Modularity in Java

Alex Buckley
Spec lead, Java Language & VM
Sun Microsystems

Devoxx 2008

Sun microsystems

Agenda

- The Modular Java Platform
- JSR 294
- Core modularity features
  - Membership
  - Accessibility
  - Dependencies
  - Versioning
- Further modularity features
  - Re-export
  - Multi-module packages

Problems

- Code modularity
  - JAR files don’t scale — “JAR hell”
    - No dependence or versioning information
    - No encapsulation of internal interfaces
    - No well-defined relationship to native packaging systems
- Platform scalability
  - SE doesn’t scale down from the server/desktop
    - Small devices have the speed to run SE applications, but not the storage to carry the entire SE platform
  - ME applications don’t run on SE
- Performance
  - Download time, startup time, memory footprint
    - Of the JRE itself, and of applications

The Modular Java Platform

Modularize the platform — and applications too

- Enables escape from "JAR hell"
  - Eliminate the class path — once and for all
  - Easily generate sensible rpm/deb/svr4/ips packages
- Enables platform scalability
  - Well-specified SE subsets can fit into small devices
  - ME applications can run on SE (subsets)
- Enables significant performance improvements
  - Less work to start up
  - Incremental download of modules on demand
  - Pre-optimization of module content during install

JSR 294

- Core language and VM features for modularity
  - Java source constructs + ClassFile attributes
  - Sits underneath a module system
- Simple
  - First major language feature since generics
- Friendly
  - Leave all questions of "format" to the module system (version format, file format, repository format, etc)
  - Support OS packaging
- Scalable
  - Support platform decomposition

Language/VM features for modularity

Everything in this presentation is independent of particular module systems

Modular programming in Java today

- Packages
  - > Package names are hierarchical
  - > Package membership is not
- Access control
  - > Types shared across packages must be public
  - > Hope no-one finds your "internal" packages
  - > Rely on comments/docs to describe "official" APIs
  - > (Why is there no package accessibility modifier?)
- Interfaces
  - > Not always desirable to have all members 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

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() { ... }
}
```

Module membership

- To avoid declaring module membership in every source file, use package-info.java

```
//  org/netbeans/core/utils/package-info.java
module  org.netbeans.core;
package org.netbeans.core.utils;
```

- If a source file and the relevant package-info.java file disagree over module membership, compile-time error

Module annotations

- Annotations of package P live in P/package-info.java
- Annotations of module M live in M/module-info.java

