IBM Software Group

WebSphere Application Server Classloader Best Practices

Andrew D. Hans
WebSphere Application Support Team Lead

Agenda

- Background: What is a Classloader?
- WebSphere Classloader Hierarchy
- J2EE structure and why it's important
- WebSphere's Application Classloader settings
- Where NOT to put your common application files
- Pitfalls and how to avoid them
- Useful links

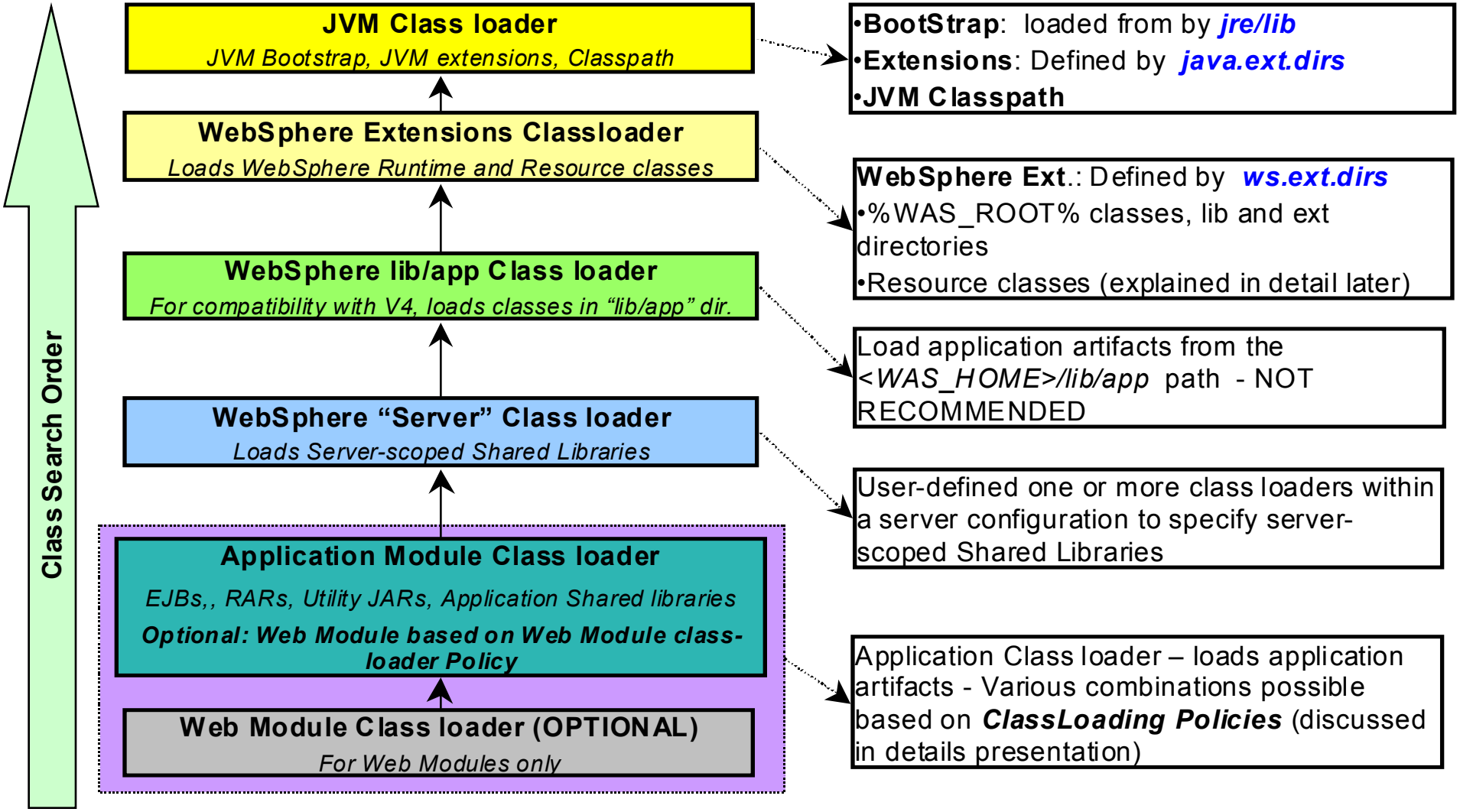
Background: What is a Classloader?

- In short – A classloader is an object responsible for loading classes and resources
- Important to note
  - A class (resource) is loaded exactly once per classloader instance
  - Once a class is successfully loaded the JVM caches it in a table that associates it with the classloader that loaded it
  - When loading a dependent class, the JVM invokes the classloader that originally loaded the previous class definition

What is Classloader Delegation?

