



# JavaOne™

[java.sun.com/javaone](http://java.sun.com/javaone)

## MODULARITY IN THE JAVA™ PLATFORM

Alex Buckley  
JSR 277 spec lead

Stanley M. Ho  
JSR 277 spec lead

TS-6185



Learn how the Java Module System makes organizing large programs easy and supports flexible deployment options in the Java Platform, Standard Edition 7.

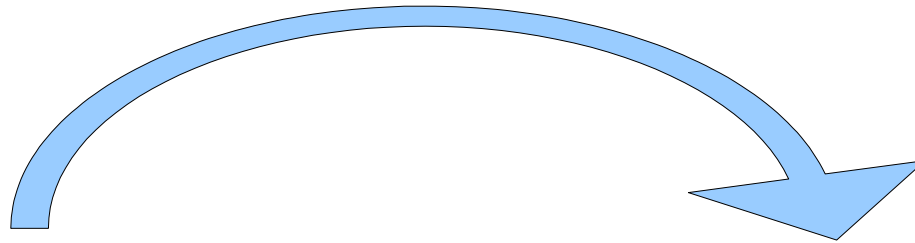


GOAL

# Agenda

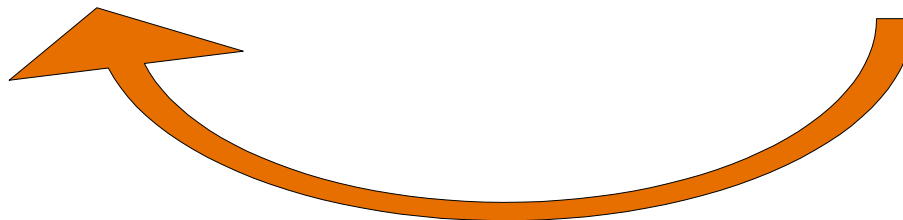
- Goals of the Java Module System
- Development with the Java Module System
- Deployment with the Java Module System
- OSGi in the Java Module System
- Closing remarks

