

# Agenda

- Is everything OK with project Jigsaw?
  - Critical Deficiencies in Jigsaw (JSR-376)
    - Posted by Scott Stark on Apr 14, 2017
  - EC reject Jigsaw
  - Concerns regarding Jigsaw

# Java 9 Features

Java 9 is Feature Complete!

## \* Modularity

- \* 200: The Modular JDK (Jigsaw/JSR 376 and JEP 261)
- \* 201: Modular Source Code
- \* 220: Modular Run-Time Images
- \* 238: Multi-Release JAR Files
- \* 261: Module System
- \* 275: Modular Java Application Packaging
- \* 282: jlink: The Java Linker

# Modularisation & Modular Architecture

- \* *Modularization* is the act of decomposing a system into self-contained modules.
- \* *Modules* are identifiable artifacts containing code, with metadata describing the module and its relation to other modules.
- \* A *modular application*, in contrast to a monolithic one of tightly coupled code in which every unit may interface directly with any other, is composed of smaller, separated chunks of code that are well isolated.
- \* *Versioning*: depend on a specific or a minimum version of a module

# Modularisation & Modular Architecture (cont.)

- \* *Characteristics of modular systems:*

- \* **Strong encapsulation:** A module must be able to conceal part of its code from other modules. Consequently, encapsulated code may change freely without affecting users of the module.
- \* **Well-defined interfaces:** modules should expose well-defined and stable interfaces to other modules.
- \* **Explicit dependencies:** dependencies must be part of the module definition, in order for modules to be self-contained.  
*A module graph: nodes represent modules, and edges represent dependencies between modules*

# Java 9 Modules Goals

- \* Java Platform Module System ([JSR 376](#))
  - \* Reference implementation: [OpenJDK Project Jigsaw](#)
- \* Module System ([JEP 261](#))
- \* Modular JDK ([JEP 200](#))
- \* Modularize the layout of the source code in the JDK ([JEP 201](#)).
- \* Modularize the structure of the binary runtime images ([JEP 220](#)).
- \* Disentangle the complex implementation dependencies between JDK packages.
- \* Internal APIs encapsulation ([JEP 260](#))
- \* Make Java SE more flexible, scalable, maintainable and secure
- \* Make it easier to construct, maintain, deploy and upgrade applications
- \* Enable improved performance