- **Every classloader has a delegation mode**
  - Parent_First - the classloader delegates the loading of classes to its parent classloader before attempting to load the class from its local classpath
  - Parent_Last - classloader attempts to load classes from its local classpath before delegating the class loading to its parent
    - Allows overriding of higher level classes by searching locally first

Class Loader Hierarchy – At a Glance



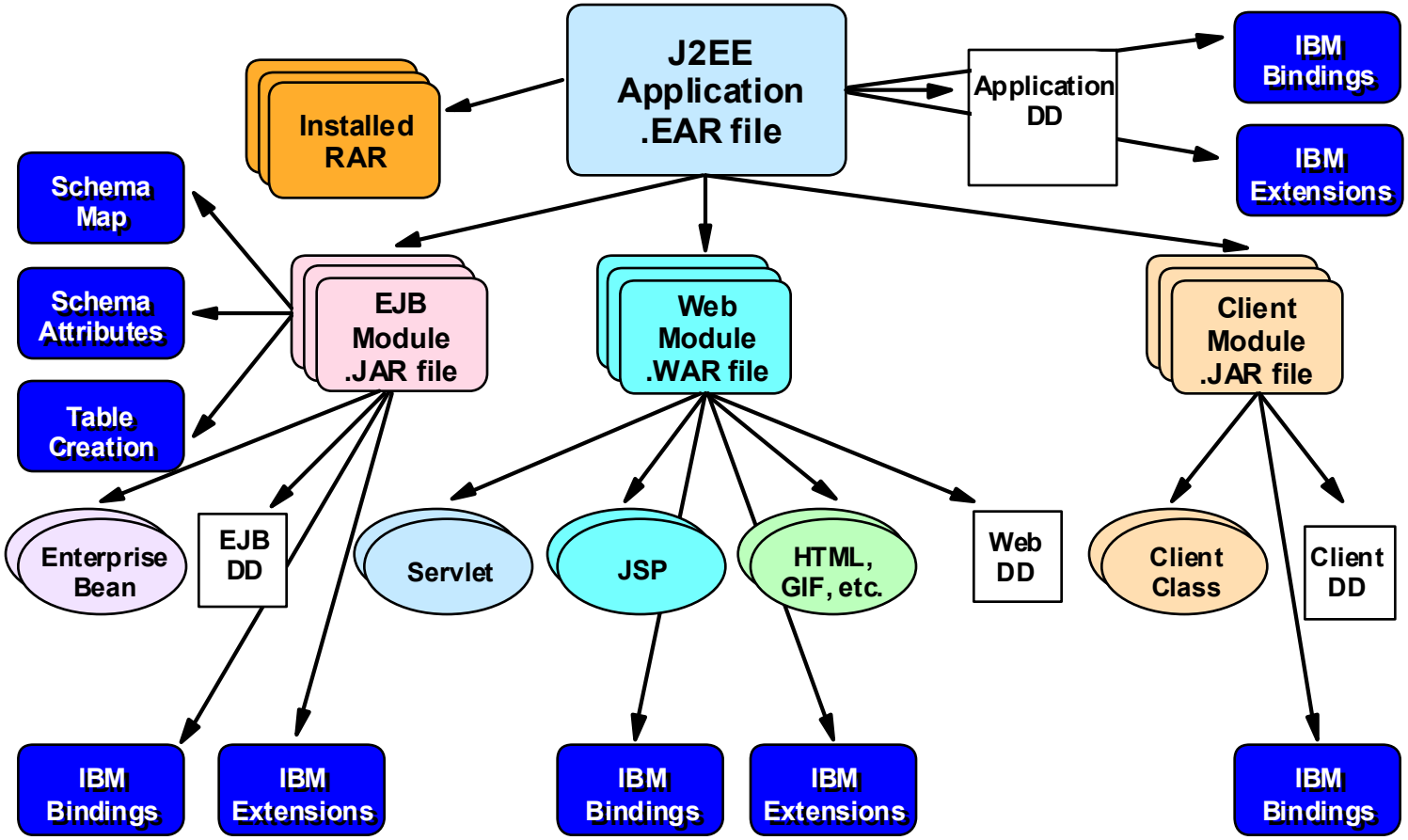
WebSphere Application Classloaders

- Loads J2EE application artifacts packaged in an EAR
  - EJB Jars
  - WARs
  - Dependency Jars
  - Application-scoped shared libraries
  - Embedded RARs
- A WAS Application classloader consists of
  - A single *Application module classloader* at the root
  - Zero or more *WAR module classloaders* having the application module classloader as its parent