```
//  org/netbeans/core/module-info.java
@Foo
@Bar("Quux")
module org.netbeans.core;
```

Compiling and running a Java module

- javac org/netbeans/core/*
- javac org/netbeans/core/utils/*

- java org.netbeans.core.Debugger

No module is an island

- A module is a component of reuse, consisting of one or more packages, with an "interface" represented by its public types
  - A package is not an atomic component of reuse, since auxiliary packages are often needed
  - module-private types enforce distinction between module interface (`public`) and module implementation (`module`)
- A module should specify its dependencies
  - Conceptually, its own private CLASSPATH
  - Under the control of the programmer, not the deployer
  - Foundation for static analysis, optimization, packaging

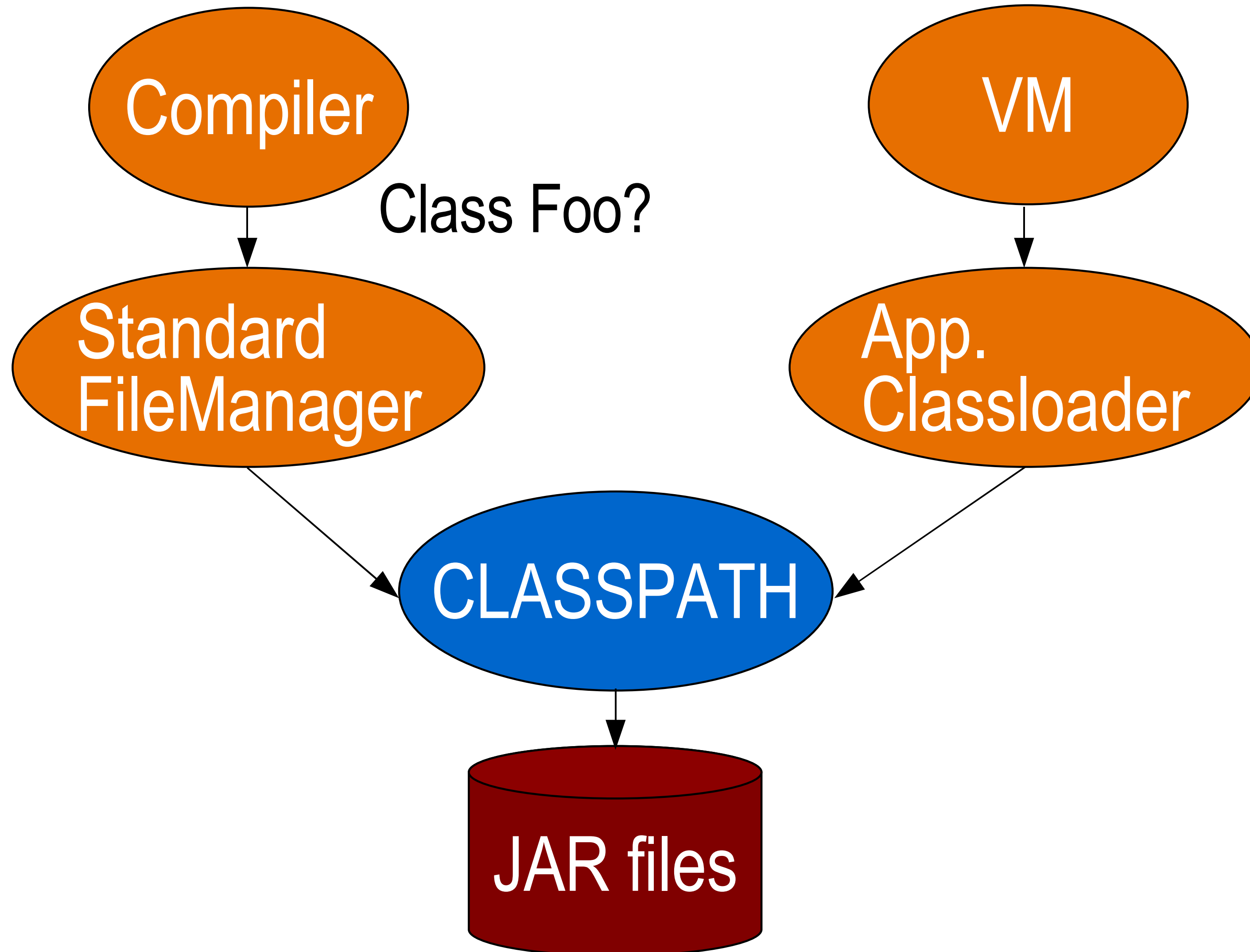
Specifying module dependencies

- Extend module annotation file with metadata:

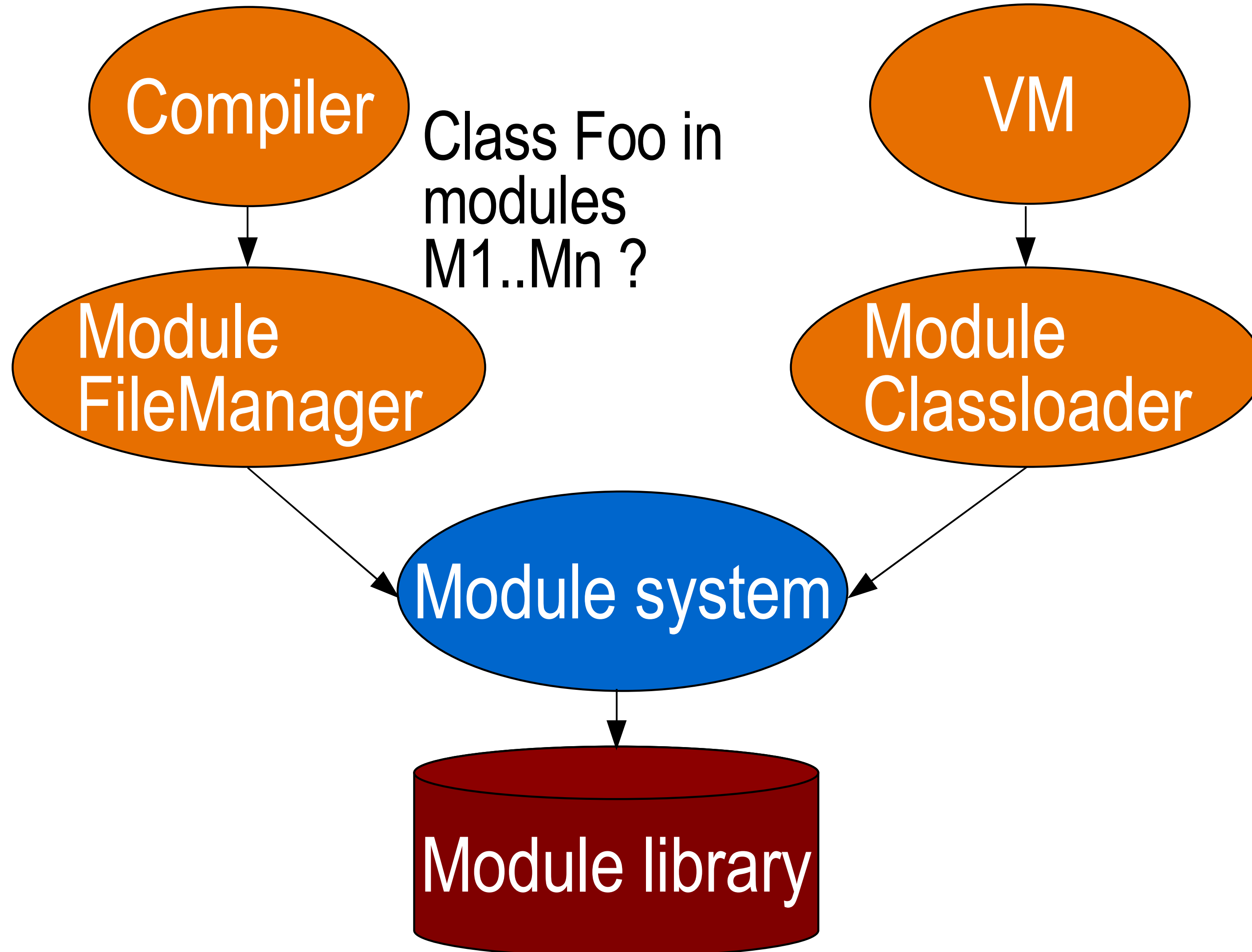
```
//  org/netbeans/core/module-info.java
@Foo
@Bar("Quux")
module org.netbeans.core {