# Java 9 Module System

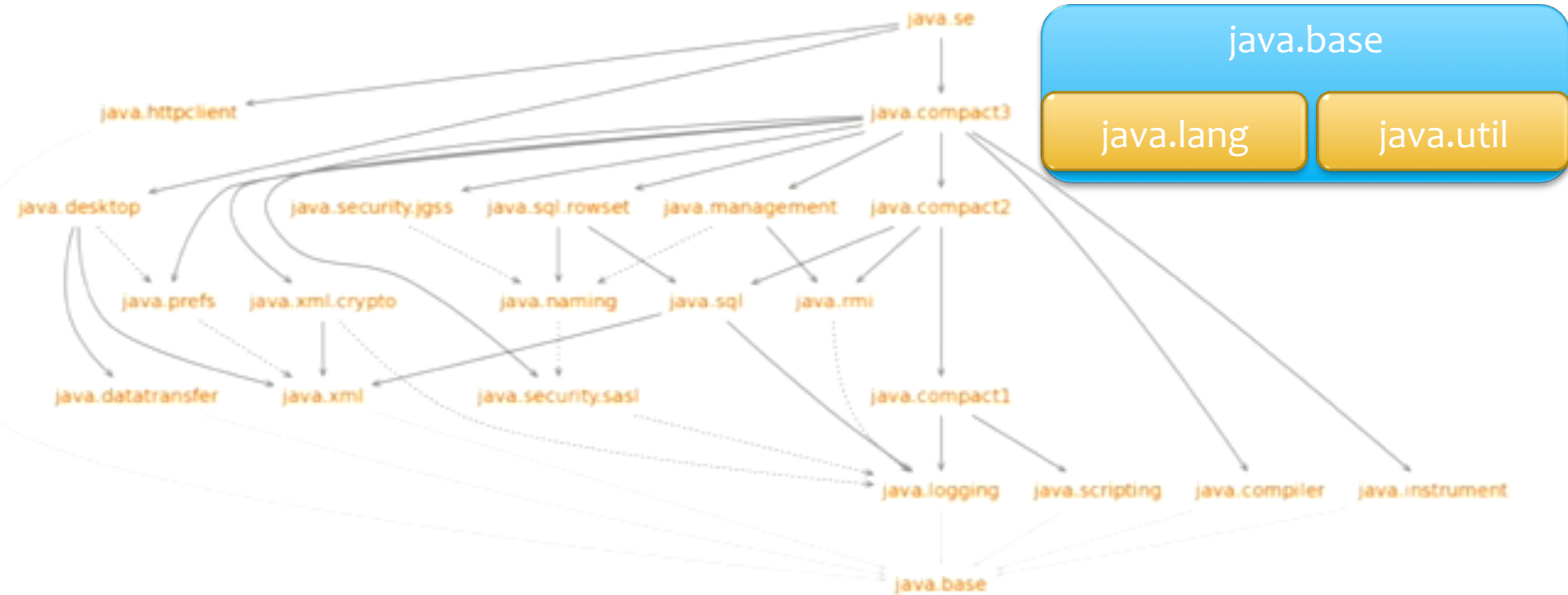
- \* Modules can either export or strongly encapsulate packages
- \* Modules express dependencies on other modules explicitly.
- \* Each JAR becomes a module, containing explicit references to other modules.
- \* A module has a publicly accessible part and an encapsulated part.
- \* All this information is available at compile-time and run-time
- \* Accidental dependencies on code from other non-referenced modules can be prevented.
- \* optimizations can be applied by inspecting (transitive) dependencies

# Benefits of Java 9 Module System

- \* **Reliable configuration:** The module system checks whether a given combination of modules satisfies all dependencies before compiling or running code
- \* **Strong encapsulation:** Modules express dependencies on other modules explicitly.
- \* **Scalable development:** Teams can work in parallel by creating explicit boundaries that are enforced by the module system.
- \* **Security:** No access to internal classes of the JVM (like `Unsafe`).
- \* **Optimisation:** optimizations can be applied by inspecting (transitive) dependencies. It also opens up the possibility to create a minimal configuration of modules for distribution.

# JDK 9 Platform Modules

- \* Module `java.base` exposes packages `java.lang`, `java.util` etc. It is the core Java module which is imported by default
- \* JDK now consists of about 90 platform modules





# Modules in Java 9

- \* A module has a *name* (e.g. `java.base`), it groups related code and possibly other resources, and is described by a *module descriptor*.
- \* Like packages are defined in `package-info.java`, modules are defined in `module-info.java` (in root package)
- \* A *modular jar* is a jar with a `module-info.class` inside it

```
1  module com.toy.anagrams {  
2      requires java.logging;  
3      requires java.desktop;  
4      exports com.toy.anagrams.ui;  
5  }
```