J2EE Structure

- Java 2 Enterprise Extension (J2EE) - an application server framework from Sun Microsystems for the development of distributed applications
- WebSphere Application Server implements the J2EE framework
  - WebSphere 4.0x supports J2EE 1.2
  - WebSphere 5.x supports J2EE 1.3
  - WebSphere 6.0x has been certified on J2EE 1.4

J2EE 1.3 Application Packaging



DD = Deployment Descriptor

Why is the J2EE structure important

- The J2EE structure is important to understand as it is the basis of the application classloaders structure
- There are multiple ways to alter the number and functioning of application classloaders but these are dependent on the J2EE application structure

Configuring Application Classloaders

- Application Classloader policy - applies to how the applications share classloaders or not
  - At the Application Server level - choose SINGLE or MULTIPLE
    - SINGLE means the EJB, embedded RAR modules and dependent JARs for all the EARs are loaded by one classloader called the Application classloader
    - MULTIPLE means the EJB, embedded RAR modules and dependent JARs for each EAR are loaded by its own classloader
      - Default setting
    - Whether the WAR is loaded by this Application classloader is dictated by the WAR classloader policy

Configuring Application Classloaders (2)

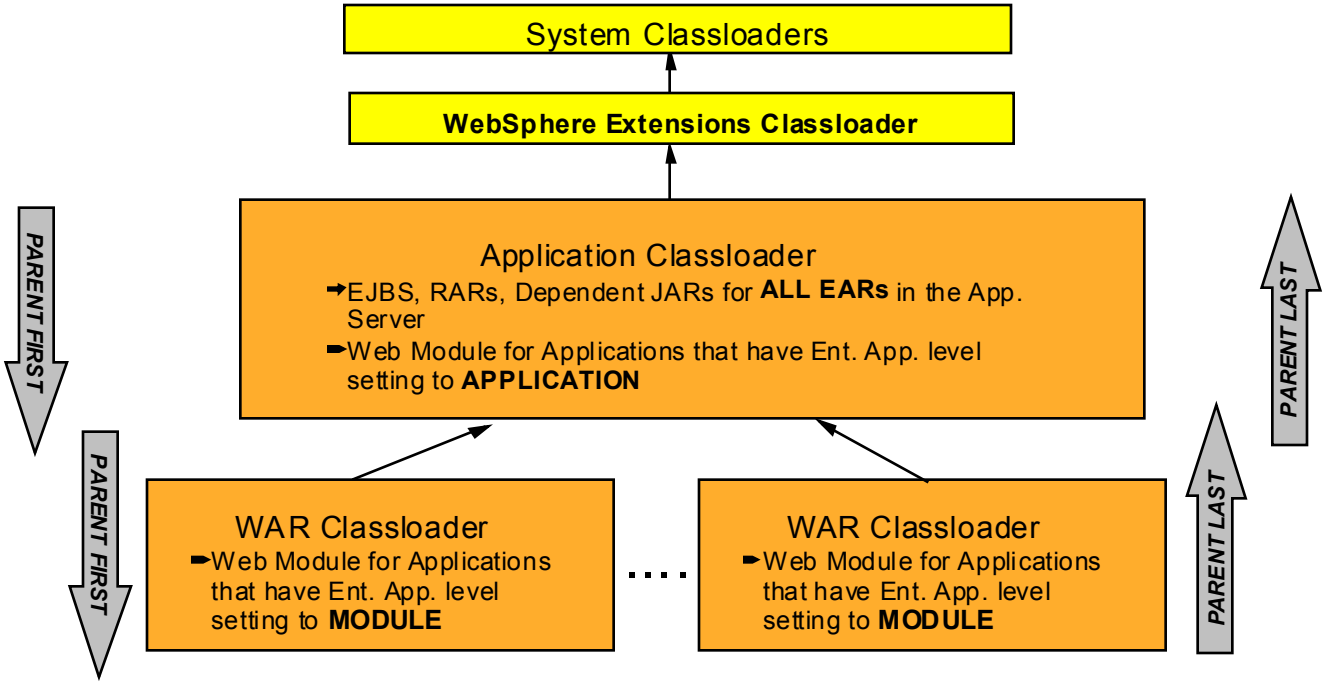
- WAR classloader policy - can choose how the WAR modules are loaded per Application
  - At the Enterprise Application level - choose APPLICATION or MODULE
    - APPLICATION: all the Web modules in the application EAR use the Application classloader (dictated by the Application Classloader Policy)
    - MODULE: every WAR uses its own classloader, different than the Application Classloader
      - Default setting
    - Selection can be made at application install time

