Java SE 9 modules

An introduction

Stephen Colebourne - @jodastephen
Engineering Lead, OpenGamma
April 2017

Topics

Modules - Project Jigsaw

https://www.flickr.com/photos/distillated/4019168958/

Modules

- JPMS - Java Platform Module System
- Main feature of Java SE 9
- Developed as Project Jigsaw
- Originally targetted at Java SE 7
    - first JSR was in 2005, 12 years ago
- Still incomplete even though very late in 9 development
- Planned release date of July 27th 2017
    - just over 100 days!

Goals

- Scale Java SE to small devices
- Improve security and maintainability
- Better application performance
- Easier to construct & maintain libraries/applications

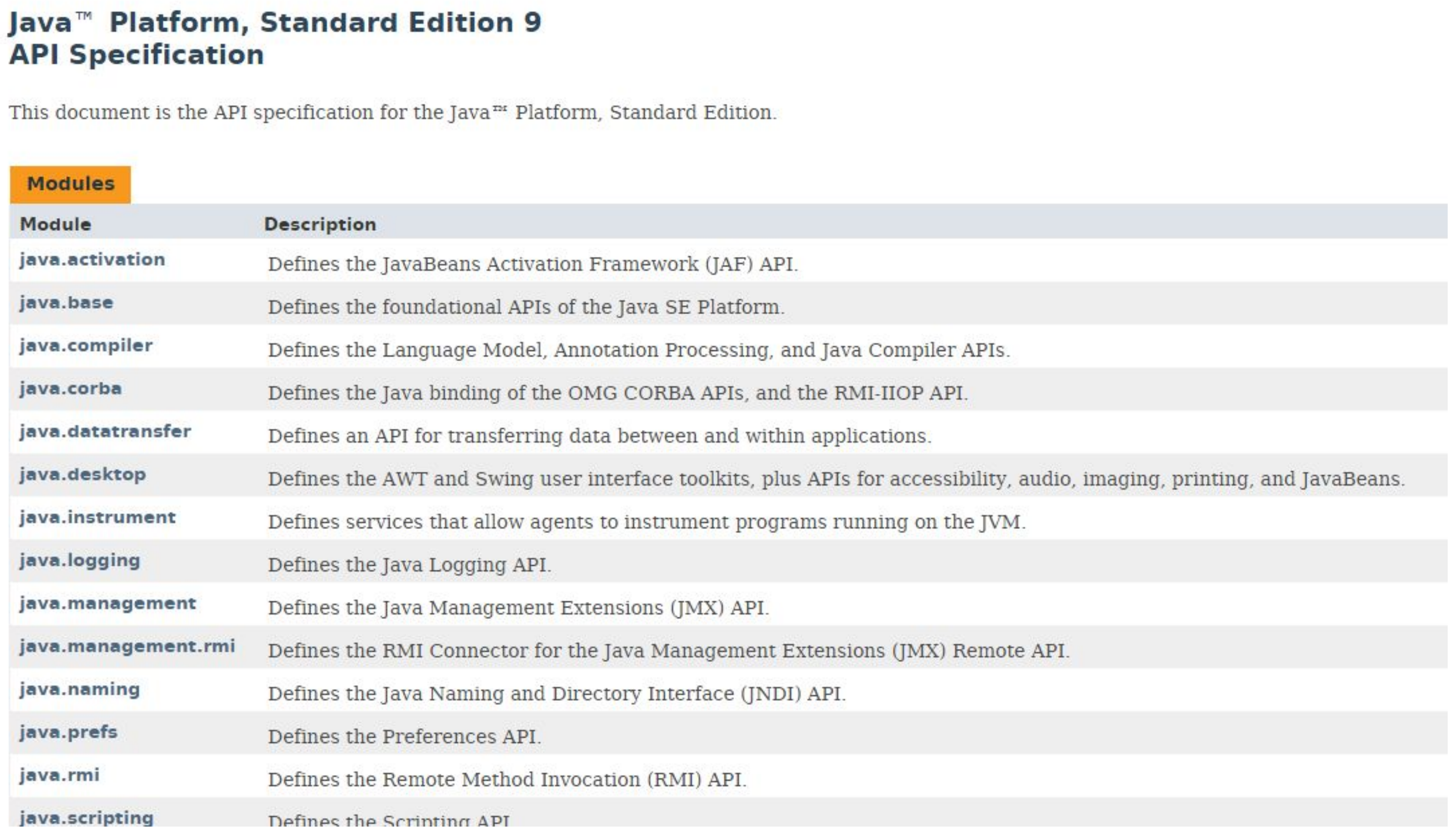
JDK too big

- Java SE 8 has 210 packages
- Many are not needed by all applications
  - CORBA, Swing, AWT, XML
- Need to split JDK into smaller units, aka modules

Modularised JDK

- 24 modules in Java SE 9
- Every application gets java.base
    - packages java.lang, io, math, net, nio, util, security, text, time
- Separate modules for
    - logging
    - sql
    - xml
    - desktop (AWT/Swing)
    - prefs
    - rmi