More classes to deploy

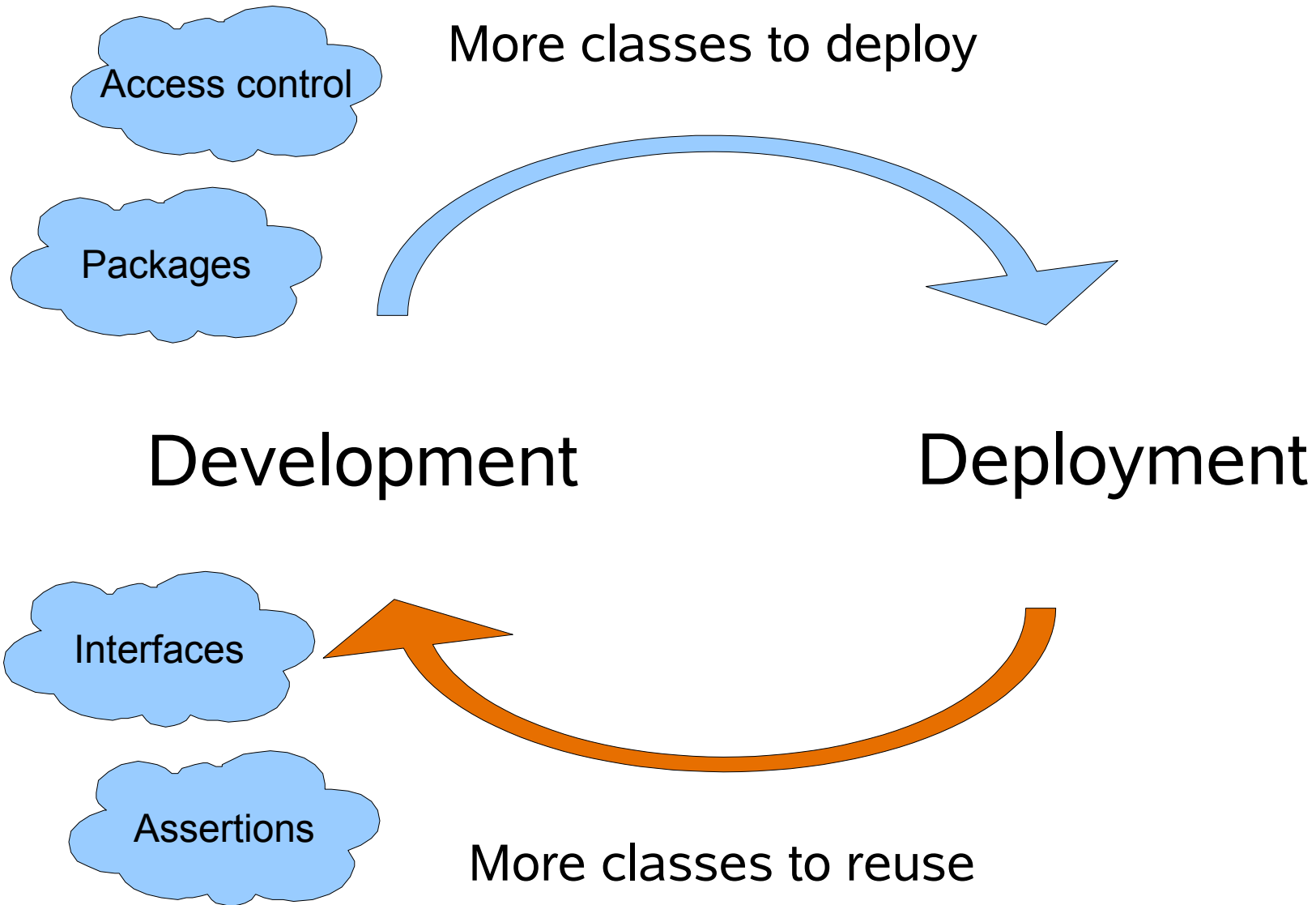


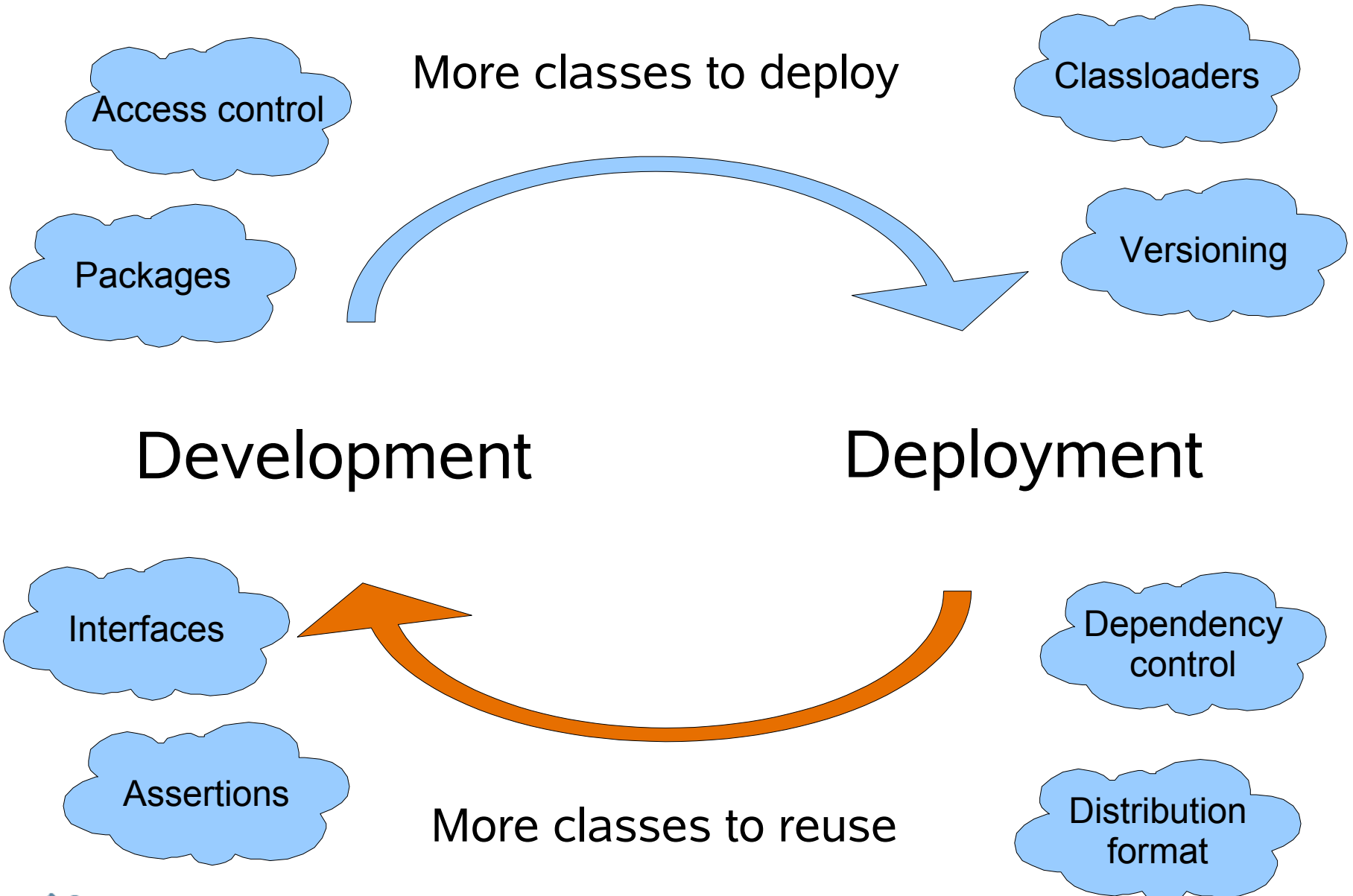
Development

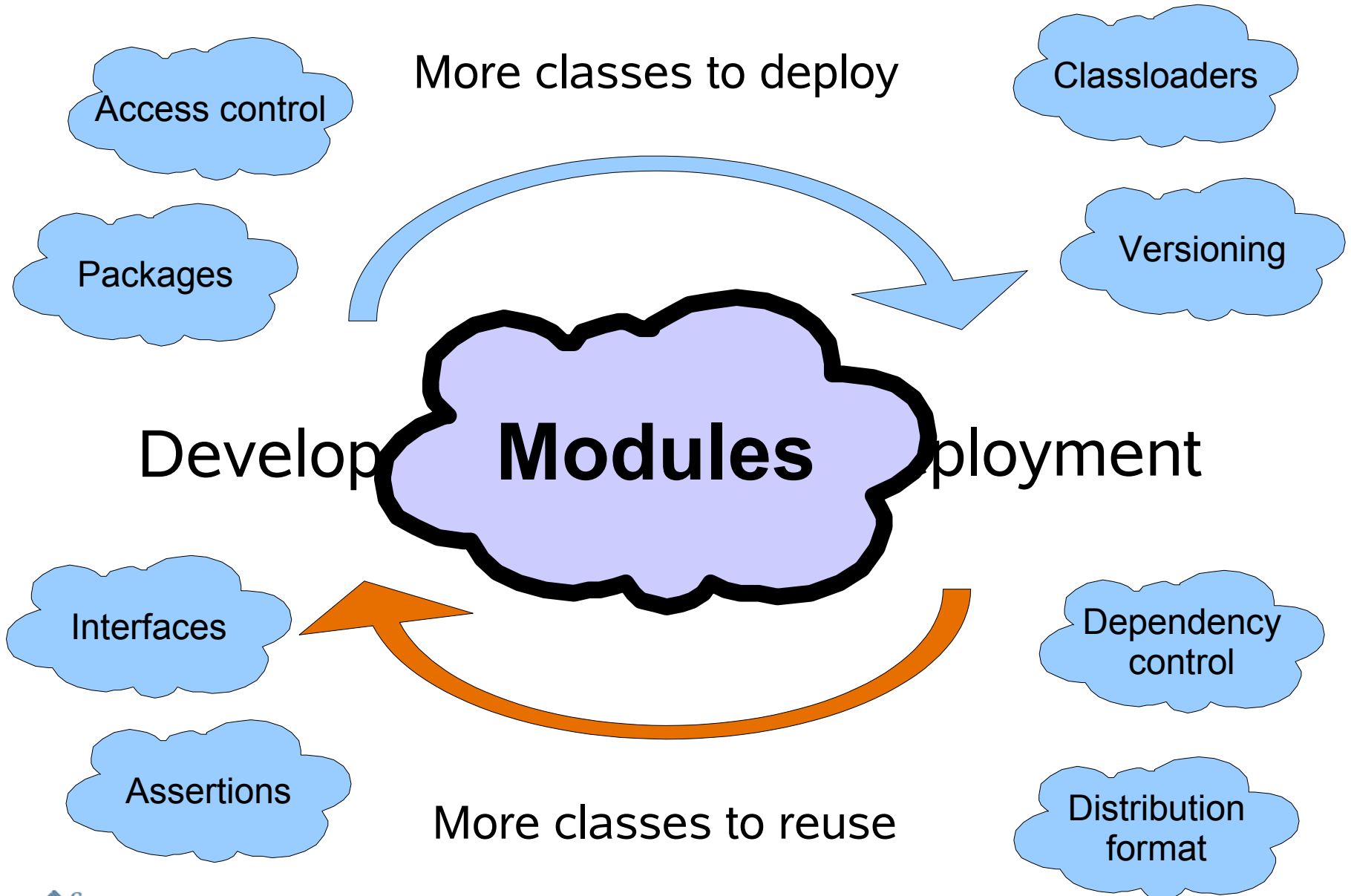
Deployment



More classes to reuse







Access control

Packages

Promote modular programming

First-class module concept in language & VM  
Reflective module metadata

## Development goals for the Java Module System

Interfaces

Support JDK™ software modularization

Platform Profiles

Endorsed Standards Override Mechanism

Assertions

## Address "JAR hell"

- Better distribution format than JAR files
- Enable side-by-side versioning
- Simple and predictable runtime model

Classloaders

Versioning

# Deployment goals for the Java Module System

- Support entire spectrum of Java programs
- Repository infrastructure to isolate modules
- Customizable for different environments  
(e.g. applets) and module systems

