JUDCon

JBoss Users & Developers Conference

Boston:2011

Zen of Modules:

Class Loading in JBoss AS 7

David M. Lloyd
Senior Software Engineer, Red Hat, Inc.

What is a Class?

- The object representing a class, interface, enum, annotation, or primitive marker type which has been *loaded* into the JVM
- Consumes memory in both the regular heap and the permanent generation
- Represented on the file system by ".class" files
- Class names are unique per class loader
- Class instances keep a strong reference to their *initiating* ClassLoader instance

What is a ClassLoader?

- Instances are responsible for loading classes by converting bytes to Class instances
- Keeps a strong reference to all Class instances that are loaded by it, as a unique mapping of class name to Class instance
- Has a *parent* which may be used for delegation
- Provides a basis for checking access between classes within a package

Class Loader Basics

- Important Terminology
  - *Initiating Class Loader* - The class loader which is responsible for loading a Class (clazz.getClassLoader())
  - *Parent Class Loader* - In a hierarchical model, the class loader to which unsatisfied requests are delegated
  - *System Class Loader* - The default delegation parent for new class loader instances
  - *Application Class Loader* - The class loader which is defined when invoking the **`java`** command

Class Loader Basics

- More Important Terminology
  - A class is *loaded* (a.k.a. *defined*) by a class loader via the ClassLoader.defineClass() method
  - A class is *linked* (a.k.a. *resolved*) via the ClassLoader.resolveClass() method
  - A class is *initialized* when it is used at runtime, or via the Class.forName(name, **true**, classLoader) method

Understanding Uniqueness

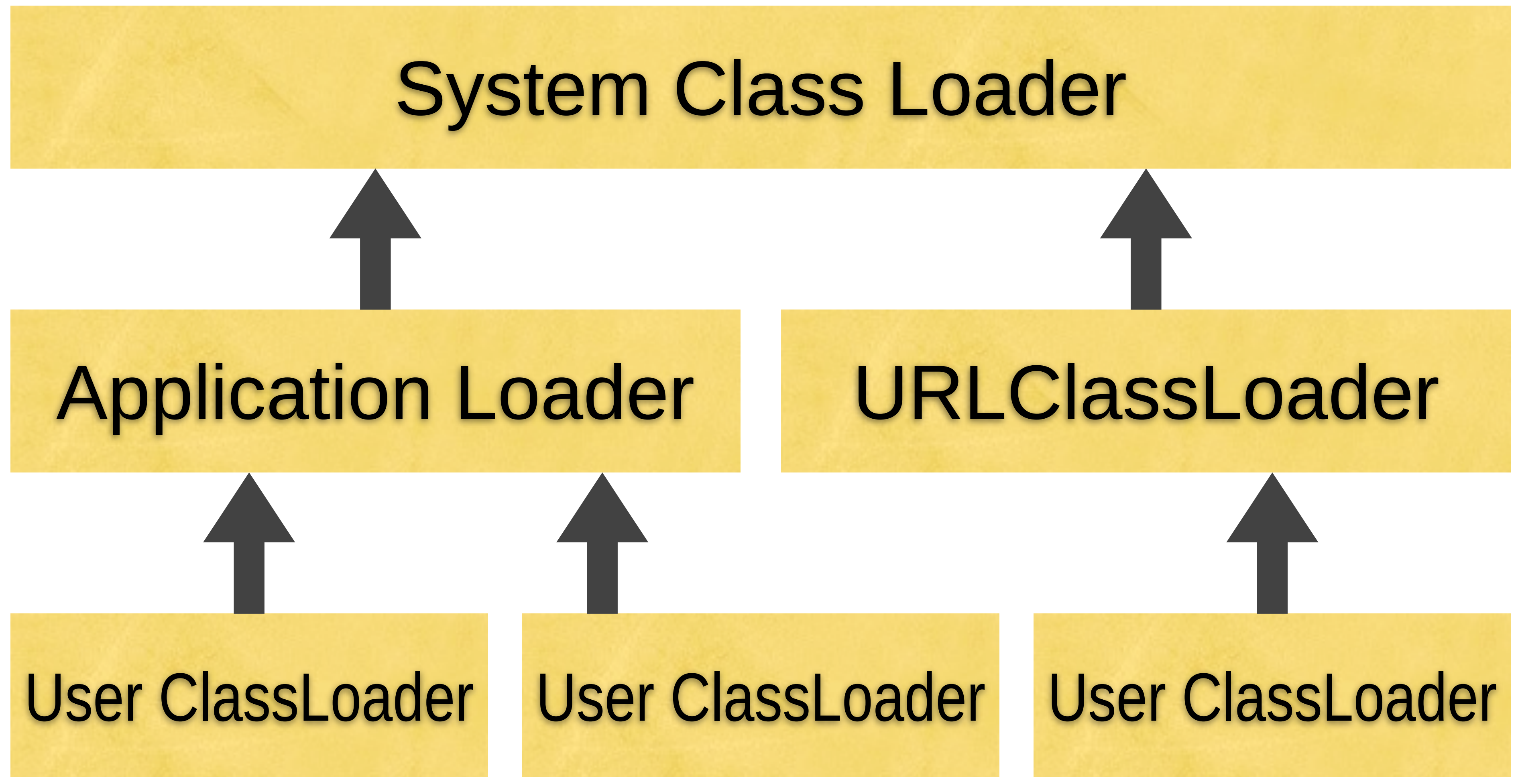
```
URLClassLoader cl1 =
   new URLClassLoader(foo1Jar, null);
URLClassLoader cl2 =
   new URLClassLoader(foo2Jar, null);

Class<?> fooClass1 = cl1.loadClass("Foo");
Class<?> fooClass2 = cl2.loadClass("Foo");

fooClass1.equals(fooClass2) // FALSE!
fooClass1.isAssignableFrom(fooClass2) // FALSE!
fooClass2.isAssignableFrom(fooClass1) // FALSE!
```