Multi-module Javadoc



Java™ Platform, Standard Edition 9
API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

Modules

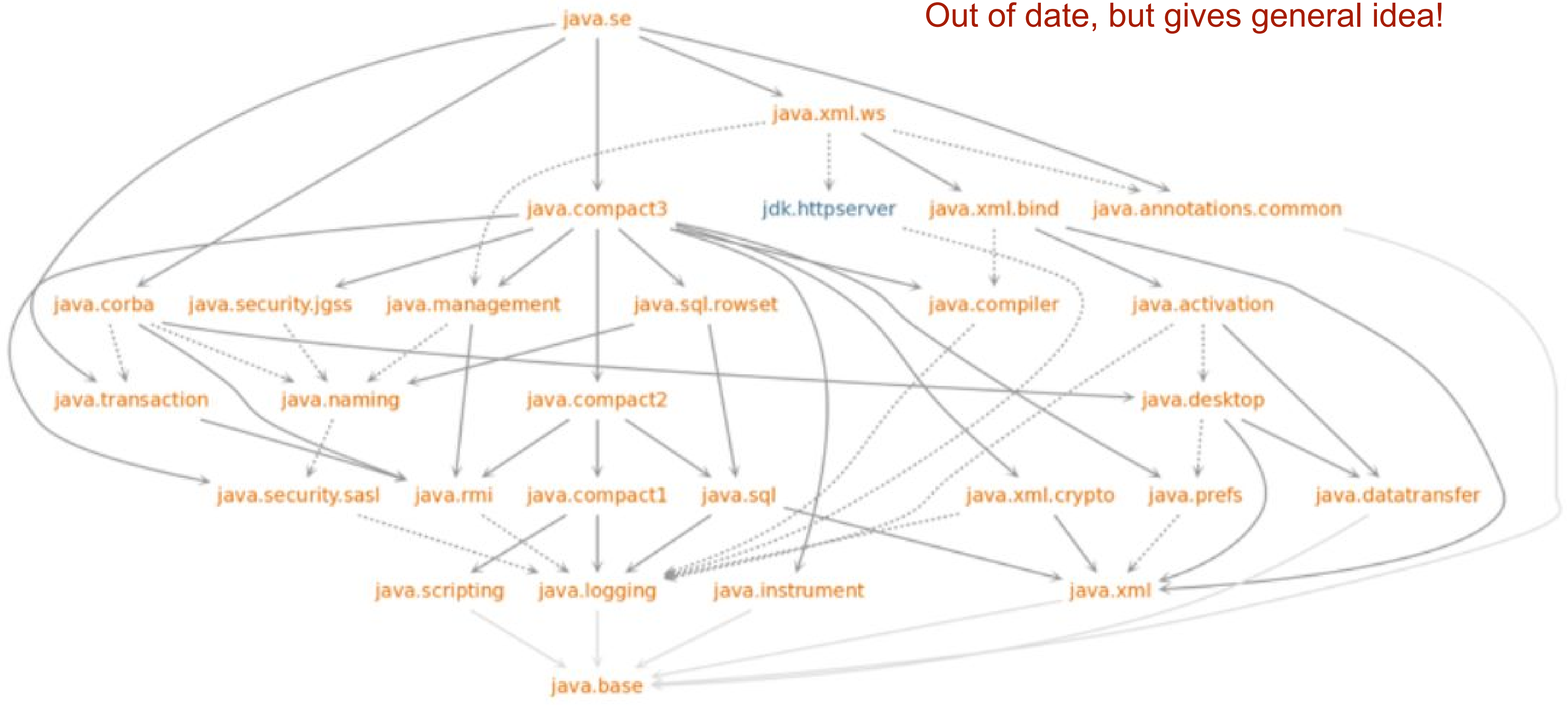
Module	Description
java.activation	Defines the JavaBeans Activation Framework (JAF) API.
java.base	Defines the foundational APIs of the Java SE Platform.
java.compiler	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
java.corba	Defines the Java binding of the OMG CORBA APIs, and the RMI-IIOP API.
java.datatransfer	Defines an API for transferring data between and within applications.
java.desktop	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
java.instrument	Defines services that allow agents to instrument programs running on the JVM.
java.logging	Defines the Java Logging API.
java.management	Defines the Java Management Extensions (JMX) API.
java.management.rmi	Defines the RMI Connector for the Java Management Extensions (JMX) Remote API.
java.naming	Defines the Java Naming and Directory Interface (JNDI) API.
java.prefs	Defines the Preferences API.
java.rmi	Defines the Remote Method Invocation (RMI) API.
java.scripting	Defines the Scripting API