Dependency control

Distribution format

# JSR 277 "Java Module System"

- JSR 294 owned development modules ("superpackages")
- All modularity discussions now happen in JSR 277
  - Development modules in the Java language
  - Deployment module framework
  - Default deployment module system: The JAM Module System

# Agenda

- Goals of the Java Module System
- Development with the Java Module System
- Deployment with the Java Module System
- OSGi in the Java Module System
- Closing remarks

# Modular programming in Java today

## ➤ Packages

- Package names are hierarchical
- Package membership is not

## ➤ Access control

- Types and members shared across packages must be made public
- Hope no-one finds your "internal" packages
- Rely on comments/documentation to describe "official" APIs

## ➤ Interfaces

- Not always desirable to have all members be public

# A typical package hierarchy

```
org/  
  netbeans/  
    core/  
      Debugger.class  
      ...  
    utils/  
      ErrorTracker.class  
      ...  
    wizards/  
      JavaFXApp.class  
      ...  
  addins/  
    ...
```

# Classes in different packages need to collaborate

org/

netbeans/

core/

Debugger.class

...

utils/

ErrorTracker.class

...

wizards/

JavaFXApp.class

...

addins/

...

org.netbeans.core is an obvious "unit"

org/

netbeans/

core/

```
Debugger.class
```

```
...
```

```
utils/
```

```
    ErrorTracker.class
```

```
    ...
```

```
wizards/
```

```
    JavaFXApp.class
```

```
    ...
```

```
addins/
```

```
...
```

# org.netbeans.core is a conceptual "module"

```
org/  
  netbeans/  
    core/  
      Debugger.class  
      ...  
      utils/  
        ErrorTracker.class  
        ...  
        wizards/  
          JavaFXApp.class  
          ...  
      addins/  
        ...
```

# Modules in the Java language

Module  
concept in the  
language

```
// org/netbeans/core/Debugger.java
module org.netbeans.core;
package org.netbeans.core;
public class Debugger {
    ... new ErrorTracker() ...
}
```

# Modules in the Java language

Module  
concept in the  
language

One module  
has many  
packages

Module  
access  
specified in  
the language

```
// org/netbeans/core/Debugger.java
module org.netbeans.core;
package org.netbeans.core;
public class Debugger {
    ... new ErrorTracker() ...
}
```

```
// org/netbeans/core/utils/ErrorTracker.java
module org.netbeans.core;
package org.netbeans.core.utils;
module class ErrorTracker {
    module int getErrorLine() { ... }
}
```

# Modules in the Java language

Module concept in the language

```
// org/netbeans/core/Debugger.java
module org.netbeans.core;
package org.netbeans.core;
public class Debugger {
    ... new ErrorTracker() ...
}
```

One module has many packages

```
// org/netbeans/core/utils/ErrorTracker.java
module org.netbeans.core;
package org.netbeans.core.utils;
module class ErrorTracker {
    module int getErrorLine() { ... }
}
```

Module access specified in the language

Module dependencies specified in the language

```
// org/netbeans/core/module-info.java
@Version("7.0")
@ImportModule(name="java.se.core", version="1.7+")
module org.netbeans.core;
```

# Compiling and running a Java module

- `javac org/netbeans/core/*`
- `javac org/netbeans/core/Utils/*`
- `java org.netbeans.core.Debugger`

# Impact of modules in the language & VM

- **module** restricted keyword
- **module-info.java** for module-level annotations
- **package-info.java** can declare module membership
- Classfile attribute for module membership
- Classfile flag for "module-private" accessibility
- Module-private accessibility enforced by the Java Virtual Machine
- javadoc and javap understand modules in .java and .class files

# Agenda

- Goals of the Java Module System
- Development with the Java Module System
- Deployment with the Java Module System
- OSGi in the Java Module System
- Closing remarks

## Address "JAR hell"

Better distribution format than JAR file  
 Enable side-by-side versioning  
 Simple and predictable runtime model

Classloaders

Versioning

# Deployment goals for the Java Module System

Support entire spectrum of Java programs  
 Repository infrastructure to isolate modules  
 Customizable for different environments  
 (e.g. applets) and module systems

Dependency control

Distribution format

# The JAM Module System

- JAVa Modules
- A standard file format and predictable runtime model across Java SE 7 implementations
  - JAR-like file format
  - Easy, reflective, extensible metadata
  - Deterministic module resolution algorithm to address JAR hell
  - Deterministic module initialization algorithm
  - Integrated in platform, tools, class libraries ...
- JAM modules are simple to read and write
  - Addresses 80% of the problem space, not 95%
  - You can make a system better by keeping things out
  - Once a feature is added, it can never be removed
  - If a feature is left out, it can always be added later

# JAM metadata

Version

```
// org/netbeans/core/module-info.java
@Version("7.0")
```

Main class

```
@MainClass("org.netbeans.core.Main")
```

Attributes

```
@Attribute(name="IDE", value="NetBeans")
```

Module dependencies

```
@ImportModules{
    @ImportModule(name="java.se.core",
                  version="1.7+"),
    @ImportModule(name="org.foo.util",
                  version="1.0", reexport=true)
```

Custom import policy

```
})
```

Initializer

```
@ImportPolicyClass("org.netbeans.CustomPolicy")
```

Platform specific

```
@ModuleInitializerClass("org.netbeans.core.Init")
```

Exported resources

```
@PlatformBinding(os="solaris", arch="sparc")
```

```
@ExportResources({"icons/**"})
```

```
module org.netbeans.core;
```

# Compiling and running a JAM module

- `javac org/netbeans/core/*`
- `javac org/netbeans/core/utils/*`
- `jam cvf org.netbeans.core-7.0.jam  
org/netbeans/core/*  
org/netbeans/core/utils/*`
- `java -jam org.netbeans.core-7.0.jam`
- `cp org.netbeans.core-7.0.jam /netbeans/repository`
- `java -module org.netbeans.core:6.0+  
-repository /netbeans/repository`