Class Loader Basics

- Hierarchical Model, a.k.a. "parent-child delegation"
  - Each Class Loader has exactly one parent (often shared)
  - Standard Java Model
  - Classes in child class loaders can "see" parent but not siblings or children



Problems with Hierarchical Model

- Difficult to share classes between class loaders
  - Shared classes must be on the common parent of all interested class loaders
  - Or, additional delegation step must be introduced
    - Danger! Mixing locking between hierarchies risks deadlock unless carefully planned (many bugs at bugs.sun.com)
- Encourages large, monolithic class loaders which contain everything
  - Cannot load JARs on demand

What is JBoss Modules?

- A standalone implementation of *modular* (non-hierarchical) class loading and program execution
- A *module* is a named set of classes (typically just a single JAR) coupled with information about what other modules it uses, as well as what classes and resources it exports, usually in the form of an XML descriptor file
- JBoss Modules is **not** a container; more like a class loader environment bootstrap which just runs a main() method within a modularized environment

What is JBoss Modules?

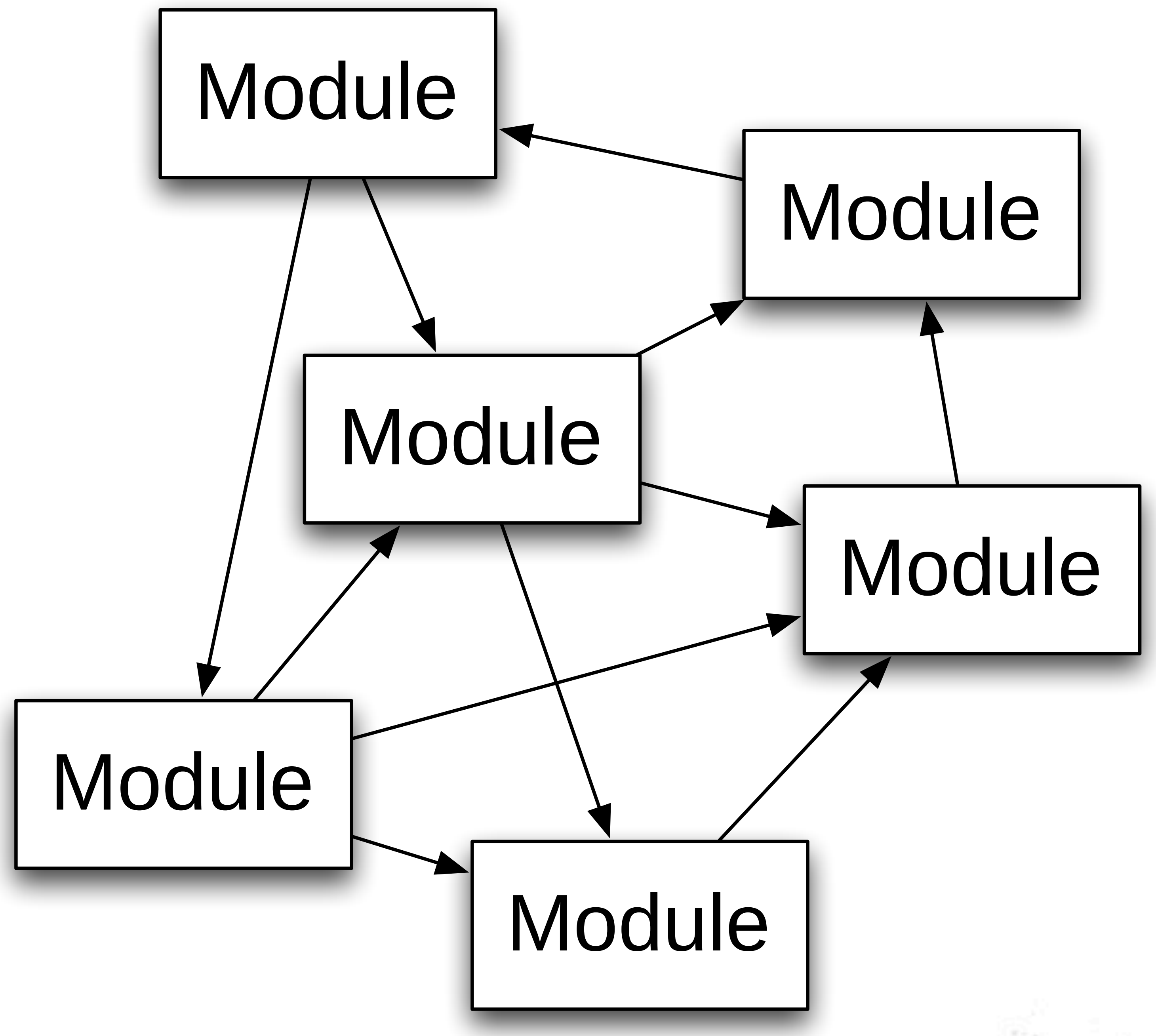
- Module names are dot-separated, a bit like package names or Maven group IDs. Examples:

```
org.jboss.shrinkwrap.api
org.apache.xalan
org.dom4j
```

- A module is typically stored on the filesystem in a directory whose name is derived from the module name as a simple XML descriptor alongside one or more JARs or directories
- Each module has a unique corresponding ClassLoader instance