Deprecated modules

- 6 modules are deprecated for removal
  - java.activation
  - java.corba
  - java.transaction
  - java.xml.bind
  - java.xml.ws
  - java.xml.ws.annotation
- Mass deletion from the JDK!
- Not in Java 9 by default!

Module graph

Out of date, but gives general idea!



Official process

- JEP
  - 200: The Modular JDK
  - 201: Modular Source Code
  - 220: Modular Run-Time Images
  - 260: Encapsulate Most Internal APIs
  - 261: Module System
  - 282: jlink: The Java Linker
- JSR
  - JSR 376: Java Platform Module System

Issues to Requirements

https://www.flickr.com/photos/distillated/4019168958/

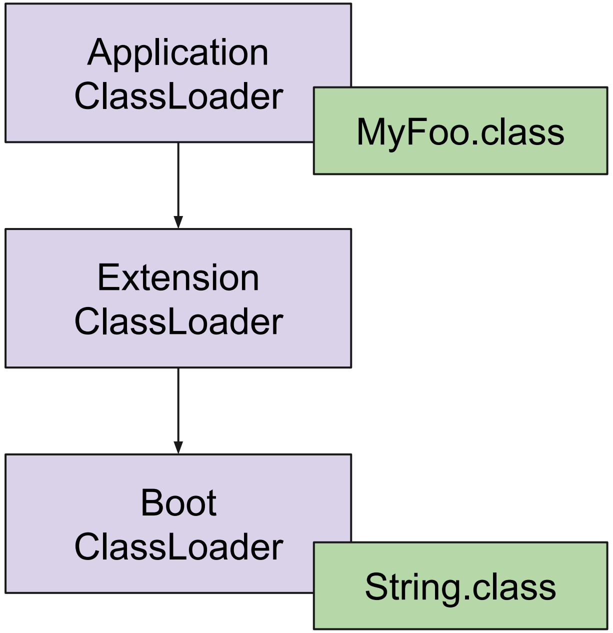
Classpath

- Classes are loaded on demand
- Classpath specifies where to find bytecode
  - `java -classpath lib/aaa.jar;lib/bbb.jar`
- Boot classpath loads the JDK - **`rt.jar`**
- Extension classpath loads standard extensions
- User classpath loads user code
  - Defaults to current directory
  - Typically specified using **`-classpath`** or **`-cp`**
  - Can be jar files or directories