     require org.w3c.dom.parser;
     require java.core;
}
```

- `require` makes the "interface" of the required module observable to the current module
  - For compatibility, code not in a named module sees all public types of all modules

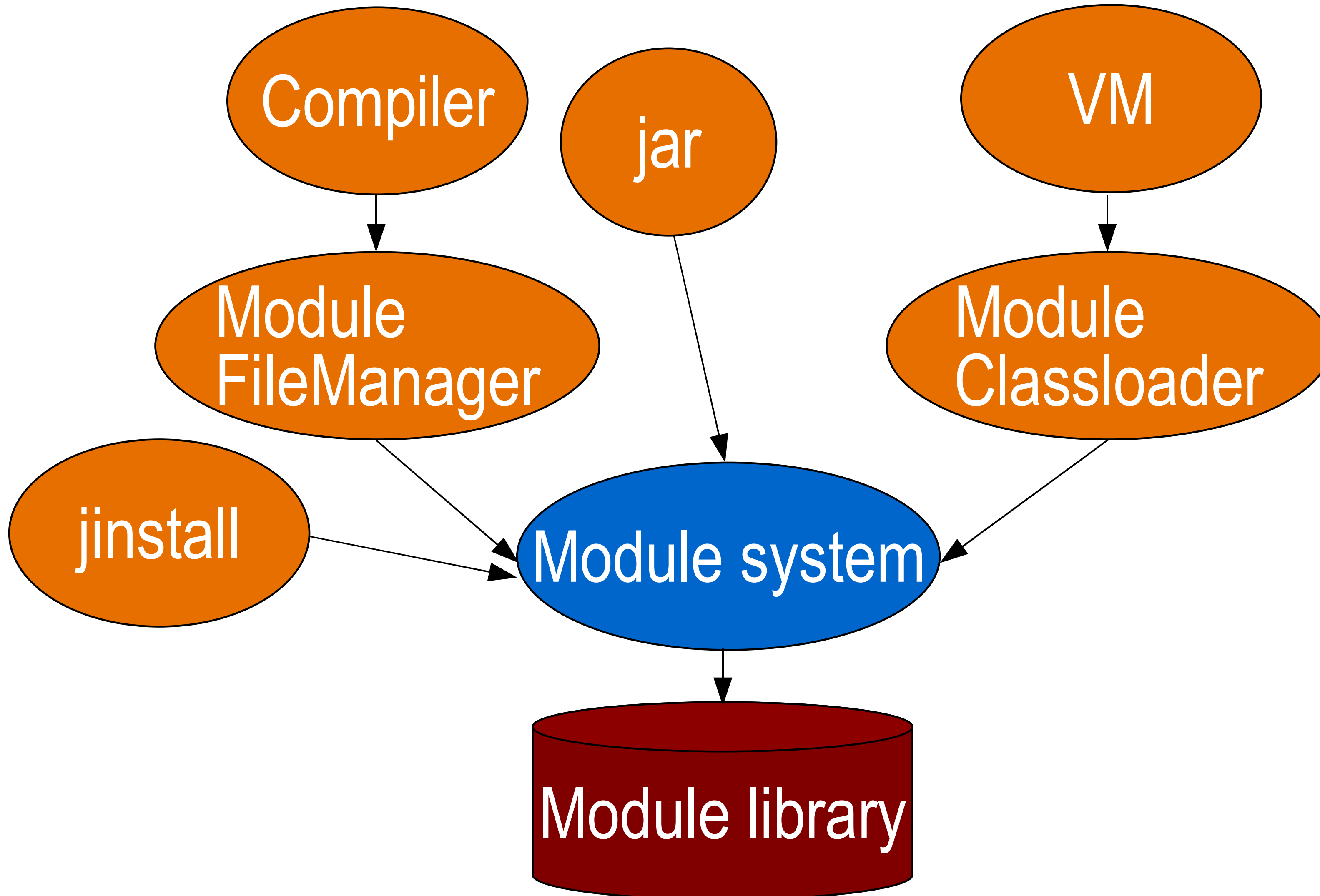
Compiler and VM today



Compiler and VM in SE 7



Compiler and VM in SE 7



Implications of dependencies

- Dependencies must be expressed in Java source
  - A Java compiler is not expected to read other kinds of file
- Annotations cannot and should not express them
  - "Should not": Dependencies govern compilation, and annotations should not change semantics of compilation
  - "Cannot": A Java compiler must parse and interpret module-info.java/class to set up a module's dependencies, before compiling source in that module.

    Parsing annotations requires their class definitions, which cannot be found in advance of dependency setup

Module friendship

- Anyone could require org.netbeans.core.utils module
- Arbitrary reuse is sometimes undesirable, so:

```
//  org/netbeans/core/utils/module-info.java
module org.netbeans.core.utils {
     require java.core;
     permit  org.netbeans.core;
}
```

- Permission granted to all if no explicit permit given
- Dependency is a two-way street; the requiring and required modules must work together

Module versioning

- Versioning is the best chance the Java community has for evolving code incompatibly
  - Change signatures and thrown exceptions
  - Change method behavior
  - Remove methods from classes
    - Binary incompatible for clients today
  - Remove methods from interfaces
    - Even if binary compatible for clients, this is source incompatible for implementers today
- Versioning should be in the programmer's hands
  - Not an artifact of CLASSPATH order set by deployer

Version support in the language

- Operands are module **identities**, not just names
- Module identity == Name @ Version

```
//  org/netbeans/core/module-info.java
@Foo
@Bar("Quux")
module org.netbeans.core@ 6.5 {
     require org.w3c.dom.parser@ 4.0;
     require java.core @ 1.7;
}
```

Version independence

- Compiler/VM knows that modules are versioned
- But version **structure**, **order**, and **ranges** are implementation details of a module system
- Structure examples
  - 1.2.3.4.5.6.7
  - A=1/B=7/C=100/D=19
  - R11V16.4M3.2T20081211090000
- Order and range examples
  - 1.2.3.4 > 1.2.3 1.6/6u10 < 1.7/b23 < 1.6/6u11
  - [1.0, 2.0) 1.0+ >=1.0 1.0

Version independence

- Compiler/VM can interpret module names
  - Trivial to get from a class's source or binary
  - Trivial to compare with String.equals()
- Compiler/VM cannot interpret module versions
  - Version comparison deferred to the module system
  - Suppose a module says: `require M @ 1.0+;`
  - Compiler/VM asks module system for M @ 1.0+
  - An answer of M @ 2.0 is OK
  - An answer of N @ 1.0 is not
- Implies a close relationship between compiler/VM and module system

Implications of versioning for the VM

- Module systems generally support multiple versions of a type at runtime
  - Classloaders are the natural mechanism for this
  - Can generally load modules with different names (M @ ... and N @ ...) into one classloader
- The VM must not allow a class in M @ X to access a module-private type in M @ Y
  - Both classes being in module M is insufficient
  - Semantics of M could have changed between X and Y
  - But the VM doesn't understand module versions!
  - VM must be able to infer module identity somehow

Diversion: What is a type's identity?

- Today:
  - > Compile-time: Type name
  - > Run-time: (Type name, Defining classloader)
- JSR 294:
  - > In theory, classloaders are an implementation detail
  - > Compile-time: (Type name, Module identity)
  - > Run-time: (Type name, Module identity)
- But:
  - > Not realistic to change run-time type identity now
  - > All stages of linking know about it

```
instructionIsTypeSafe(invokevirtual(CP), Environment, _Offset,
                      StackFrame, NextStackFrame,
                      ExceptionStackFrame) :-