# org.netbeans.core-7.0.jam

`/META-INF/MANIFEST.MF`

Module metadata

`/MODULE-INF/MODULE.METADATA`

`/MODULE-INF/bin/xml-windows.dll`

Native libraries

`/MODULE-INF/bin/xml-linux.so`

`/MODULE-INF/lib/xml-parser.jar`

JAR files

`/org/netbeans/core/module-info.class`

`/org/netbeans/core/Main.class`

`/org/netbeans/core/Debugger.class`

`/org/netbeans/core/utils/ErrorTracker.class`

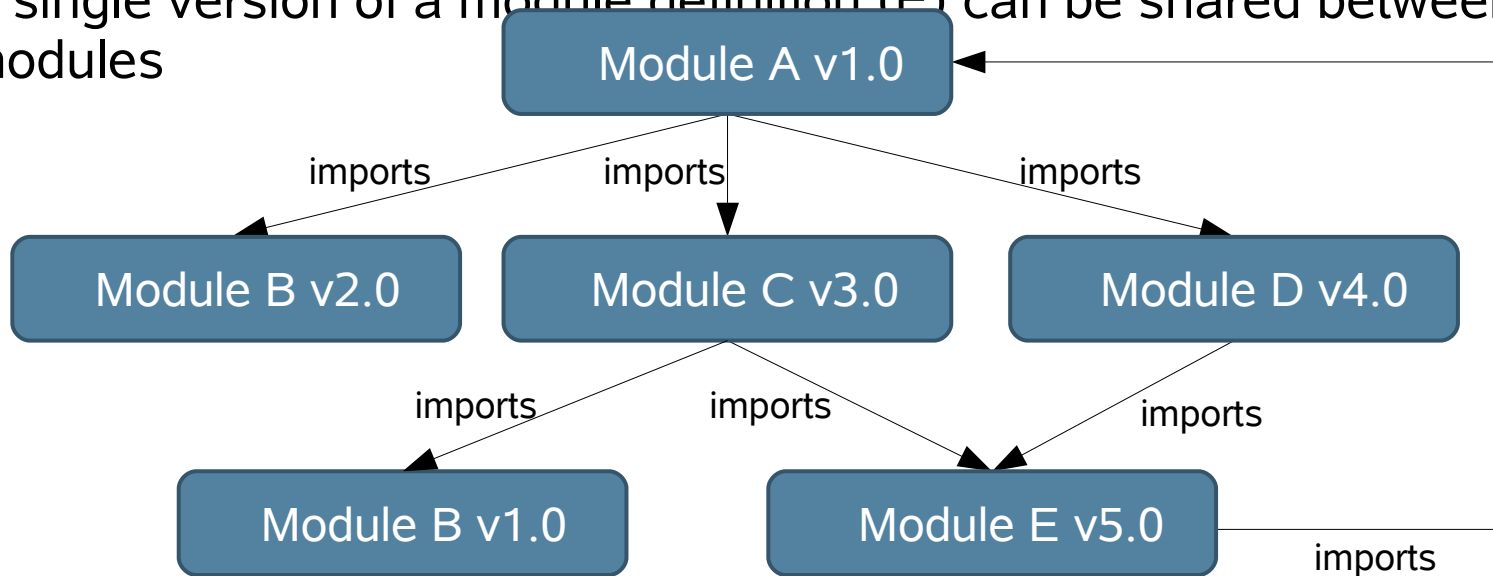
`/icons/graphics.jpg`

Other resources

➤ .jam.pack.gz extension for Pack200-gzipped JAM module

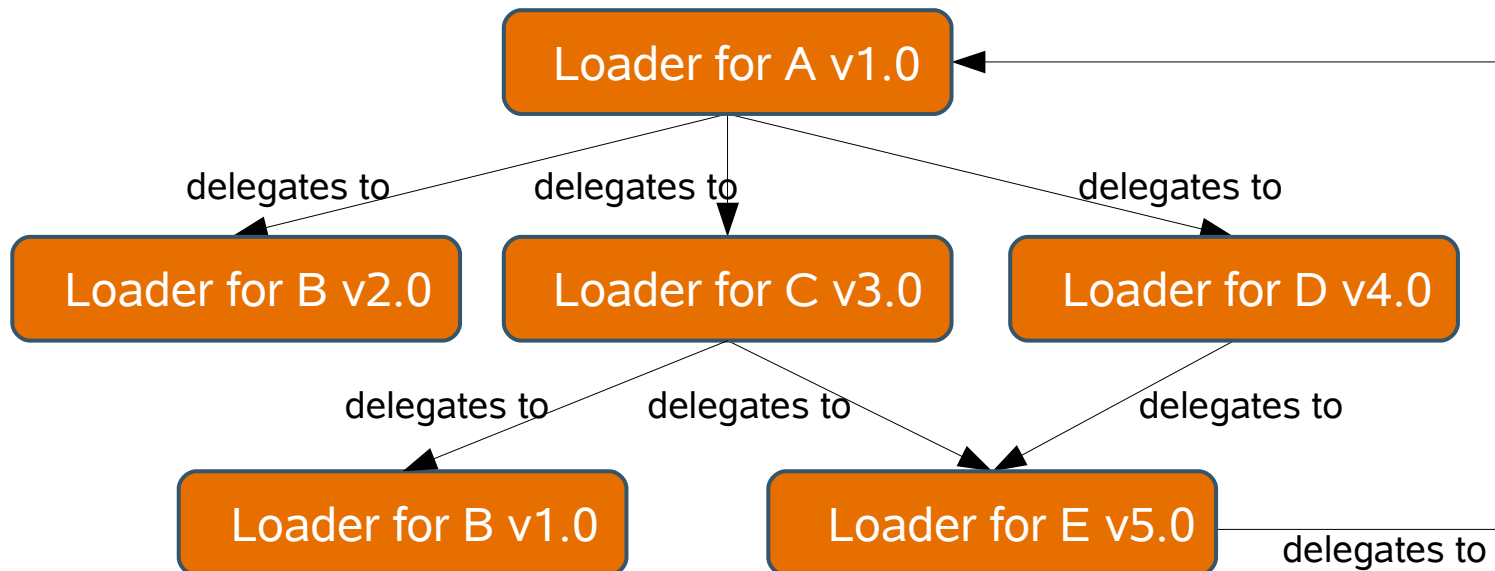
# No more JAR hell

- A JAM module definition imports other JAM module definitions by name
- Module resolution finds module definitions to satisfy imports predictably
- Different versions of the "same" module definition (B) can co-exist
- If different versions of a module definition satisfy an import, use the highest
- A single version of a module definition (E) can be shared between modules



# After a JAM module definition is resolved...

- A JAM module instance is created and initialized
- A JAM module instance has its own classloader
- If JAM module definition A imports JAM module definition B, then A's classloader delegates to B's classloader for classes exported by B
- Classloading is completely deterministic



## Address "JAR hell"

- Better distribution format than JAR files
- Enable side-by-side versioning
- Simple and predictable runtime model

Classloaders

Versioning

# Deployment goals for the Java Module System

**Support entire spectrum of Java programs**

Repository infrastructure to isolate modules  
 Customizable for different environments