ClassLoader

- Every class is loaded by one class loader
- Most class loaders delegate to a parent
- One class loader for each of the three classpaths

ClassLoader

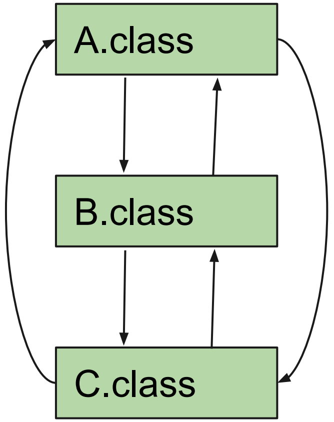


- Get class from ClassLoader
- Look in parent first
- Searching can be slow

Questions

- What happens if class not found on classpath?
- What happens if class appears twice on classpath?
- Does order matter in the classpath?
- What happens if same class is loaded by two different class loaders?

Classpath



Once on the classpath

- A can call B and vice versa
- A can call C and vice versa
- B can call C and vice versa

Jar files meaningless

Requirement 1

- Reliable configuration
  - replace classpath with graph of dependencies
  - know what is needed to run the application
- When application starts
  - check everything present
  - check no duplicates
  - potentially perform lookup optimisations

Packages are all we have

- Applications & libraries need to be multi-package
- More things public than desirable
- Often see packages with "impl" or "internal" in name
- Clear sign of a missing language feature

Example

- Jackson JSON/XML serialization
  - com.fasterxml.jackson.databind
  - com.fasterxml.jackson.databind.deser
  - **com.fasterxml.jackson.databind.deser.impl**
  - com.fasterxml.jackson.databind.jsontype
  - **com.fasterxml.jackson.databind.jsontype.impl**
  - com.fasterxml.jackson.databind.ser
  - **com.fasterxml.jackson.databind.ser.impl**
  - com.fasterxml.jackson.databind.ser.std
  - com.fasterxml.jackson.databind.type
  - com.fasterxml.jackson.databind.util

Example

- You were never supposed to use code in:
    - sun.misc.*
    - jdk.internal.*
    - etc.

Requirement 2

- Strong encapsulation
    - module can declare an API to other modules
    - packages not on the API are hidden
- Use this to enhance security
    - non-public elements cannot be accessed

JPMS basics

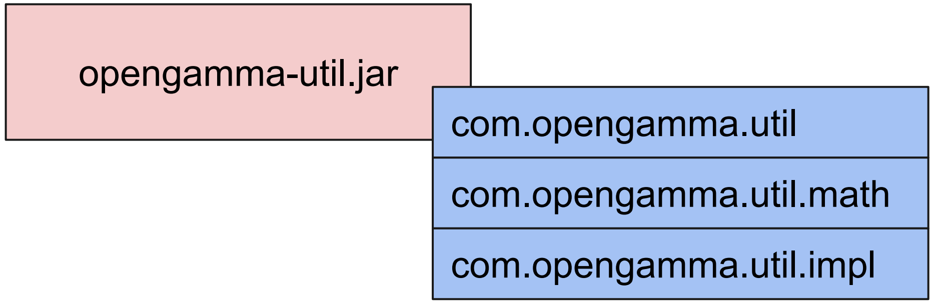
https://www.flickr.com/photos/distillated/4019168958/

What is a module?