Configuring Application Classloaders (3)

- Classloader Mode – can be set for Application Classloader and WAR classloader
  - PARENT_FIRST - default
    - Search the immediate parent first and then its policy would determine if that was successful, if not its parent
  - PARENT_LAST
    - Tries to find and load the class from its own classloader and if the class was not found, it delegates to its immediate parent classloader and then the immediate parent classloader's policy would take control

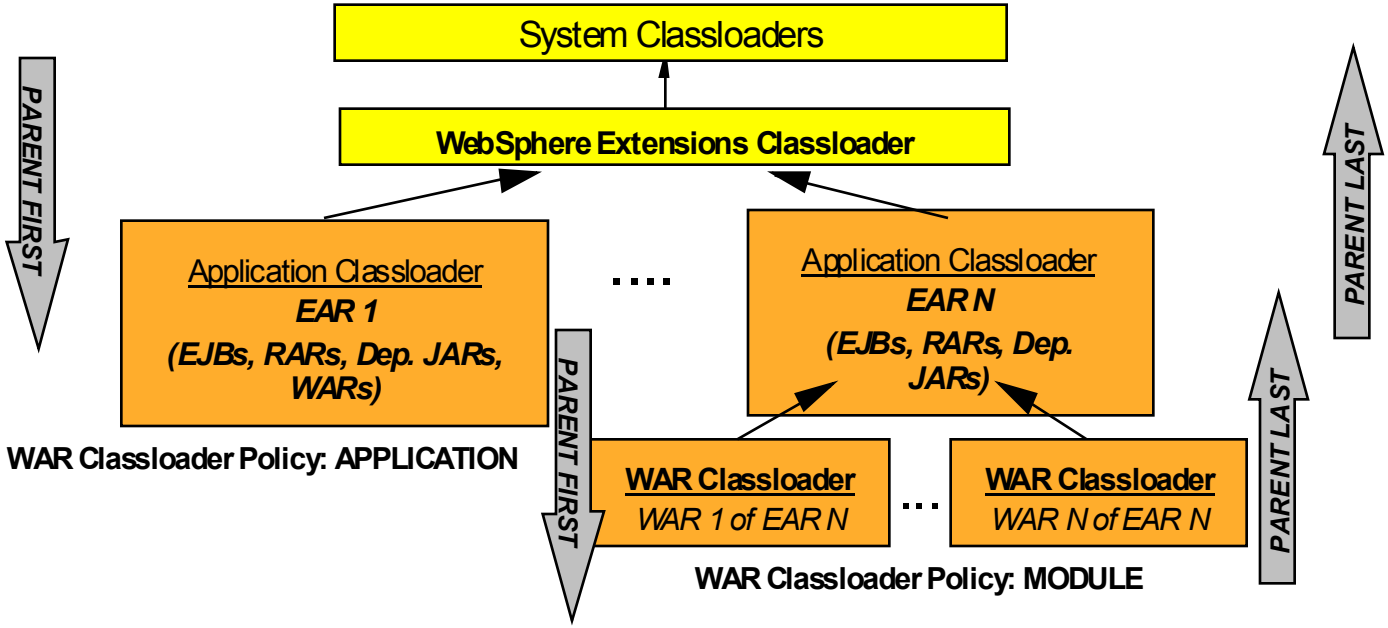
Application Classloader Policy - SINGLE

- Application Classloader policy of SINGLE and WAR classloader policy of APPLICATION or MODULE
  - Pros: Each Application EJBs, embedded RARs, Dep. JARs can reference other classes in other Applications
  - Cons: Cannot start and stop individual Applications without affecting others and not considered to be J2EE compliant