dependency

encapsulation

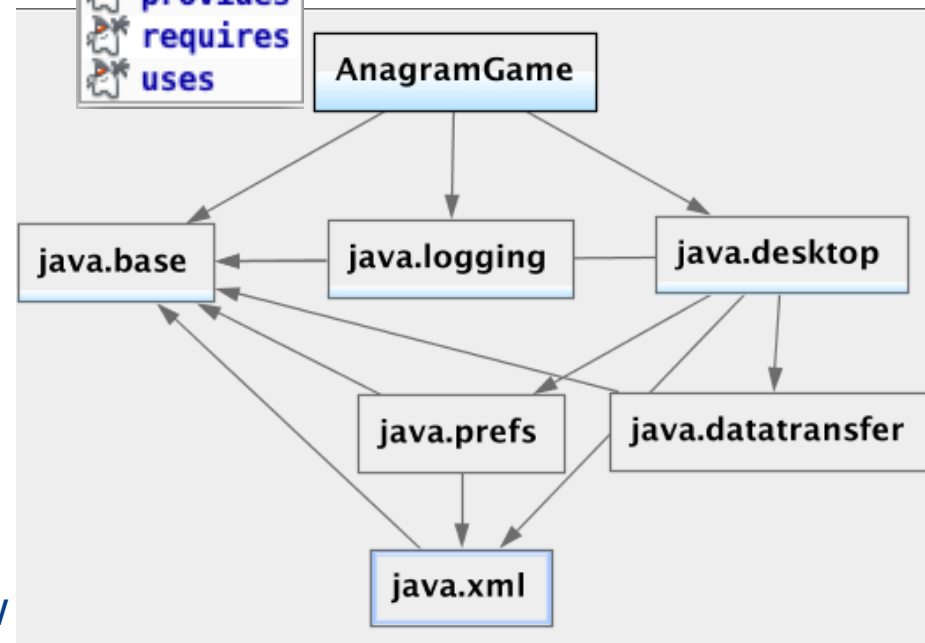
Only public classes of an exported module can be accessed by other modules

# module-info.java

The screenshot shows an IDE with a project named 'AnagramGame'. The 'Projects - Todo' pane on the left shows the project structure: 'Source Packages' containing '<default package>' (with 'module-info.java'), 'com.toy.anagrams.lib' (with 'StaticWordLibrary.java' and 'WordLibrary.java'), and 'com.toy.anagrams.ui' (with 'About.java' and 'Anagrams.java'). Below are 'Test Packages', 'Libraries', and 'Test Libraries'. The 'Start Page' pane on the right shows the 'module-info.java' file with the following code:

```
1 module AnagramGame {  
2     requires java.logging;  
3     requires java.desktop;  
4 }  
5
```

A tooltip is visible over the 'requires' keyword, listing module relationships: exports, opens, provides, requires, and uses.



# module-info.java


























- \* Exports a package
- \* Allows to use reflection on types in the package
- \* Provides a service provider
- \* Requires another module
- \* Uses a service

# Public Review Ballot for JSR 376


Source: [DZone](#)


EC

Azul Systems, Inc. 	Credit Suisse 	Eclipse Foundation, Inc. 	Fujitsu Limited 
Gemalto M2M GmbH 	Goldman Sachs & Co. 	Grimstad, Ivar 	Hazelcast 
Hewlett Packard Enterprise 	IBM 	Intel Corp. 	Keil, Werner 
London Java Community 	MicroDoc 	NXP Semiconductors 	Oracle 
Red Hat 	SAP SE 	Software AG 	SouJava 
Tomitribe 	Twitter, Inc. 	V2COM 	

## Icon Legend

Yes 

No 

Abstain 

Not voted 

# RedHat

Source: [DZone](#)

	Jigsaw	Class-Loader	OSGi	Java EE (Spec)	ext/lib
Allows cycles between packages in different modules	✗	✓	✓	✓	✓
Isolated package namespaces	✗	✓	✓	✓	✗
Allows lazy loading	✗	✓	✓	✓	✓
Allows dynamic package addition	✗	✓	✓	✓	✗
Unrestricted naming	✗	✓	✗	✓	✓
Allows multiple versions of an artifact	✗	✓	✓	✓	✓
Allows split packages	✗	✓	✓	✓	✓
Module redefinition	✗	✓	✓	✓	✓
Allows textual descriptor	✗	✓	✓	✓	✓
Theoretically Possible to AOT-compile	✓	✓	✓	✓	✓