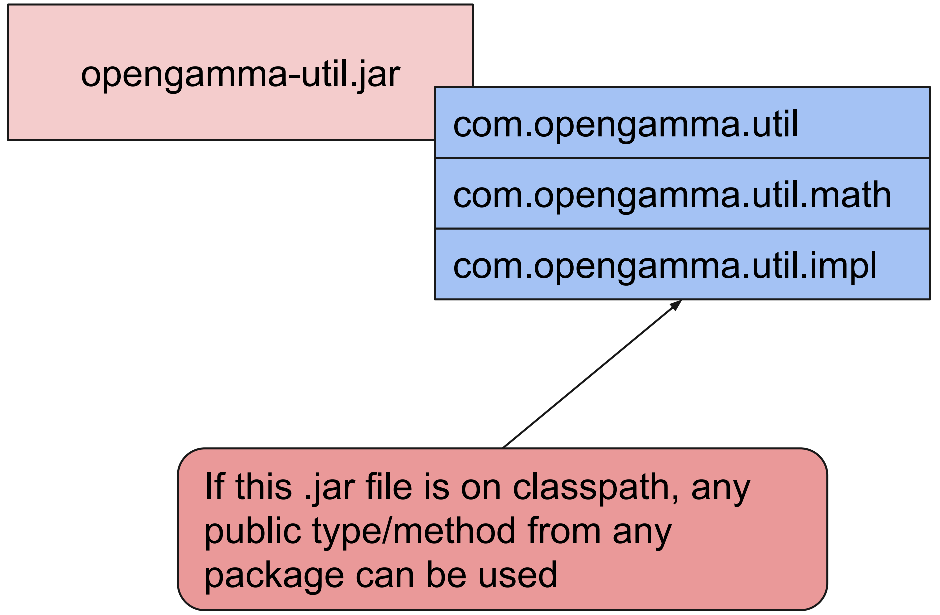
- Named
- Set of packages (classes/interfaces)
- Module metadata (module-info.class)

- Typically packaged as a .jar file
- Enforced by the JVM
    - JVM understands classes, interfaces, packages, and now modules

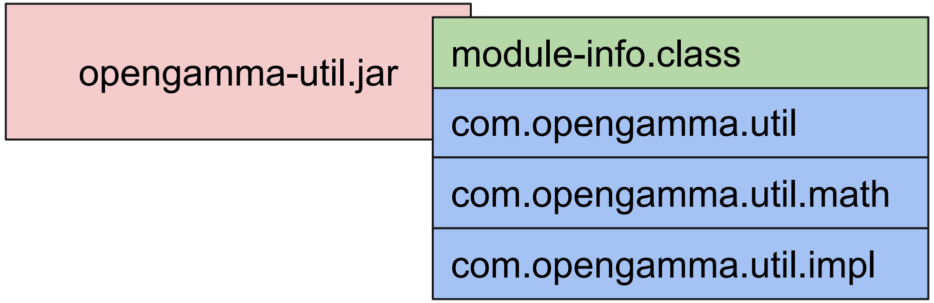
Without modules



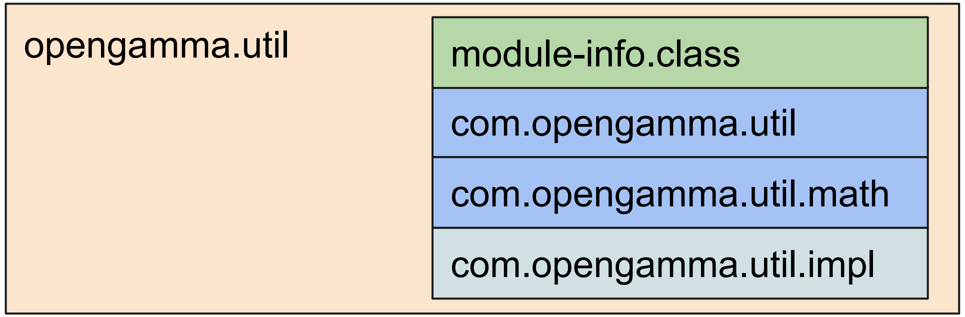
Without modules


opengamma-util.jar
com.opengamma.util
com.opengamma.util.math
com.opengamma.util.impl
If this .jar file is on classpath, any public type/method from any package can be used

