form part of Server are placed in System Classpath.

All the EJB modules (JARs) within an application have a single CL, whereas each web module (WAR) has a separate CL. The structure is such that the System Classpath acts as the parent for the EJB modules, which, in turn, serves as the parent for each of the web modules. In other words, each instance of the web module is a direct child of the EJB module.

It is common for web applications to call EJBs, Server application classloader architecture allows JSPs and servlets to see the EJB interfaces in their parent classloader. This architecture also allows web applications to be redeployed without redeploying the EJB tier. It is more common to change JSP pages and servlets than to change the EJB tier.

Application Classloading in Server

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

Every application receives its own classloader hierarchy, and the parent of this hierarchy is System CL. This isolates applications so that one application cannot see the CLs or classes of another. In CLs, no sibling or friend concepts exist. Application CLs can only see their parent classloader - the System CL. This allows Server to host multiple isolated applications within the same JVM.

Server automatically creates a set of CLs when an application is deployed. The base application CL loads any JAR files in the application. A child CL is created for each web application.

Packaging JAR, WAR and EAR

Pramati Server classloading is centered on the concept of an application. An application is normally packaged in an Enterprise Archive (EAR) containing application classes. Everything within an EAR is considered part of the same application. Other applications include:

- An Enterprise JavaBean (EJB) JAR. All the EJB module files such as the enterprise beans, homes, remote interfaces, helper classes packaged within the module, and helper classes in ext JARs are packaged as a JAR.
- A web application WAR. All the web module files such as .jsp, .html, Util classes, and servlets etc., are packaged as a WAR.

An EAR is a package of all the JARs and WARs.

If you deploy a JAR and a WAR separately, they are treated as two applications by the Server. If they are deployed together within an EAR file, they are treated as one application. This produces a classloader arrangement that allows servlets and JSPs to find the EJB classes.

All JARs should be explicitly placed in the system classpath.

Placing application files in other paths

Application files may also be placed in the:

- System classpath
- Pramati's Extension directory (<install_dir>/lib/ext)

You can place a driver such as the Oracle JDBC Driver *classes12_01.zip* in the system classpath.

Packaging helper JARS

Applications usually have shared utility classes. If you create or acquire utility classes that you will use in more than one application, you must package them with each application as separate JAR files. The JAR files should be self-contained and should not have any references to the classes in the EJB or web components. Common types of shared utility classes are data access objects or JavaBeans, which are passed between the web and EJB tier.

You can also add shared utility classes to the Java system classpath. If you modify your utility classes and they are in the Java system classpath, however, you will have to restart the Server after you modify the utility classes.

Helper JARs may include the utility jars that serve to provide some additional functions, and may be duly referenced by using the path entries in MANIFEST.MF.

Functionality of MANIFEST.MF

Every JAR is a zip file that contains the META-INF directory within itself. This META-INF directory is the home of MANIFEST.MF file.

Besides other information, the MANIFEST.MF contains details regarding:

Main-Class: For example, com.pramati.j2ee server

Class-Path: For example, a.jar

The J2EE specification provides the manifest classpath entry as a means for a component to specify that it requires an auxiliary JAR of classes. You only need to use this manifest classpath entry if you have additional supporting JAR files as part of your JAR or WAR file. In such cases, when you create the JAR or WAR file, you must include a manifest file with a classpath element that references the required JAR files.

The following is a simple manifest file that references a utility.jar file:

Manifest-Version: 1.0 [CRLF]

Classpath: utility.jar [CRLF]

In first line of the manifest file, you must always include the *Manifest-Version* attribute, followed by a new line (CR | LF |CRLF) and then the *Classpath* attribute. More information about the manifest format can be found at: <http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR>

The manifest classpath entries refer to other archives relative to the current archive in which these entries are defined. This structure allows multiple WARs and JARs to share a common library JAR. The manifest file itself should be located in the archive at META-INF/MANIFEST.MF.

For more information, see <http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html>

Effective path in case of multiple MANIFEST.MF files

The effective path to be used if there are references to multiple MANIFEST.MF files is according to the order in which they appear.

For example, let's say we have the following jar files:

a.jar ; b.jar ; c.jar and d.jar.

The classpath in the MANIFEST.MF of a.jar is: b.jar /..folder/c.jar, and the classpath of b.jar refers to d.jar.

The effective class path to be used is: a.jar;b.jar;d.jar;c.jar.

Important

In other words, if e.jar is loaded at run time, and makes a call to f.jar, which needs g.jar to be loaded first, you will get an exception if g.jar is placed lower in the CL hierarchy than f.jar.

What happens when:

e.jar calls f.jar - The search begins only after the request reaches the parent BootStrap CL that looks for f.jar in the Extension CL, and then System CL where it finds it.

f.jar calls for g.jar - Once found, f.jar makes a call to g.jar, which, since it is not in the same CL, the search will again take the 'highest to lowest' approach – in the meanwhile, e.jar, not being able to run f.jar, would give off an exception.