# Main arguments

- \* JSR-376 is **not** a standard
  - \* Jigsaw is good for modularising JDK itself but not for being used to real applications on top of the JVM
- \* JSR-376 is based on the idea of subtracting capabilities and adding restrictions
  - \* this reduces the ability for application developers to easily adapt applications to this modular world.
- \* Libraries that deal with services, class loading, or reflection in any way will break and require a lot of refactoring to abide with the Jigsaw's best practices

# Reinvention, Not Standardization

- \* The Jigsaw implementation has worked successfully for modularising Java itself, but is largely untried in wider production deployments of any real applications on top of the JVM.
- \* Many application deployment use cases which are widely implemented today are not possible under Jigsaw, or would require a significant re-architecture.

# Reductive Design Principles

- \* Jigsaw's key design points are predicated on a reductive approach to forward compatibility, which works well for modularising Java itself, but becomes restrictive for the broader use cases that application deployments have.
- \* JSR-376 is based on the idea of subtracting capabilities and adding restrictions
  - \* this reduces the ability for application developers to easily adapt applications to this modular world.
- \* Libraries that deal with services, class loading, or reflection in any way will break and require a lot of refactoring to abide with the Jigsaw's best practices
- \* The patterns introduced within Jigsaw will create backwards- and forwards-compatibility problems that will be very difficult to fix later.



# Compatibility with JavaEE

- \* For Java EE 9 to be based on Jigsaw, vendors need to completely throw out compatibility, interoperability, and feature parity with past versions of the Java EE specification.

# Issues

- \* *concealed package conflicts*
- \* *split packages*
- \* *duplicate packages*
- \* *multiple module versions*
- \* *module naming restrictions*
- \* *cyclic dependencies*
- \* *JSR-250's awkward place*
- \* *service loader changes*
- \* *reflection behavior changes*

# Cyclic Dependencies

- \* “dependency cycles among modules are forbidden during compilation, link, and run time ... The proposed JVM specification does not specify that module cycles are forbidden during class resolution or initialization”.
- \* This is not true during run time.
- \* “An automatic module reading all other modules may create cyclic dependency, which is allowed after the module graph has been resolved. Recall that cyclic dependency between modules is not allowed during the module graph resolution. That is, you cannot have cyclic dependency in your module declarations.” [Sharan K. (2017)]
- \* `opens`: Allows to use reflection on types in the package

# Automatic Modules

- \* “automatic modules bring more harm than good”
  - \* Tools like [moditect](#) can modularize an archive based on dependency metadata
  - \* `Module-Name` metadata, which was removed by the proposal, would allow someone to choose a more descriptive name than the default that is created by the automodule name algorithm which only uses elements of the filename as the module name. This might cause name conflicts with other namespaced modules