Dependency control

Distribution format

# A framework for module systems

## ➤ Recap:

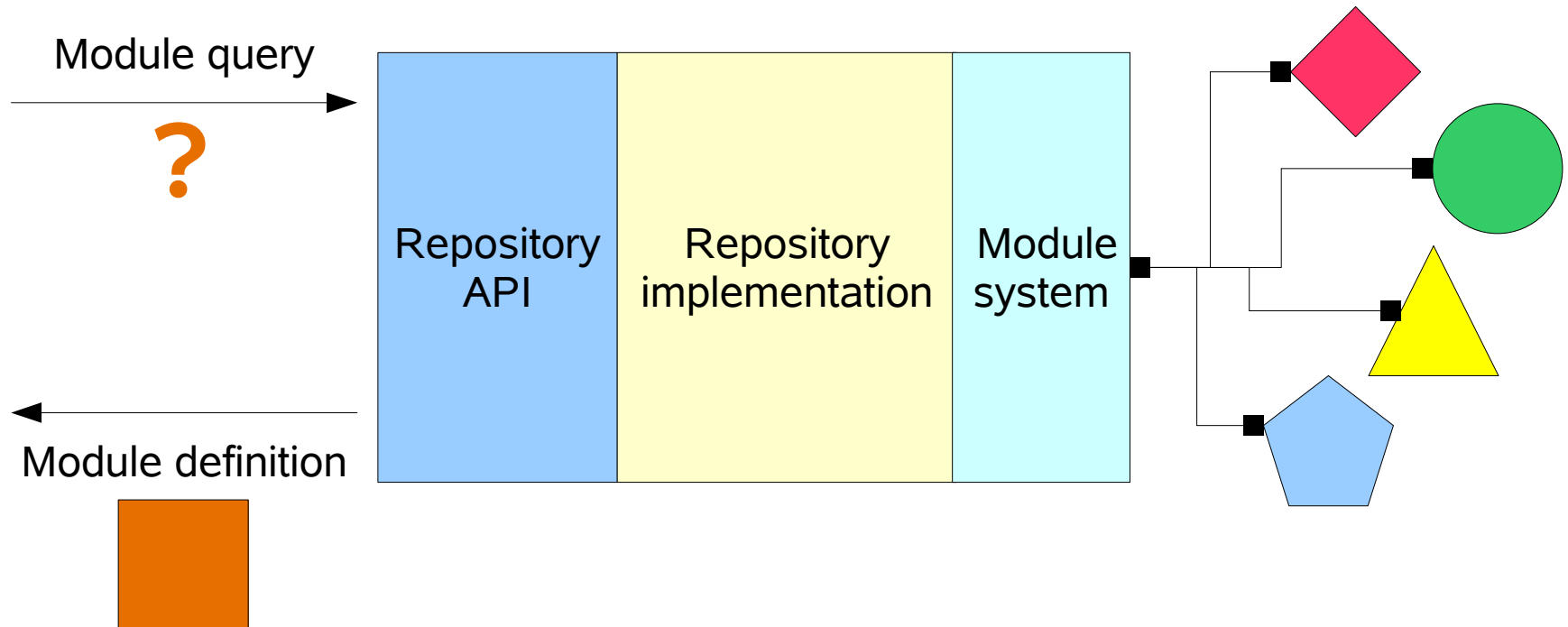
- A JAM module definition is resolved
- A JAM module instance is created with its own classloader

## ➤ In the abstract:

- A module definition is resolved
- A module instance is created with its own classloader

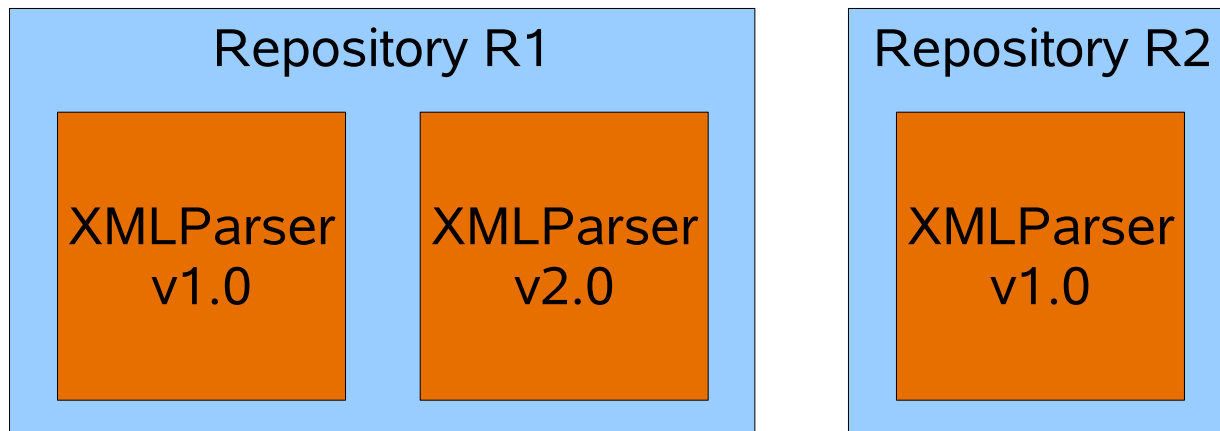
# Where do module definitions come from?

- Module definitions come from repositories
- A repository is a lightweight abstraction for obtaining module definitions regardless of the underlying module system