Hence, placing the jar files in the proper hierarchy is of utmost importance so as to not provide those exceptions.

Case Studies

Case 1

There are two classes class1 and class2, and there are two jar files EJB_JAR1 and EJB_JAR2.

The two classes are required only by EJB_JAR1, and not by EJB_JAR2.

The contents for EJB_JAR1 would be the beanclasses + class1 + class2 + ejb_jar.xml (present in the META-INF folder of the JAR file.)

Case 2

There are two classes class1 and class2, and there are 'n' number of jar files EJB_JAR1... EJB_JARn in EAR1.

The two classes are required only by all the JAR files.

The contents for EAR1 would be EJB_JAR1 +... + EJB_JARn + common.jar (that has class1 and class2.)

Note: Remember, the MANIFEST.MF should have a reference to the classpath entry of common.jar.

Deployment scenarios

In code generation and compilation stage of deployment process, certain classes and JARs are generated.

- **IMPL generated classes:** IMPL generated classes are executed at server side. EJB module JAR and all utility JARs referred by EJB module JAR through manifest entries are set in the classpath.
- **Client.jar:** The Client.jar contains RMI stubs for Home and Remote interfaces for all EJBs in an application. This JAR is required to be placed in the client classpath if the client is running on another VM.

This JAR is located at <path-up-to-node>\archives\ <application name>\classes

<path-up-to-node> refers to <install-dir>\server\nodes\default, where 'default' could be a folder created by a user with a different name.

Undeployment

Undeployment of an application unbinds those resources currently being used by the running application, from the Naming Service. It deletes generated files, Remote/Home interfaces, and stubs from the file system.

Redeployment

Server allows you to deploy newer versions of application components such as EJBs while Server is running. A class should be reloaded when:

a) Modifications are made in JSPs

This is the case when after a class has been loaded using a CL, the JSP page is modified.

After making the changes in the JSP and after recompiling the page, if you try to load the class again using the CL, you will find that the JSP page does not reflect the changes. The reason is although recompilation updates the file, the CL does not load the new file because it picks up JSP pages loaded earlier using a map to its existing paths.

To overcome this, the JSP engine uses smart code generation to check the time stamp of the modified file against the existing file. It changes the class name with an incremental value and places it in the same path, so that the CL picks up the updated file.

b) Application is redeployed

When you have to deploy a new version of an already deployed application on your system, you need to create a new application CL.

You cannot use the existing CL because classes might have changed in the new version, which you should reload.

Smart code generation skips classes that have not been modified, loads new classes and creates a new CL structure.

Stopping a Server

Stopping a Server stops all services currently running on Server and the current CL hierarchy ceases to exist.

Java clients

Classes/JARs required for Java clients to access EJBs

The Classes and JARs that are required by the Java clients for accessing EJBs are:

- Application Client.jar
- Application Helper JARs

How are each of these available?

Application client.jar

During the preparation phase, RMI Stubs of Home and the Remote Interfaces of EJBs are placed in the client.jar.

Application helper JARs

These are to be provided by the application, with a reference in the client Classpath.

Accessing Message Server from application

If a J2EE application that is being accessed through a Java client throws a runtime exception, the client needs to have that exception in its Classpath. A typical example is the message class that is used in typecasting the message.

Where should the message class be?

This class is needed by the accessing points – beans, jsps, servlets, etc.

These application exceptions need to be packaged as separate JAR files as util.jar within the EJB module, and the same JAR file must also be placed on the client VM at runtime.

To avoid a classcast exception while casting a JMS message content to a particular class, the class must be packaged in the EJB module's jar in which the MDB resides. This is the preferred method to avoid any classcast exceptions.

The class can also be packaged in the utility.jar, which is referred by the EJB module through the Classpath entries in MANIFEST.MF.

Specifying JDBC drivers

Distributing JDBC drivers with application

The JDBC drivers must not be packaged in EAR. Instead, they should be available to the system classpath.

setup.bat

Packaging the JDBC drivers within the setup.bat is the best option as all the entries are directly stored in the system classpath.

JRE lib/ext

The JDBC drivers can be placed in JRE lib/ext.

Note: The use of pramati's lib/ext directory is highly recommended.

Problems

Some of the common classloader problems are:

Excessive visibility

This occurs when a class is duplicated and surfaced as:

ClassCastException

The cause of this error is simple, say, casting an Integer to String. Often, the situation is one of "same class, different loader" type – where the source and target types have the same class name.

Class duplication

This problem arises when:

- 1 cross-application dependencies are managed by copying dependent JARs into each application.
- 2 an application is reloaded, the container creates a new classloader and uses it to reload the application classes. In this scenario, however, the original classes may still be accessible.
- 3 some JNDI implementations do not serialize objects that are locally bound and retrieved. Thus, when applications in the same JVM store and retrieve an object, that instance is shared, and the duplicate class problem can occur.