# Inadequate Isolation

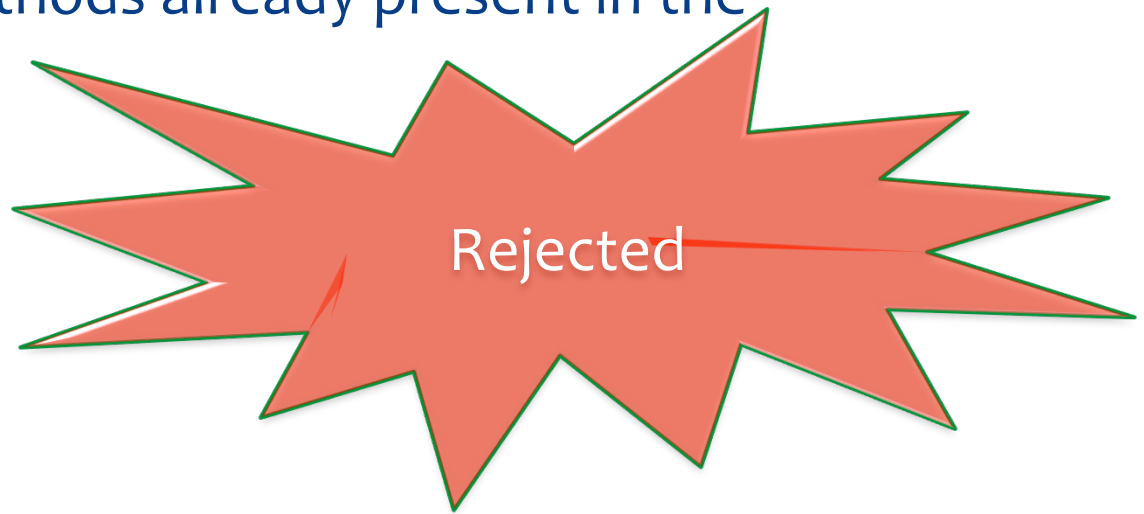
- \* Jigsaw doesn't resolve multiple versions; it doesn't support fully isolated package namespaces and separated module classloaders
- \* It attempts to use a single class loader for all JDK modules, and then reuse that approach for application modules on the module path.  
Solution: use multiple classloaders for platform and application modules
- \* “Two modules with the same package names in them are considered to be different versions of the same module”.
- \* “Two modules with the same name are considered to be two versions of the same module”.
- \* Concealed package conflicts
  - \* When two modules have the same package name in them, but the package is private in both modules, the module system cannot load both modules into the same layer.

# No “current module”

- \* Applications that rely on Thread Context Class Loader (TCCL) will exhibit incorrect behaviour with Jigsaw
- \* To dynamically install and redefine modules, Jigsaw uses a hierarchical grouping called Layers that support multiple parents.
  - \* Layers do not scale, introduce locking problems, visibility issues, complex resolution for parent / child-first dilemmas etc.
  - \* graphs should be used instead
- \* Modules are always loaded eagerly within a layer, instead of lazily
  - \* platform modules must be divided into two groups: the eagerly resolved platform modules, and a set of optional modules that are only loaded if explicitly specified on the command line.
- \* In Jigsaw the JVM module path cannot have modules added to it at runtime.

# Layer primitives

- \* There are proposed primitives for Jigsaw that add the ability to dynamically modify a module in a few specific ways.
- \* Ability to dynamically add packages and other module characteristics as they are discovered.
- \* The code to make this change is a very small patch that exposes a small number of methods already present in the implementation.



# Module Naming Conventions

- \* module names must be valid Java language names (Java-qualified identifier)
  - \* -, : (used in Maven) are not allowed
  - \* Recommended to use the reverse-domain name pattern to give modules unique names.
- \* Containers have the ability to bypass these naming restrictions, but in order to do so, they must generate bytecode for their module descriptors (as the descriptor building API enforces the javac naming rules).
- \* Module version strings in Jigsaw are constrained by a format which does not reflect any current versioning practices.
  - \* a module system should support versioning schemes that reflect a users' or containers' best practices in common use



# Module Descriptors

- \* Module descriptors are in bytecode format (.class)
  - \* An uncommon implementation choice
- \* There is no need for module descriptors to be a part of the Java language specification

# Java 9 Modules & Services

- \* Java 6 uses a Query-based approach, the `ServiceLoader`:

```
ServiceLoader<Provider> serviceLoader =  
    ServiceLoader.load(Provider.class);  
for (Provider provider : serviceLoader) { return provider; }
```

```
ServiceLoader<Provider> serviceLoader =  
    ServiceLoader.load(Provider.class).stream().filter(...);
```

- \* However, the `ServiceLoader` has a number of problems:
  - \* it isn't dynamic (you cannot install/uninstall a plugin/service at runtime)
  - \* it does all service loading at startup (as a result it requires longer startup time and more memory usage)
  - \* it cannot be configured; there is a standard constructor and it doesn't support factory methods
  - \* it doesn't allow for ranking/ordering, i.e. we cannot choose which service to load first