    CP = method(MethodClassName, MethodName, Descriptor),

    MethodName \= '<init>',

    MethodName \= '<clinit>',

    parseMethodDescriptor(Descriptor, OperandArgList, ReturnType),

    reverse(OperandArgList, ArgList),

    currentClassLoader(Environment, L),

    reverse([class(MethodClassName, L) | OperandArgList],
            StackArgList),

    validTypeTransition(Environment, StackArgList, ReturnType,
                        StackFrame, NextStackFrame),

    canPop(StackFrame, ArgList, PoppedFrame),

    passesProtectedCheck(Environment, MethodClassName,
                         MethodName, Descriptor, PoppedFrame),

    exceptionStackFrame(StackFrame, ExceptionStackFrame).
```

Fundamental rule of Java modularity

- Modules whose identities share a name but not a version **must** go in different classloaders
  - > A module system is generally free to assign modules to classloaders as it sees fit
  - > But a module system is not free to load multiple modules of the same name and different versions into a single classloader
- VM can treat a class's defining classloader as a proxy for the class's module's version
  - > VM doesn't need to know a module's version per se
  - > Module name is trivially available from a class def

Access control of module-private types

- Runtime module = (Module name, Defining CL)
  - Analogous to runtime package:
    (Package name, Defining CL)
- Accessibility for module-private members exploits concept of "same runtime module"
  - If accessed type has same defining CL as accessing type, VM only has to compare module names to ensure types' modules have same identity
    - Access succeeds if names equal
  - If accessed type has different defining CL from accessing type, then types' modules are either different versions of same name, or different names entirely
    - Access fails always

Further features for modularity

Virtual modules

- Multiple vendors might wish to have competing implementations of the "same" module

```
module org.netbeans.core.utils {
    require java.core @ 7.0;
}
```

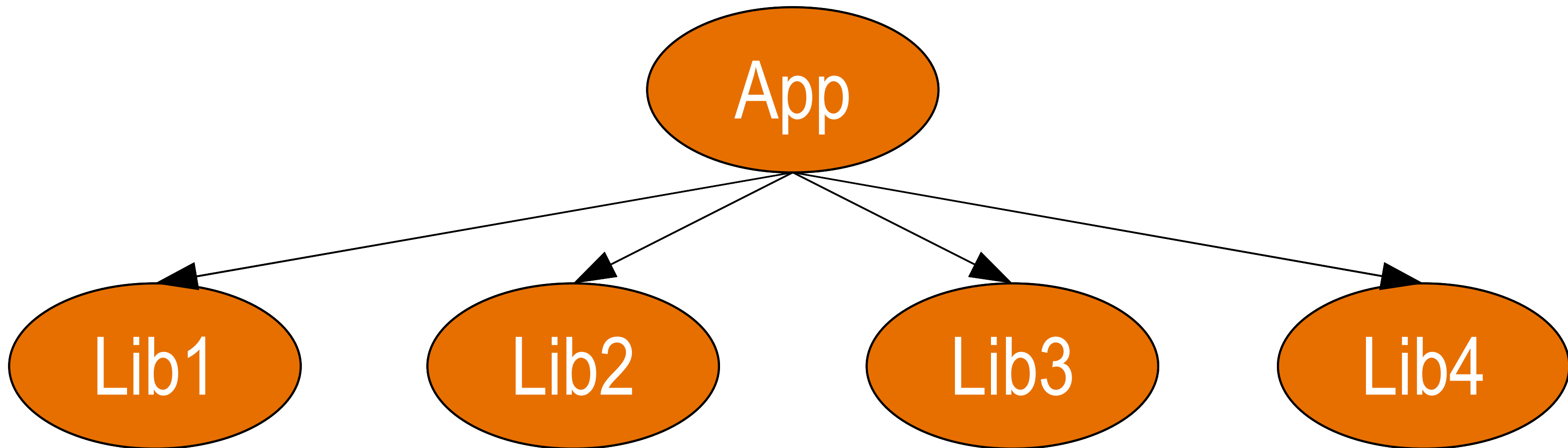
```
//  jdk/module-info.java
module jdk @ 1.7 { provide java.core @ 7.0; }
```

```
//  jrockit/module-info.java
module jrockit @ 9.5 {provide java.core @ 7.0; }
```