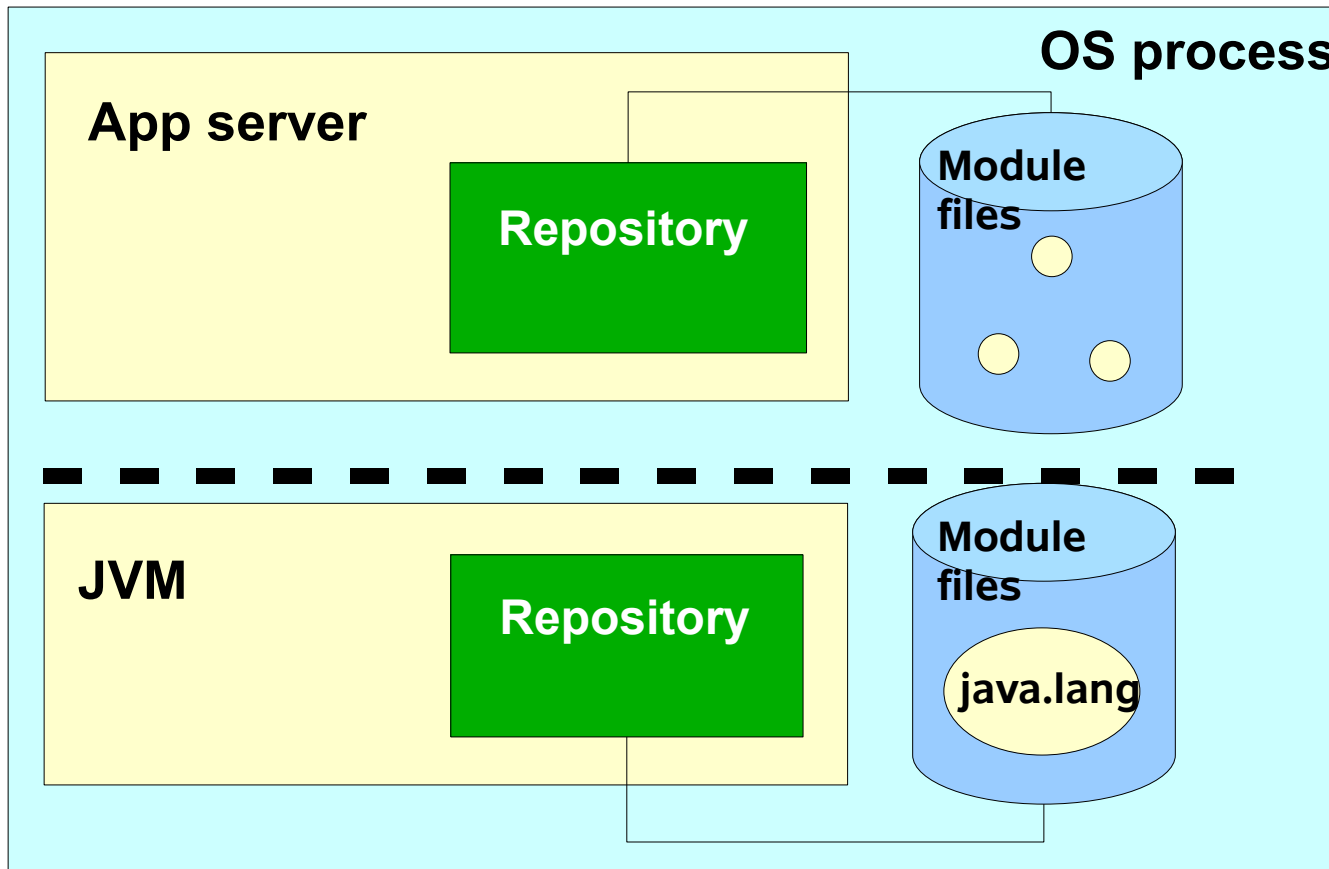


# Repositories isolate module definitions

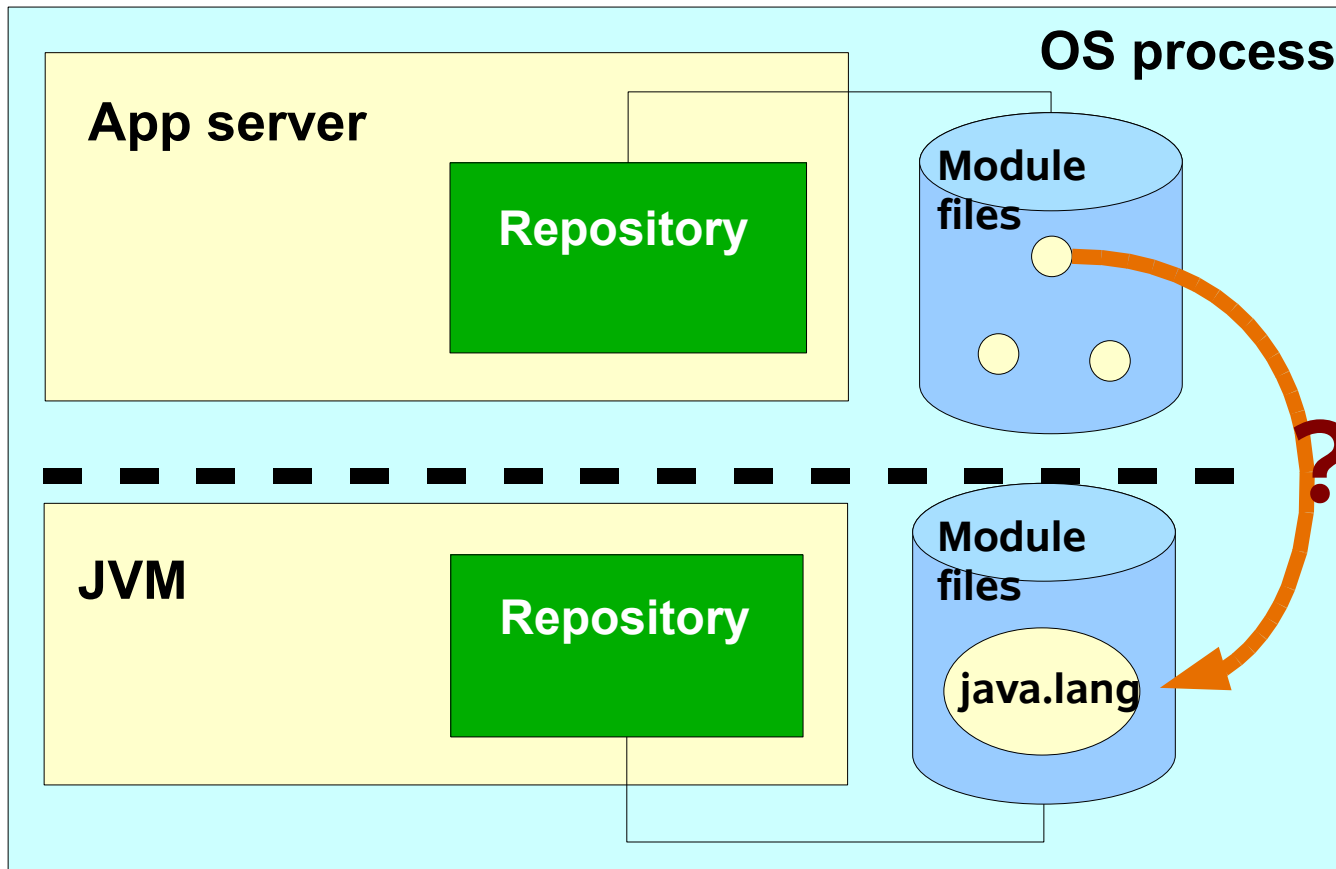
- A class from one module definition is isolated from a class from another module definition