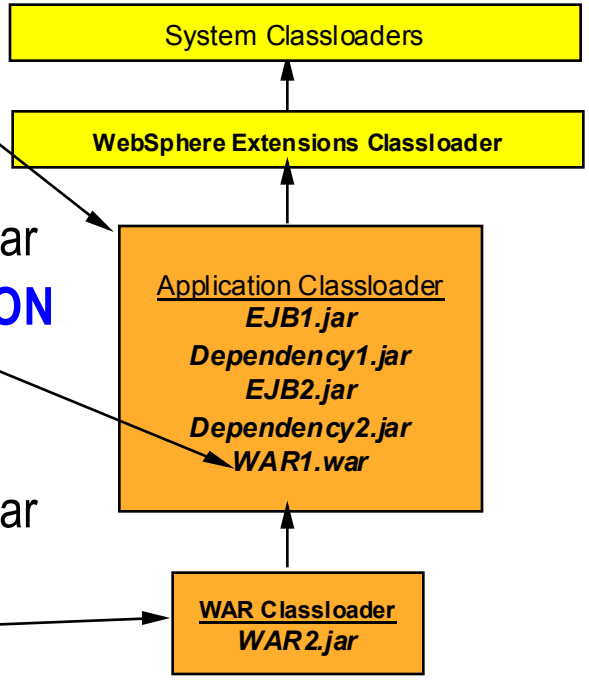
Application Classloader Policy - MULTIPLE

- Application Classloader policy of MULTIPLE and WAR classloader policy of APPLICATION or MODULE
- Pros:
  - Can restart each application without affecting others
  - Classes within each EAR can reference all the classes with the same EAR even if in different modules of the EAR
- Cons: Classes within one EAR cannot reference classes in another EAR



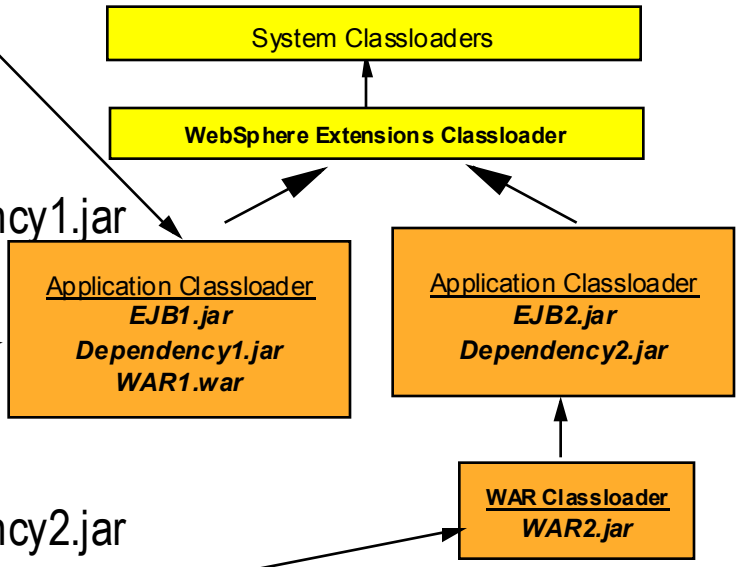
Example 1

- Application classloader policy: **SINGLE**
- Application 1
  - Module: EJB1.jar
  - Module: WAR1.war
    - MANIFEST Class-Path: Dependency1.jar
    - WAR Classloader Policy = **APPLICATION**
- Application 2
  - Module: EJB2.jar
    - MANIFEST Class-Path: Dependency2.jar
  - Module: WAR2.war
    - WAR Classloader Policy = **MODULE**



Example 2

- Application classloader policy: **MULTIPLE**
- Application 1
  - Module: EJB1.jar
  - Module: WAR1.war
    - MANIFEST Class-Path: Dependency1.jar
    - WAR Classloader Policy = **APPLICATION**
- Application 2
  - Module: EJB2.jar
    - MANIFEST Class-Path: Dependency2.jar
  - Module: WAR2.war
    - WAR Classloader Policy = **MODULE**



Where NOT to put files

- Do NOT put files that are needed by your application in the following directories
  - JVM classpath (including <Java_Home>/jre/*)
- Why?
  - The JVM classloader is the parent of all other classloaders and will have visibility from all classloaders at a lower level and will override any WebSphere supplied files by the same name
  - If you place a file here it cannot find any files that are in a lower classloader

Where NOT to put files (continued)

- Do NOT put files that are needed by your application in the following directories as any files placed here can interfere with the WebSphere runtime
  - <WAS_HOME>/lib
  - <WAS_HOME>/lib/ext
    - Custom user registry implementation should go here as they are required by the WSAS runtime at startup
  - <WAS_HOME>/classes
    - Should only be used for test patches supplied by WAS support