With modules

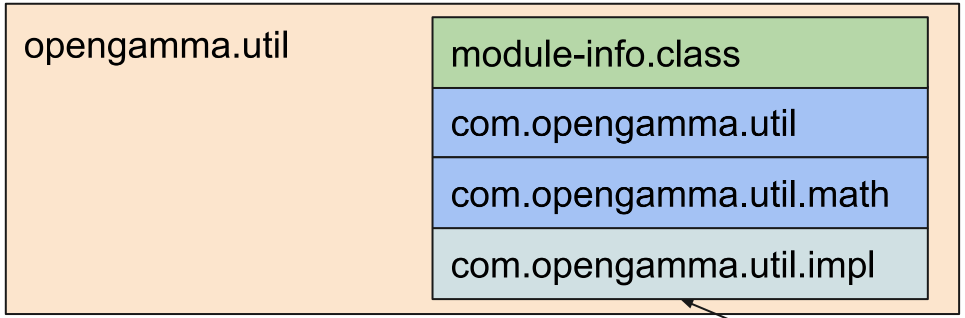


With modules



```
module opengamma.util {
  exports com.opengamma.util;
  exports com.opengamma.util.math;
}
```

With modules



```
module opengamma.util {
  exports com.opengamma.util;
  exports com.opengamma.util.math;
}
```

Non-exported packages cannot be seen outside the module

How is it enforced

- Modulepath as well as classpath
- Jar must have a module-info.class
- Module rules enforced when jar on modulepath
- Rules not enforced when jar on classpath
- JVM determines and validates module graph
  - checked at startup
  - advanced use cases can alter parts of the graph at runtime

Modular JDK

- JDK modules always run in module mode
- Classpath sees whole JDK
- Modules specify which JDK modules they need
- Changes JDK structure
    - no more `rt.jar`
    - packaged as `.jmod` files
    - no more boot classpath
    - no more extension classpath

JPMS module-info

https://www.flickr.com/photos/distillated/4019168958/

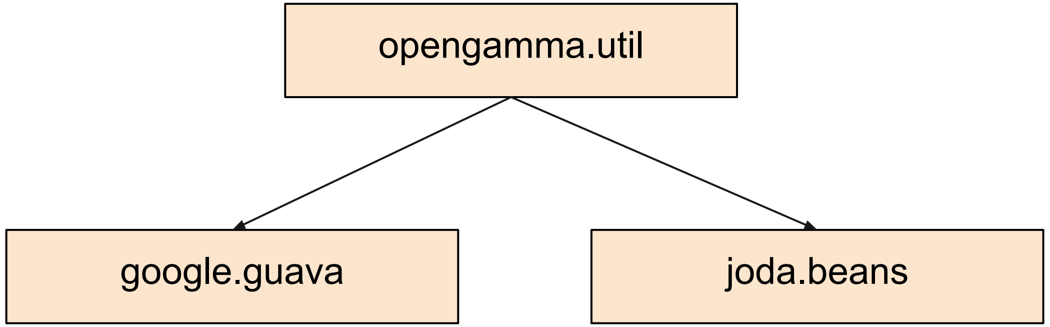
Creating a module

- Two basic questions
- What modules does it depend on?
  - depends on **`java.base`** by default
- What packages does it export?
  - nothing exported by default

Module metadata

```
module opengamma.util {
  // other modules this module depends on
  requires google.guava;
  requires joda.beans;
  // packages this module exports to other modules
  exports com.opengamma.util;
}
```