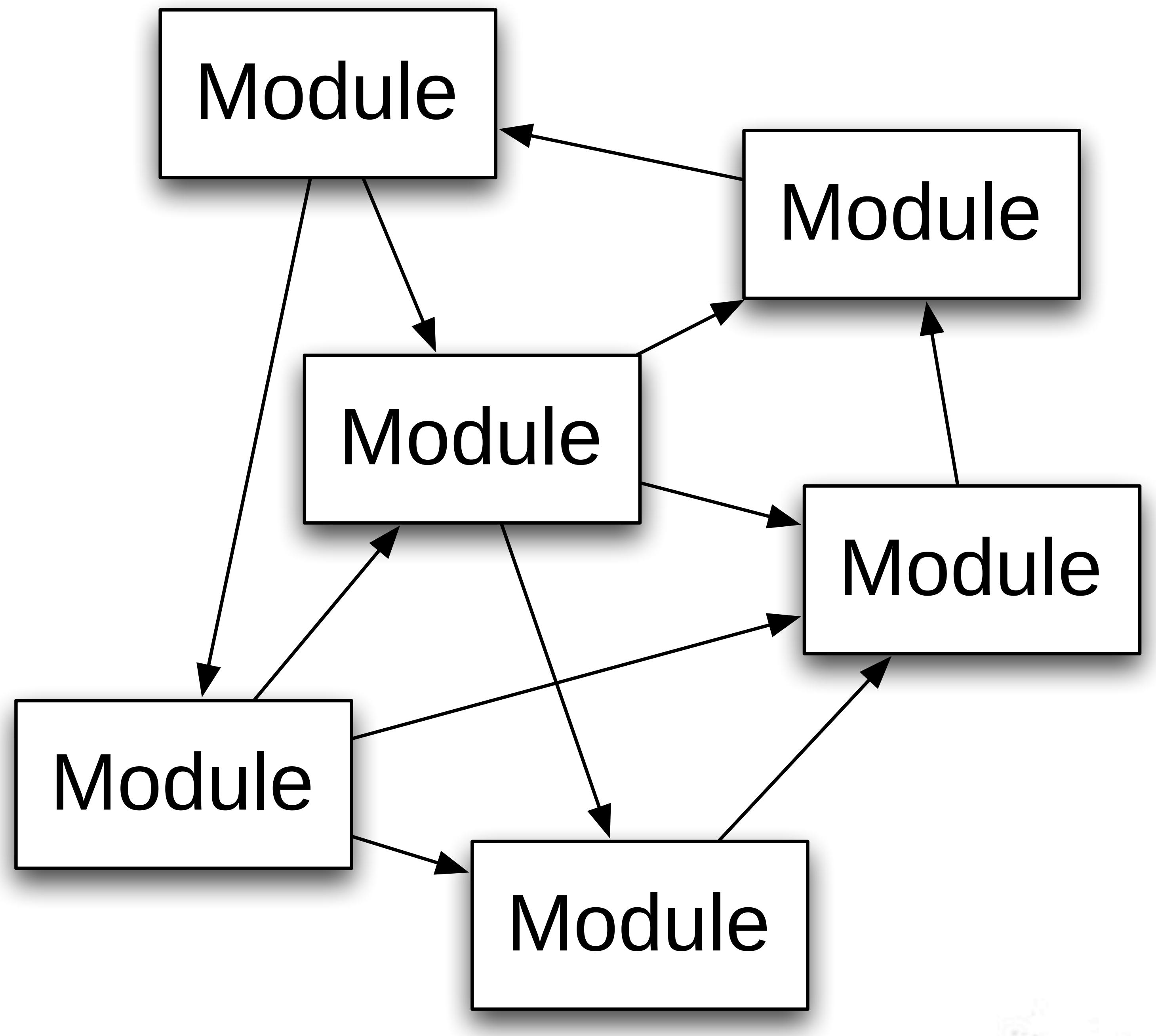
Module Delegation

- Modules delegate to one another as peers (no parents)
- A graph, not a tree
- Much like how multi-module projects are configured in an IDE
- JDK classes are a Module too
- All modules “see” `java.*`



Module Delegation

- No more "big ball of mud"
- Every module is isolated from every other module
- A module imports only the modules that it directly uses (and does not "see" classes or resources that it does not use)
- In particular, modules do not normally "see" their **transitive** dependencies



Why Modules in JBoss AS?

- Lightweight - very small memory footprint per module
  - No extra caches or blacklists - module class loaders need only what the JDK provides
- Simplified Model
  - No domains or shared caching
  - Exceptions, logging, and structure are much less cryptic
- Only load what you need
- Concurrency - allows classes to be loaded from multiple threads at once
- Fast, fast, fast - O(1) class resolution

Modules in JBoss AS

- Every JAR shipped with JBoss AS 7 is a module and can be found in the `jboss-7.0.xx/modules` directory
- Internally, the standalone server is booted simply by executing the org.jboss.as.standalone module (with a couple extra command-line options)

```
java -jar jboss-modules.jar [...] org.jboss.as.standalone
```

- However this is normally encapsulated by our “standalone.sh” or “standalone.bat” scripts

Modules and JavaEE Deployments

- In AS 7+, every *deployment* consists of one or more *modules*
- Thus JavaEE modules are implemented as, well, modules
- JavaEE specification specifies that some deployment types employ so-called “child-first” class loading
  - Implemented simply by loading from the module itself before its imports rather than after