# Java 9 Service Loader

- \* Java 9 modifications to Java 6 `ServiceLoader`:
  - \* No relative services; the new module-based service locator does not have relative behaviour
  - \* Ordering of services (as they were discovered) is lost
  - \* all service interfaces and implementations on the module path are flattened into a single, global namespace
  - \* No extensibility / customizability of service loading; the service layer provider must provide a fixed mapping of available services up front
  - \* multiple-site declarations; every module that uses a service must also declare that the service is being used in the module descriptor; no global layer-wide service registry

# Reflection

- \* `setAccessible()` method is disallowed from being invoked from modules which are not specifically granted access to the module in which the corresponding member exists.
  - \* this restriction is not consistently applied
    - \* legacy classpath-based code
    - \* the *unnamed* module
- \* Security justification: less reflective access means fewer CVEs.
- \* Compatibility: `opens` keyword allows modules to opt in to allowing inter-module reflection. Reflection access problems will be found only at runtime
- \* The consuming module must somehow grant open access to the specification implementation(s)

# Distribution Model

- \* how a module system manages independently developed, versioned, and packaged units of software?
- \* Overridable Descriptor Approach
  - \* a module can redefine the module specification of its full tree of dependencies (Maven, JBoss)
- \* Flexible Resolution System Approach
  - \* a module analyzes detailed data about the modules in the system and produces a solution (OSGi)
- \* Jigsaw
  - \* no flexible resolution system, no adequate metadata to generate solutions for independent software composition, no override

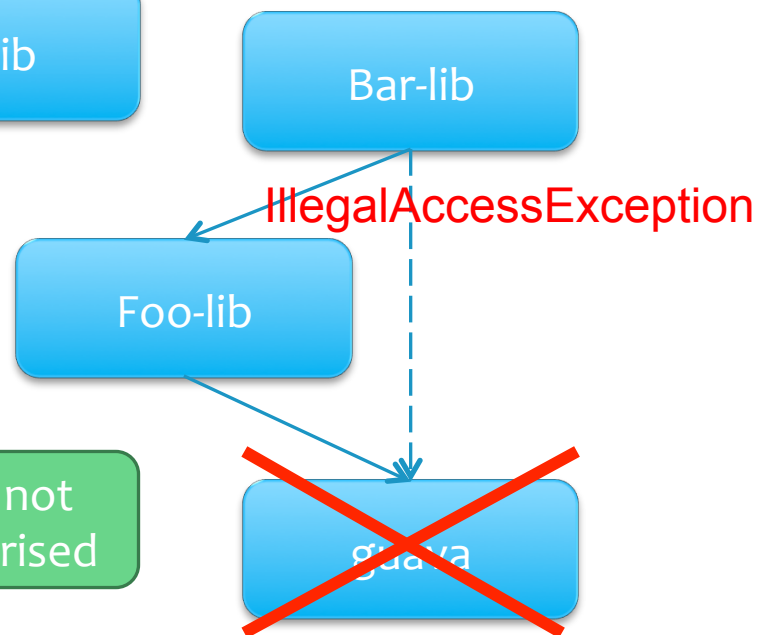
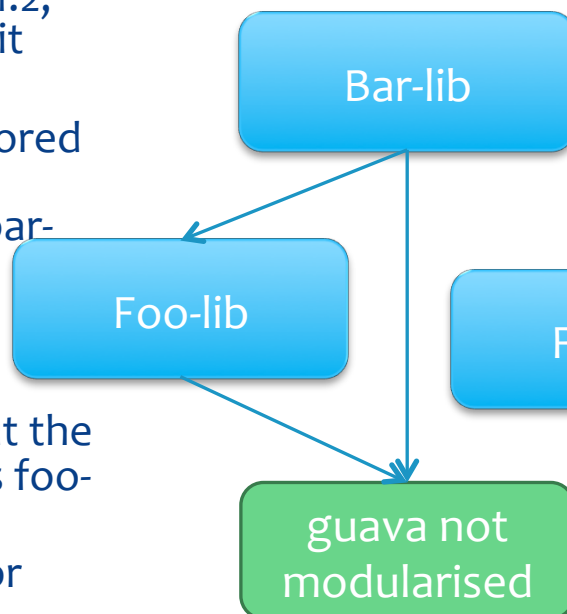
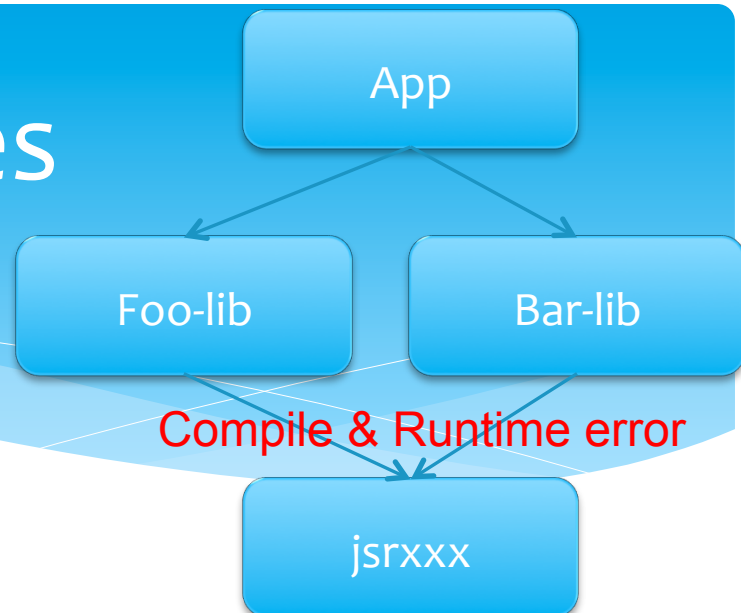
# Decentralized artifact repositories vs. centralized module registry