```
//  ibm-jre/module-info.java
module ibm-jre @ 3.6 {provide java.core @ 7.0; }
```

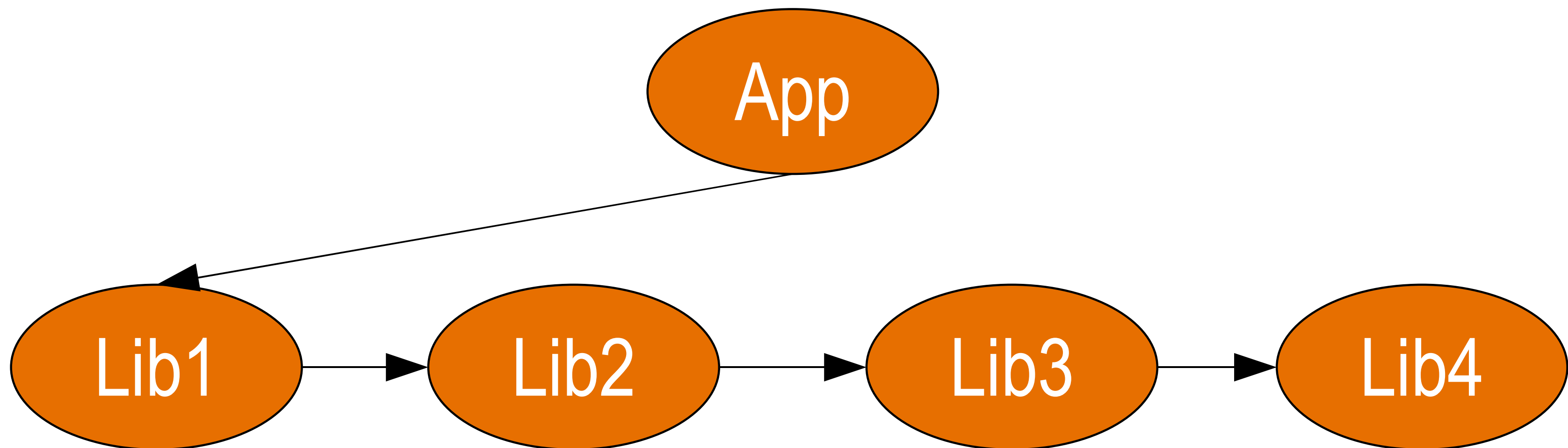
Type observability

- Consider an app using the CLASSPATH today:



Type observability

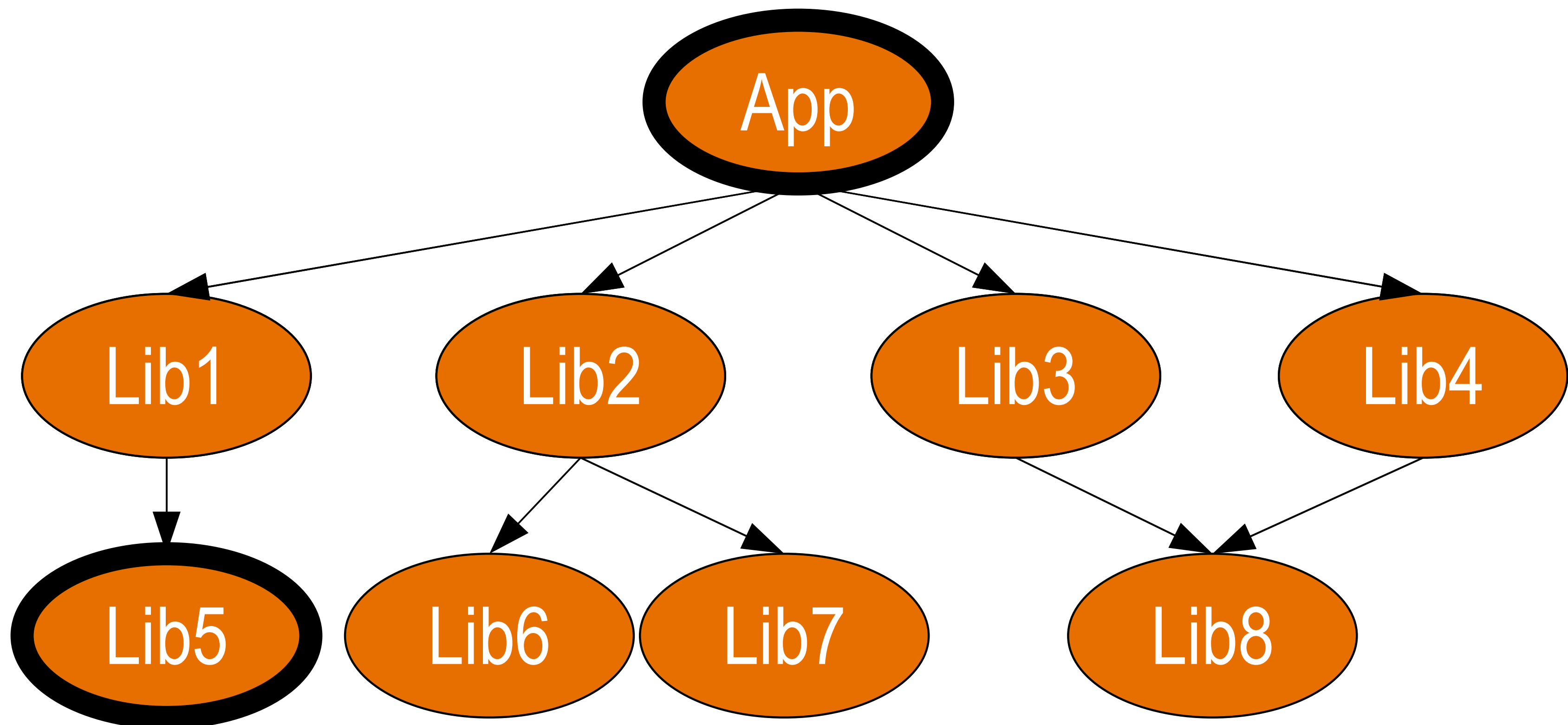
- App can see all public types from all Libs



- (Though the first lib to have type T dominates later libs with the same T)

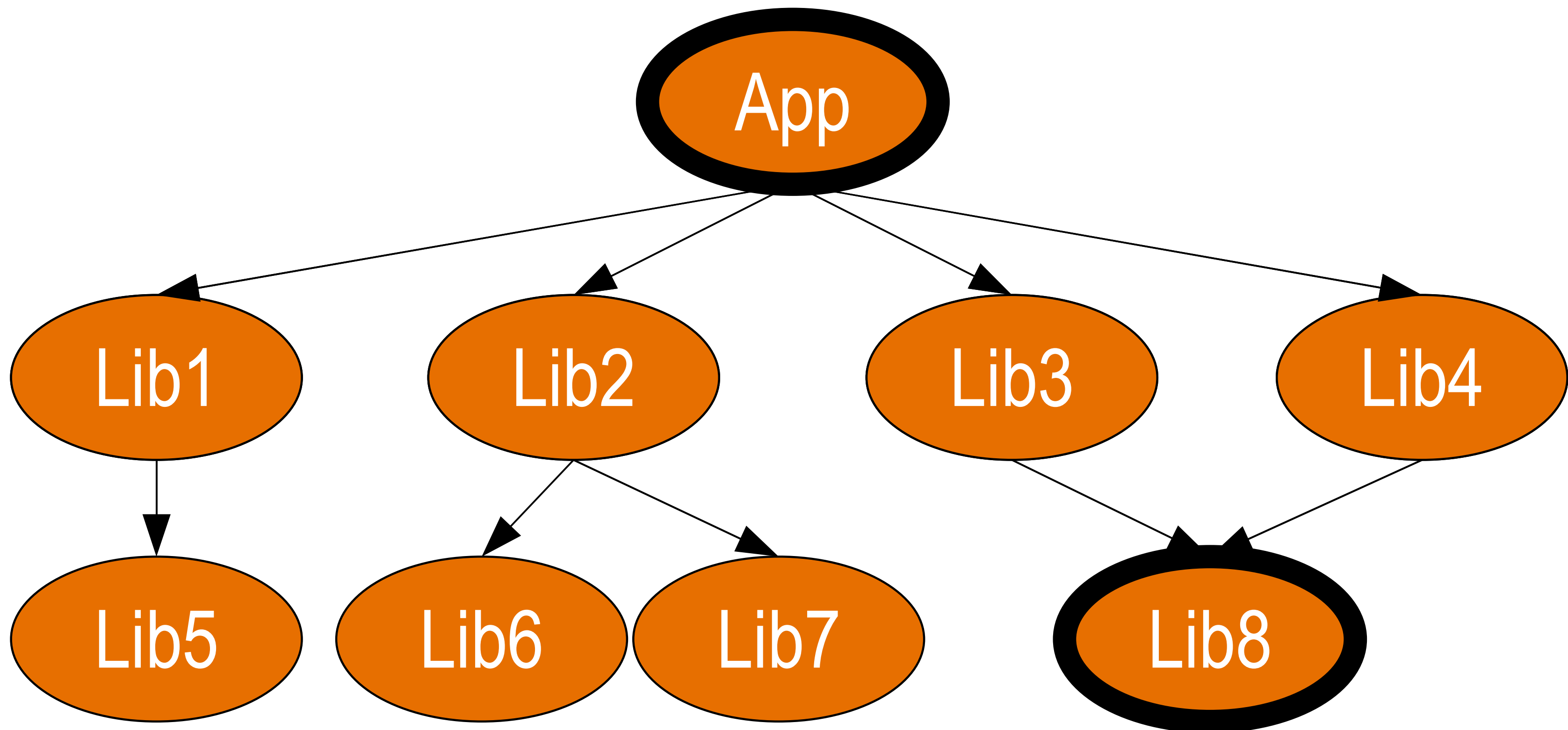
When dependencies are a graph...

- Should App see Lib5's public types? In general, no



Observability is sometimes transitive

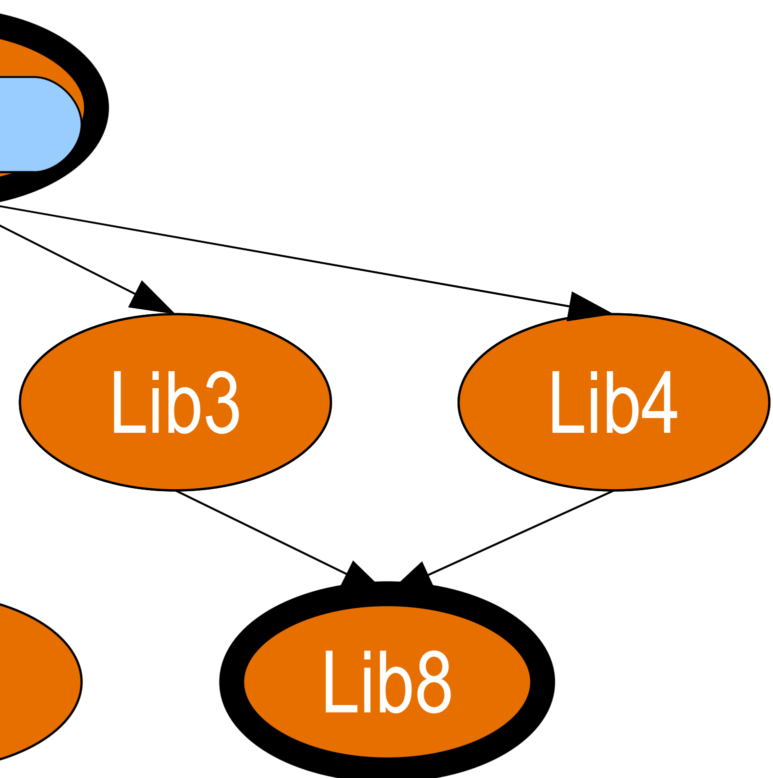
- It's likely that App will need to see Lib8's types, in order to exchange objects between Lib3 and Lib4



Observability is sometimes transitive

- It's likely that App will need to see Lib8's types, in order to exchange objects between Lib3 and Lib4

This is not the same problem as ensuring Lib3 and Lib4 use the same Lib8, desirable though that is



Transitive observability: Re-export

- Could default to "re-exporting" public types
  - Closest to current CLASSPATH behavior
  - Requiring module can disable re-export
  - Lib3 and Lib4 would re-export Lib8's types up to App