Modules and JavaEE Deployments

- Each deployment module has a set of *implicit imports*
  - Java EE APIs
  - Java SE APIs
  - JBoss Modules APIs
- Each deployment module has a set of *optional imports*
  - Logging APIs
  - JBoss-specific APIs (Infinispan, Remoting, VFS, etc.)
  - User-installed modules

Modules and JavaEE Deployments

- To add a module dependency, use a simple directive in MANIFEST.MF:

```
Dependency: org.apache.commons.logging
```

- In addition, Class-Path and Extension-List references are treated as modular imports of deployments and installed extensions
  - Sibling deployments
  - Arbitrary external JARs

Common Problems

- Many of the same kinds of problems are possible as before, due to the nature of class loading
- New tools to identify and solve problems cleanly

Common Problem #1
OutOfMemoryError: PermGen Space

- At run time?
  - Overuse of String.intern()
  - Overuse of bytecode generation libraries or user-created class loaders (or both)
- On first boot?
  - Too many classes loaded; they don’t fit
  - Solution: adjust sizing of permanent generation via configuration

Common Problem #1
OutOfMemoryError: PermGen Space

- After a few redeploys?
  - Probably a memory leak!
  - Deployment will create new modules and release references to old modules to be GC’d
  - If a class from another class loader keeps a reference (directly or indirectly) to any deployment class or instance, none of the classes in the deployment can be GC’d, and thus are leaked

Common Problem #1

OutOfMemoryError: PermGen Space

Common Leak Causes

- Static field in another deployment class containing an instance of the removed deployment class
- Frameworks which maintain caches
- Persistent Thread-locals
  - Automatic cleaning is very infrequent; threads are often long-lived (especially in thread pools)
  - Thread pools in AS7 will generally auto-clean thread locals more aggressively
- Framework bugs
  - Old versions of Apache Commons Logging and other frameworks

Common Problem #2

ClassNotFoundException

- Or sometimes, NoClassDefFoundError
- Occurs when a class is not visible to the **loading** module
- CNFE when loading a missing class using Class.forName()
- NCDFE when a linking class has a missing dependency
- Most common cause: missing Class-Path, Dependency, or Extension-List entry
  - The EE spec requires that portable applications use Class-Path to express dependencies between **most** sibling modules within an application (EAR)!

What???

- It’s true!
- In particular, by spec, a WAR cannot normally see:
  - EJB JARs or other WARs, even in the same EAR
  - Vendor-specific APIs
- Likewise, an EJB JAR cannot normally see:
  - WARs or other EJB JARs, even in the same EAR
  - Vendor-specific APIs

Common Problem #3

ClassCastException

- Sometimes, you get a ClassCastException between two types of the same name
- Because the two classes come from different modules (i.e. it is duplicated between them)
- Usually a packaging problem
- Common scenario: bundling EJB local interfaces into WARs within an EAR
  - The WAR's version is passed to the EJB which has the EAR's version
  - The classes come from different modules, so CCE

Common Problem #3

ClassCastException

- Best Practice
  - Look for nested JARs which are duplicated between your EAR and WARs
  - Remove extra copies and use Class-Path instead
  - Or, factor out common libraries into external JARs or modules, and use Class-Path or Dependency in your MANIFEST.MF to load them
- Other Solutions
  - If you require that the versions differ, for some reason, use @Remote (and make sure call-by-reference is disabled)

Modules and OSGi

- The JBoss OSGi implementation shipping with JBoss AS 7 uses JBoss Modules at its core
- Bundles are Modules
- Allows new levels of interaction between JavaEE and OSGi
- Excellent performance

Modules and Future Java EE

- JSR-294 and Project Jigsaw hope to bring modularity directly into the Java language by Java SE 8
- It is likely that a future Java EE specification will “go modular”

Modules and You

- See direct performance benefits in your Application Server deployments
- Make some sense of the JBoss AS class loading architecture
- Later session (2:30pm today) focuses on using JBoss Modules for your applications

Q & A

JUDCon

JBoss Users & Developers Conference

Boston:2011