- \* In a centralized module registry, a set of artifacts must resolve in a 100% mutually consistent way and consistency must be guaranteed for **all** artifacts.
- \* Maven Central repository cannot meet this need

# Examples

- \* Duplicate spec dependencies
- \* Dropped exported transitive
- \* Over-eager shading
- \* Qualified Open

1. foo-lib opens foo.beans to bar-lib (only bar-lib has access, works with 1.1)
2. myapp uses foo-lib from maven
3. zeta-lib uses new bar-lib 1.2, which has new methods it needs
4. bar-lib 1.2 recently refactored and has moved its bean introspection code to a bar-lib-impl module
5. myapp wants zeta-lib, but the upgrade of bar-lib breaks foo-lib
6. myapp must now refactor foo-lib



# Other issues

- \* What about independent implementations?



# References

- \* [Critical Deficiencies in Jigsaw](#)
- \* [EC Rejects Jigsaw](#)
- \* Bateman A. (2016), “Prepare for JDK 9”, [JavaOne](#).
- \* Bateman A. (2016), “Introduction to Modular Development”, [JavaOne](#).
- \* Bateman A. & Buckley A. (2016), “Advanced Modular Development”, [JavaOne](#).
- \* Buckley A. (2016), “Modules and Services”, [JavaOne](#).
- \* Buckley A. (2016), “Project Jigsaw: Under The Hood”, [JavaOne](#).
- \* Bateman A., Chung M., Reinhold M. (2016), “Project Jigsaw Hack Session”, [JavaOne](#).
- \* Evans, B. (2016), “An Early Look at Java 9 Modules”, *Java Magazine*, Issue 26, January-February, pp.59-64.
- \* Mak S. & Bakker P. (2016), *Java 9 Modularity*, O’Reilly (Early Release)
- \* Reinhold M. (2016), *Problem modules reflective access*, [Voxxed](#)
- \* Sharan K. (2017), *Java 9 Revealed*, Apress.

# Questions