Module graph

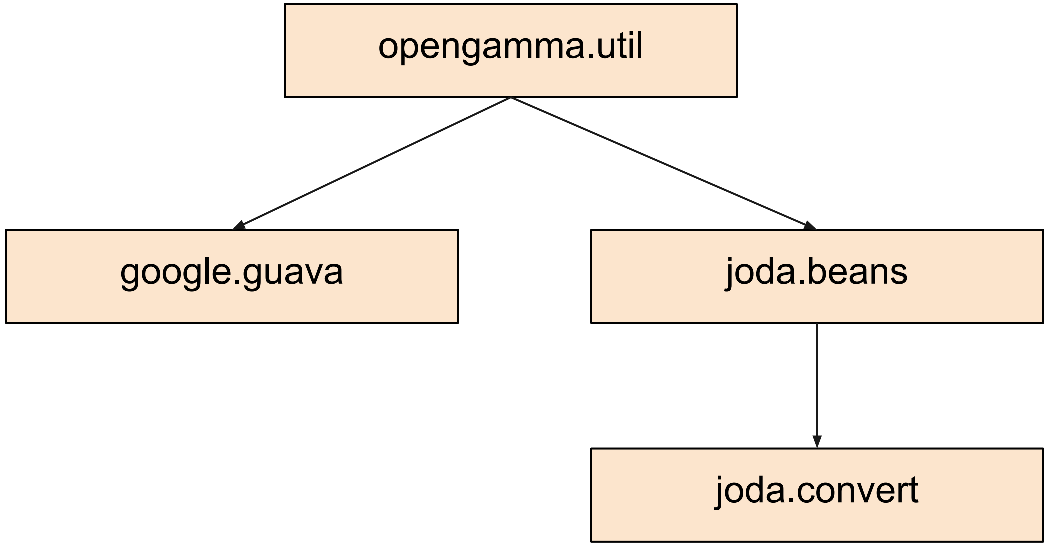


Dependencies

- **`org.joda.beans`** depends on **`org.joda.convert`**

```
module joda.beans {
  // other modules this module depends on
  requires joda.convert;
  // packages this module exports to other modules
  exports org.joda.beans;
}
```

Module graph

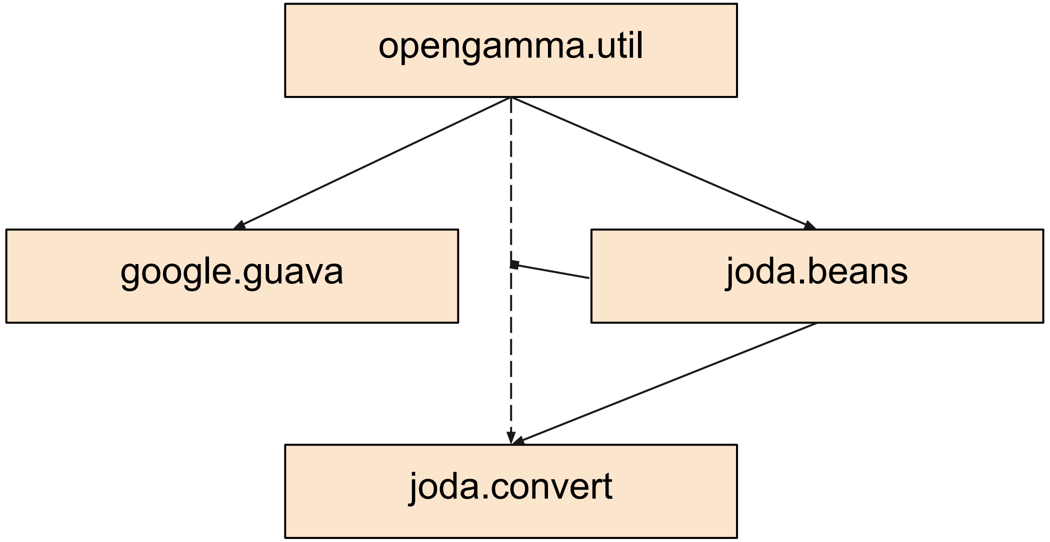


Transitive dependencies

- But, Joda-Beans exposes types from Joda-Convert
- Express concept using "requires transitive"

```
module joda.beans {
  // other modules this module depends on
  requires transitive joda.convert;
  // packages this module exports to other modules
  exports org.joda.beans;
}
```