```
//  org/netbeans/core/module-info.java
@Foo
@Bar("Quux")
module org.netbeans.core @ 6.5 {
    private require org.w3c.dom.parser @ 4.0;
    require java.core @ 1.7;
}
```

Scalability

- To modularize the JDK, existing packages must be split at module boundaries
  - ¼ of java.lang in module A
  - ½ in module B
  - ¼ in module C
- Supports faster download and startup
- Generally applicable for incremental delivery of large apps, and apps with optional classes

Multi-module packages

- If module M depends on package P directly, it's not possible to split P over multiple modules
  - A module system generally could not know how many modules (e.g. JAR files) to load to complete the package
- If module M depends on module N, N could define all of package P itself, or just a part and get the rest from other modules
  - P is a **multi-module package**
  - Module dependency promotes abstraction: N hides the fact that P is multi-modular from N's clients

Danger Will Robinson!

- A module system will generally give each module its own classloader
- Defining types from the same package in different classloaders is **bad** if the classloaders are visible to each other
  - > Loader constraints
  - > Package-private access
- **Dynamic** split packages are **unsafe**
- **Static** split packages are **safe**
  - > Just ensure they don't turn into dynamic split packages!
  - > Offer a language feature to ensure that

Multi-module packages

```
module M @ 1.0 {
    require ... @ ...;
    local require N @ 3.0;
}
```

- Think of M as physically including the types in N
- At runtime (including Java compiler execution!), N's types **must** be loaded by the same classloader as M's types
- Any `require` dependency **may** share classloaders
  - Only `local` dependencies must share

Status

- JSR 294 already has a world-readable mailing list
- JSR 294 will have a world-writable mailing list too
- Implementation in the OpenJDK Jigsaw project

- jsr-294-comments@jcp.org
- http://blogs.sun.com/abuckley/
- http://blogs.sun.com/mr/
- Modularity Q&A tonight

End

Diversion: Modules as types

- OO type systems focus on encapsulation because it supports information hiding, the foundation of reuse
  - > Private state, public operations
- Modules are really a type system extension
- The usual tradeoffs in type system design apply
  - > Safety v. Performance
  - > Expressiveness v. Decidability
  - > Parameterization (Versions)
  - > Reification (Classloaders)
- Java uses strong static typing, so the compiler must know about module membership+dependence