Normally, the existence of duplicate classes is not a problem. In order for an exception to occur, an instance created in one CL must be passed into the CL of the other. This can be achieved through any storage facility visible to both classloaders, including:

- Static fields
- Global collections
- JNDI

Low visibility

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

ClassNotFoundException

This error occurs during dynamic loading, via any method that explicitly loads a class (e.g., `Class.forName()` or `ClassLoader.loadClass()`).

NoClassDefFoundException

This error occurs when

- the application code tries to instantiate an object using the `new` operator, or
- if dependencies of a previously loaded class cannot be resolved. This is confusing due to invisible dependencies.

Mutilated visibility

Although rare, these are serious errors.

IncompatibleClassChangeError

This error indicates that a superclass or interface has changed.

ClassCircularityError

This error indicates that the current class exists as one of its own superclasses or superinterfaces.

UnsupportedClassVersionError

This error occurs when loading a class compiled in a more recent JDK version than the one in which it is running.

VerifyError

This error occurs when the code in the class violates one of the constraints imposed by the JVM.

ClassFormatError

This error indicates that the class file format is invalid, often due to corruption.

FAQs

If a message is being cast, where does the class come from?

To avoid a classcast exception while casting a JMS message content to a particular class, the class must be packaged in the EJB module's jar in which the MDB resides. This is the preferred method to avoid any classcast exceptions.

The class can also be packaged in the utility.jar, which is referred by the EJB module through the Classpath entries in MANIFEST.MF.

What goes into the client jar during Preparation?

During the preparation phase, RMI Stubs of Home and the Remote Interfaces of EJBs are placed in the Client.jar.

Where to place application exceptions when using Java clients?

If a J2EE application that is being accessed through a Java client throws a runtime exception, the client needs to have that exception in its Classpath. These application exceptions need to be packaged as separate JAR files (util.jar) within the EJB module, and the same JAR file must also be placed on the VM client so as to be able to catch them at runtime. Refer to the exceptions in the section on 'Common Classloader problems'.

Where to place application exceptions when Concentrators access EJBs in Pramati Server?

Concentrators are clients as other web servers, portal applications etc. that run on other servers and try to access EJBs deployed on Pramati Server. To allow concentrators to access EJBs in the Server, do the same as for handling exceptions.

How to specify the Classpath option when starting Server from a remote VM using Java program?

If the Server is started from a remote VM using a Java program, the classpath option must be specified giving the complete path for running the application. A code sample for the same is provided below:

```
// code snippet for starting J2ee server.  
ArrayList commandList=new ArrayList();  
if(System.getProperty("os.name").startsWith("Windows")) file://for windows &  
NT
```

```
commandList.add(javaHome + File.separator + "bin" + File.separator +
"javaw");
else file://any other OS
commandList.add(javaHome + File.separator + "bin" + File.separator +
"java");
commandList.add("-D<a>=<b>"); // add jvm options; for every jvm option add
one entry like this.
commandList.add("-classpath");
commandList.add(); file://add server classpath here;
// that is the class path set by the setup.bat file
commandList.add("-Dinstall.root=/server");
commandList.add("com.pramati.J2eeServer");
// following two lines are not required for default server.
commandList.add("-config");
commandList.add("/server/nodes/StandAlone//co
nfig/server-config.xml"); // for stand alone server.
or
commandList.add("/server/nodes/Cluster//confi
g/cluster-config.xml"); // for cluster node.
commandList.add("-username"); // this is not required if username password is
commandList.add(""); // "root" and "pramati" respectively.
commandList.add("-password"); //
commandList.add(""); //
commandList.add("-redirect");
commandList.add("-noCommandLine");
Runtime.exec(commandList.toArray(new String[0]));
// end of code snippet
```

Classloading guidelines

Here are some of the best practices for classloading:

Declare dependencies

Make dependencies explicit. Hidden or unknown dependencies will be left behind when you move your application to another environment.

Grouping dependencies

Ensure that all dependencies are visible at the same level or above. If you move a library, make sure all dependencies are still visible.

Minimize visibility

Dependency libraries should be placed at the lowest visibility level that satisfies all dependencies.

Share libraries

Avoid duplicating libraries. Use the parent attribute to share classes across a set of applications. Use the tag in the global application.xml file to share classes across all applications.

Keep configurations portable. Choose configuration options in the following order:

- 1 Standard J2EE options
- 2 Options that can be expressed within your EAR file
- 3 Server level options
- 4 J2SE extension options

Use the correct CL

If you call `Class.forName()`, always pass the CL returned by `Thread.currentThread().getContextClassLoader`. If you are loading a properties file, use `Thread.currentThread().getContextClassLoader().getResourceAsStream()`.

Glossary

ClassLoader

An object that performs classloading at runtime.

ClassLoading

The process of installing a class file into the JVM to create a class.

BootStrap CL/Primordial CL

JVMs in-built CL responsible for storing the core system classes.

User-defined CL

An instance of a subclass of CL with user defined behavior (overridden methods) for certain aspects of class loading. It is responsible for loading all classes other than the core system classes.

Class Loader Hierarchy