# Simple deployment with repositories

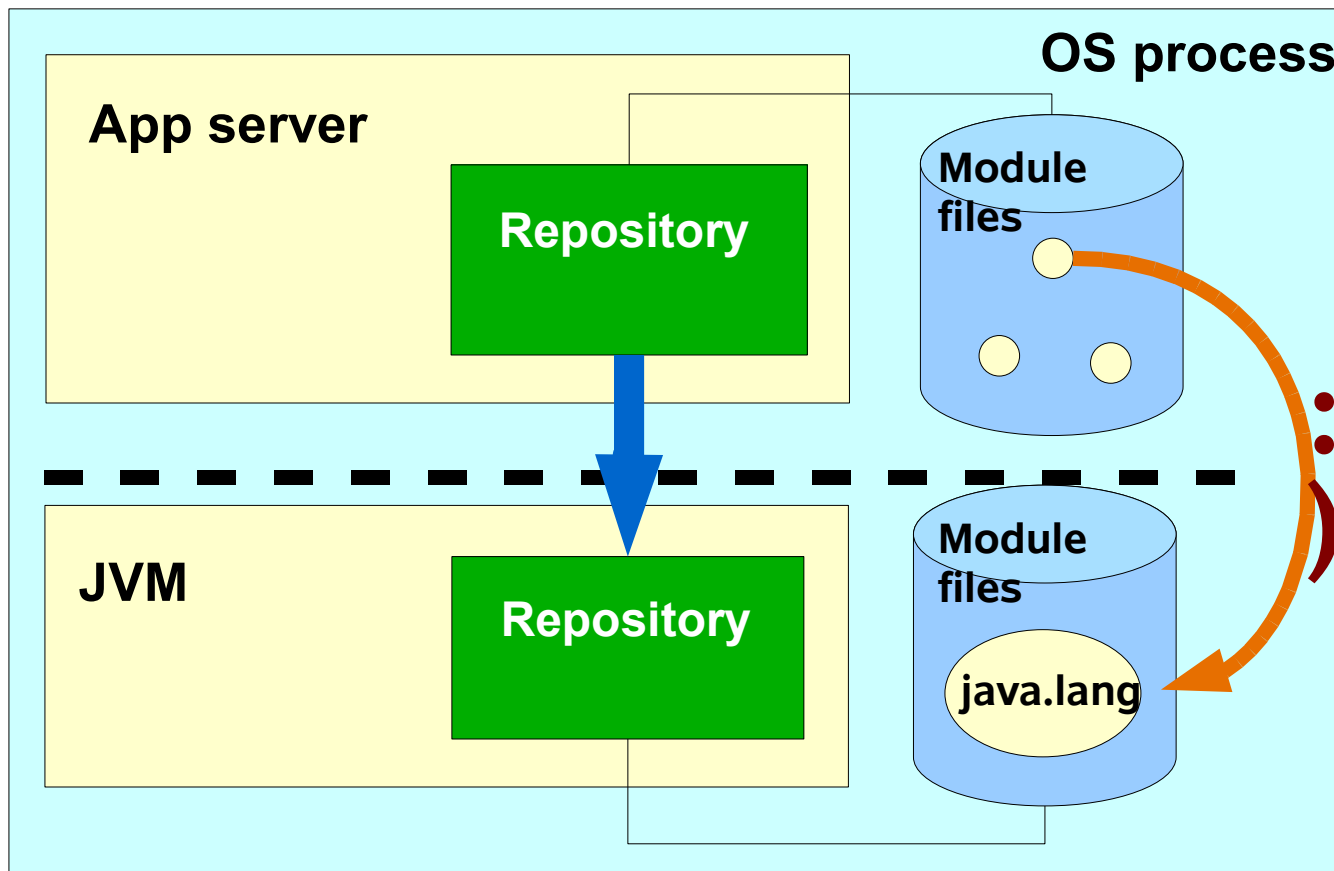


# How can user modules depend on platform modules?

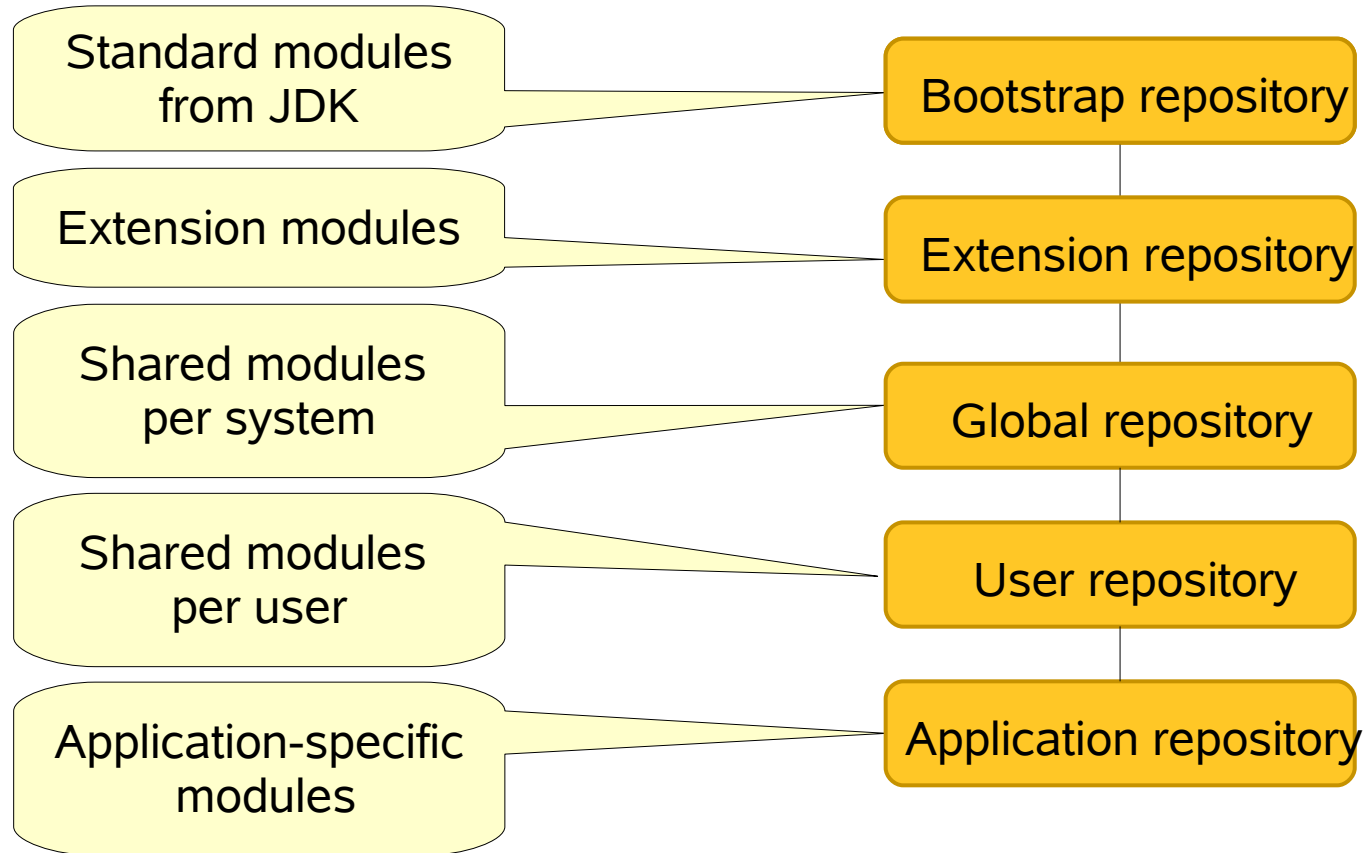


# Repositories must communicate!

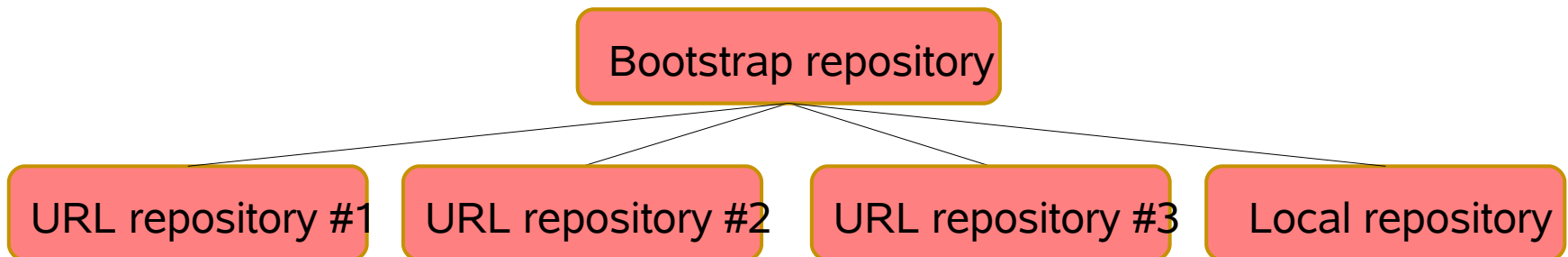
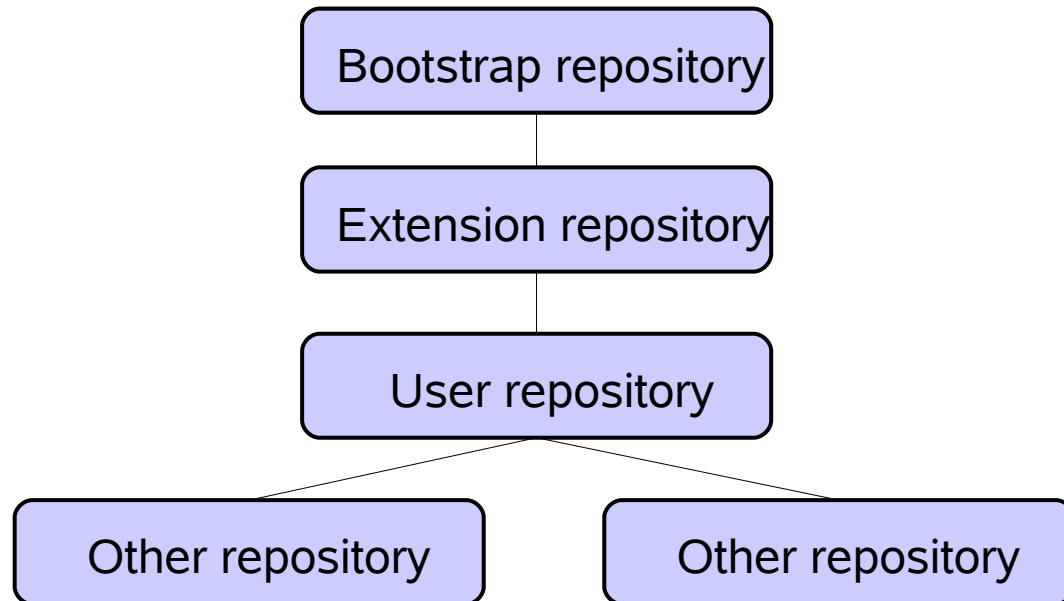
- No existing module system can interoperate with "itself"