Pitfalls – Too much visibility

- This occurs when a class or jar is visible to more files than intended
- **Example:** A file named Ex1.jar is located in a shared library that is associated with multiple application servers but is only needed by one WAR file running on one server
- **Dangers:** With this jar visible to so many applications on more than one server, an application added at some later point may accidentally pick it up
- **Recommendations:** Place the jar only as high in the hierarchy as needed and move higher or associate more only when needed
  - This will reduce the risk of causing problems when more applications are added or configuration changes made in the environment

Pitfalls – Too little visibility

- This occurs when a class or jar is too low in the hierarchy to be visible to all the application code that will need access to it
- **Example:** A file named Ex2.jar is located within the root of an EAR file (Ex2_A.ear) however a WAR file in another EAR (Ex2_B.ear) also needs to use this jar
- **Dangers:** The Ex2.jar will need to be visible to both Ex2_A.ear and Ex2_B.ear
  - Placing it into both EAR files is not recommended as the jar will now be in both locations and multiple instances loaded several classloaders
  - Placing it in a shared library associated to the servers these run on also is not recommended as it can cause the pitfall of too much scope just discussed
- **Recommendations:** Creating a shared library to hold the Ex2.jar and associating it to the two applications that need it will eliminate the problem and avoid the pitfall of too much visibility as well

Pitfalls – Duplicated files

- This occurs when a file is in more than one location in a configuration
- **Example:** Multiple EAR files contain a jar Ex3.jar
- **Dangers:** This will cause the Ex3.jar to be in the environment many times and if large enough can cause memory, file space, or maintenance issues
  - This is a classic cause of a ClassCastException
- **Recommendations:** Pull the Ex3.jar from each EAR file and place it into a shared library to associate with the applications or servers they run on

Pitfalls – Chained dependencies

- This occurs when one jar needs visibility to a second jar which in turn depends on a third jar
- **Example:** A.jar is located within a WAR file and depends on B.jar located at the root of the EAR file which depends on C.jar which is also in the WAR file
- **Dangers:** This will fail since the searching for C.jar will begin with the classloader from which B.jar was loaded
  - Classic cause of ClassNotFound and NoClassDefFoundError messages
- **Recommendations:** Keep these dependencies in mind and try to place dependent jars in the same classloader unless needed higher in the hierarchy

Pitfalls – Picking up the wrong version of a file

- This occurs when multiple versions of the same jar are within the configuration and mistakenly the wrong one is being used
- **Example:** Jar ex5.jar is within an EAR file and different versions of the same class files reside in ex5_A.jar within a shared library associated to the application server
- **Dangers:** With the default delegation the ex5_A.jar will be picked up first
  - Classic cause of Linkage errors
- **Recommendations:**
  - Carefully set the delegation modes to pick up the appropriate version of the jar (normally by switching to Parent_Last)
  - Limit versioning as much as possible
  - Do NOT override a WebSphere provided jar by placing another version in the JVM classpath

How to resolve a ClassNotFound problem

- Ask yourself the following questions
  1. What file cannot be found?
  2. Where is the file that cannot be found?
  3. Where is the file that is looking for it?
  4. Does it have visibility to the file it is trying to load?
  5. Which classloader is attempting to load the class?
- If you answer these questions you should be able to correct a ClassNotFound problem by enabling visibility to class that cannot be found

Useful Classloader Links

- Best Practices for Using Common Application Files
  - http://www-1.ibm.com/support/docview.wss?uid=swg27006159
  - Gives step by step instructions and a decision tree to help you decide which setting is best for your configuration
- IBM Education Assistant: WebSphere Application Server Classloader
  - http://www-1.ibm.com/support/docview.wss?uid=swg27005457
  - Gives multiple presentations on the WAS classloader
- TroubleShooting Classloader Issues
  - http://www-1.ibm.com/support/docview.wss?uid=swg21219358
- Must gather information for Support
  - http://www-1.ibm.com/support/docview.wss?uid=swg21137435

Additional WebSphere Product Resources

- Discover the latest trends in WebSphere Technology and implementation, participate in technically-focused briefings, webcasts and podcasts at: www.ibm.com/developerworks/websphere/community/
- Learn about other upcoming webcasts, conferences and events: www.ibm.com/software/websphere/events_1.html
- Join the Global WebSphere User Group Community: www.websphere.org
- Access key product show-me demos and tutorials by visiting IBM Education Assistant: www.ibm.com/software/info/education/assistant
- Learn about the Electronic Service Request (ESR) tool for submitting problems electronically: www.ibm.com/software/support/viewlet/probsub/ESR_Overview_viewlet_swf.html
- Sign up to receive weekly technical My support emails: www.ibm.com/software/support/einfo.html

Questions and Answers