# Repository communication in the JDK



# A repository tree isolates different branches



# Agenda

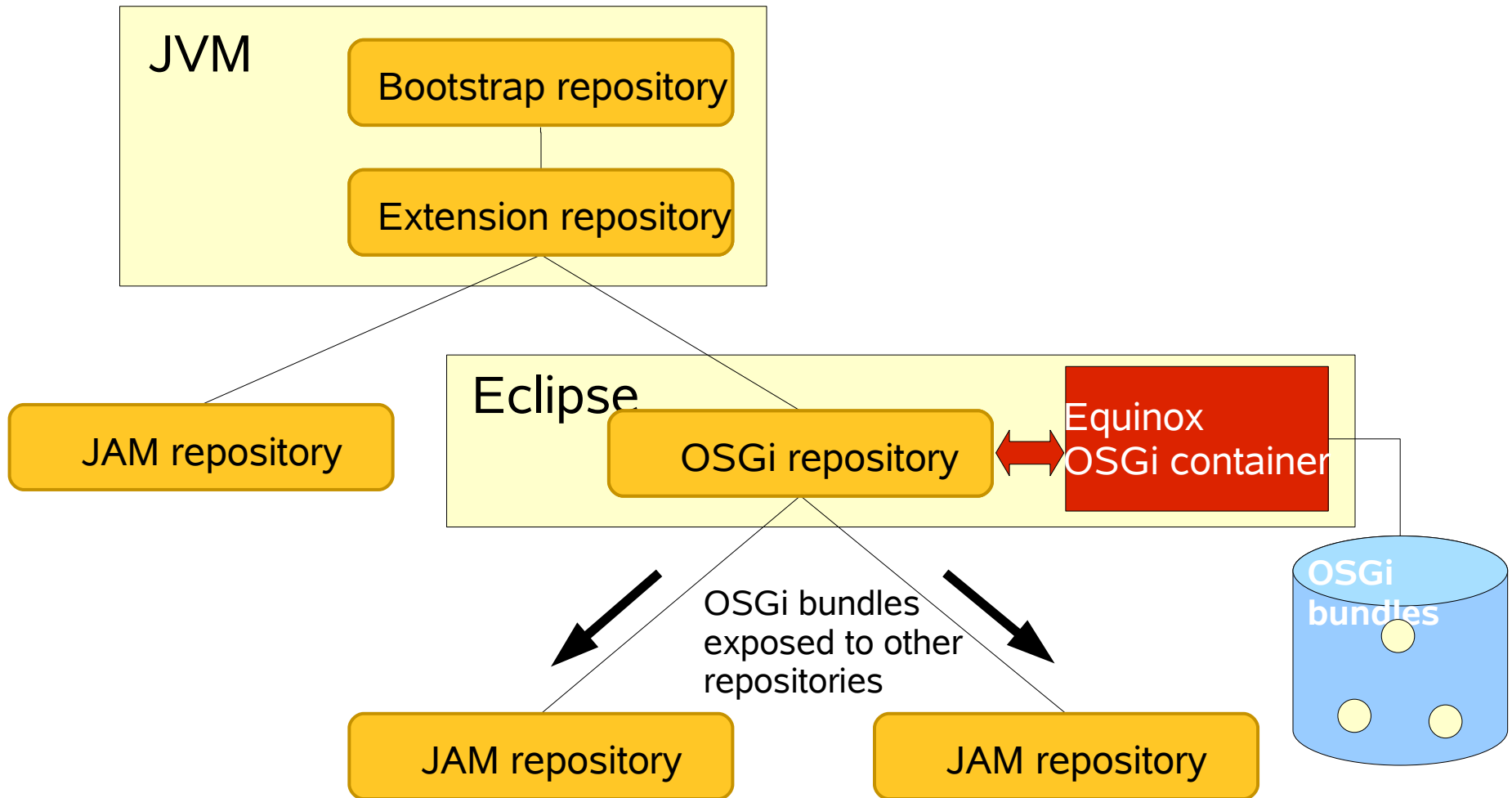
- Goals of the Java Module System
- Development with the Java Module System
- Deployment with the Java Module System
- OSGi in the Java Module System
- Closing remarks

# OSGi in the Java Module System

## ➤ Recap:

- The Java Module System is a framework
  - Defines key abstractions which concrete module systems can support
  - Allows different concrete module systems to co-exist and compete
  - Invents fundamental concepts missing in existing module systems
  - Repository is the key abstraction
- An OSGi container could implement the Repository API
- A JAM module in JDK7 will be able to import an OSGi bundle

# OSGi can implement the deployment framework



# Agenda

- Goals of the Java Module System
- Development with the Java Module System
- Deployment with the Java Module System
- OSGi in the Java Module System
- Closing remarks

# Recap

- JSR 277 defines the Java Module System
- The Java Module System has three components
  - Development modules (**module**)
  - Deployment module framework (Repository)
  - Default deployment module system (JAM)
- Development modules are easy
- Deployment module framework allows the platform's module system to interoperate with other module systems
- The JAM Module System exploits development modules and is simple and predictable

# Status

- JSR 277 Early Draft Review 2 is in progress
  - Spec will be out soon
  - Reference Implementation (OpenJDK™ Modules) around the same time
  - <http://openjdk.java.net/projects/modules/>
- Subscribe to the JSR 277 Expert Group observer list
- [jsr-277-comments@jcp.org](mailto:jsr-277-comments@jcp.org)
- <http://blogs.sun.com/abuckley/>
- <http://weblogs.java.net/blog/stanleyh/>
- Modularity BOF
  - BOF-5032, Wednesday 8.30pm

# THANK YOU



Alex Buckley

Stanley M. Ho

TS-6185