Module graph

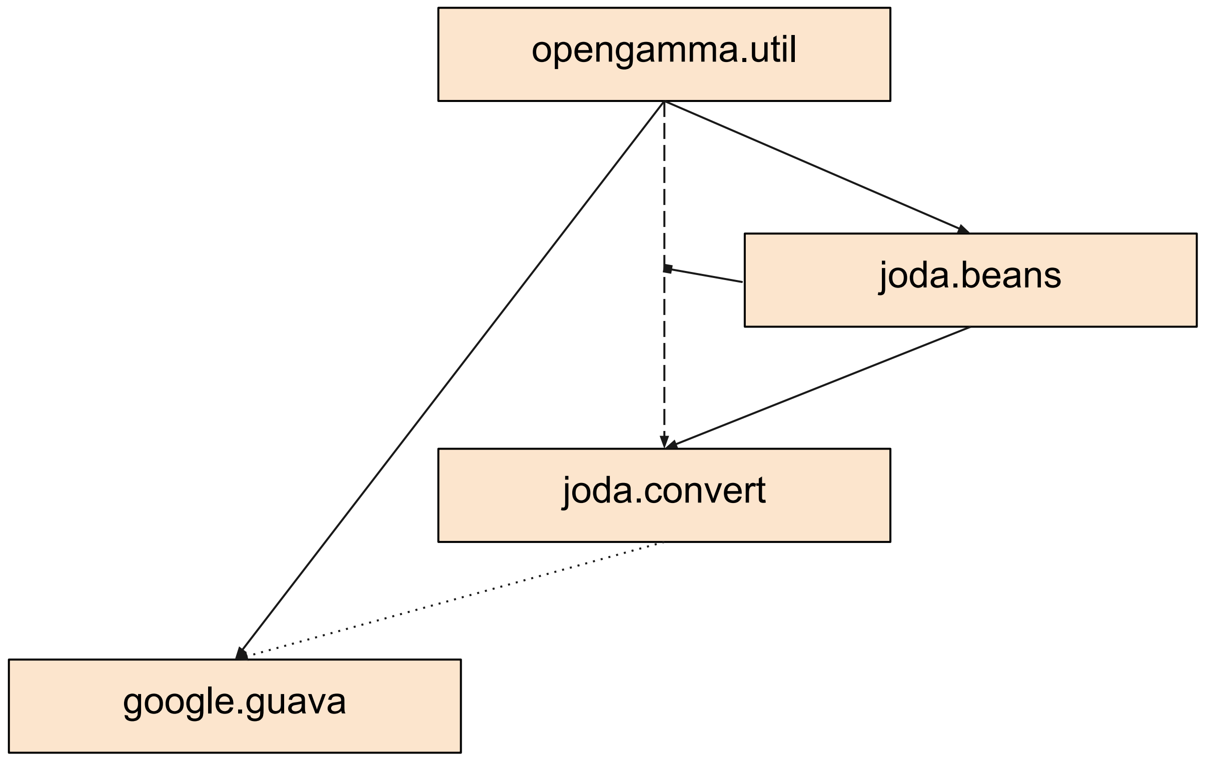


Optional dependencies

- But **`org.joda.convert`** has optional dependency
- If Guava exists at runtime, Joda-Convert adapts

```
module joda.convert {
  // other modules this module depends on
  requires static google.guava;
  // packages this module exports to other modules
  exports org.joda.convert;
}
```

Module graph



Checks

- Module path contains a set of modules
- One module is the root
- Other modules are resolved to form a graph
- System ensures all necessary modules are available
- System ensures no module found twice
- System ensures same package not in two modules
- System ensures graph has no cycles

Modulepath

- A coherent modulepath is your problem
- Versions, and their selection, not part of JPMS
    - do not put version in module name
- Typically will be done by Maven or Gradle
- Just like the classpath is assembled today

Open to debate as to how much this achieves the reliable configuration requirement

Services

- SQL drivers, XML parsers, etc.
- Can be specified in module-info
- Module that contains the service implementation:
    - `provides com.foo.XmlParser with com.foo.MyXmlParser`
- Module that uses the service:
    - `uses com.foo.XmlParser`

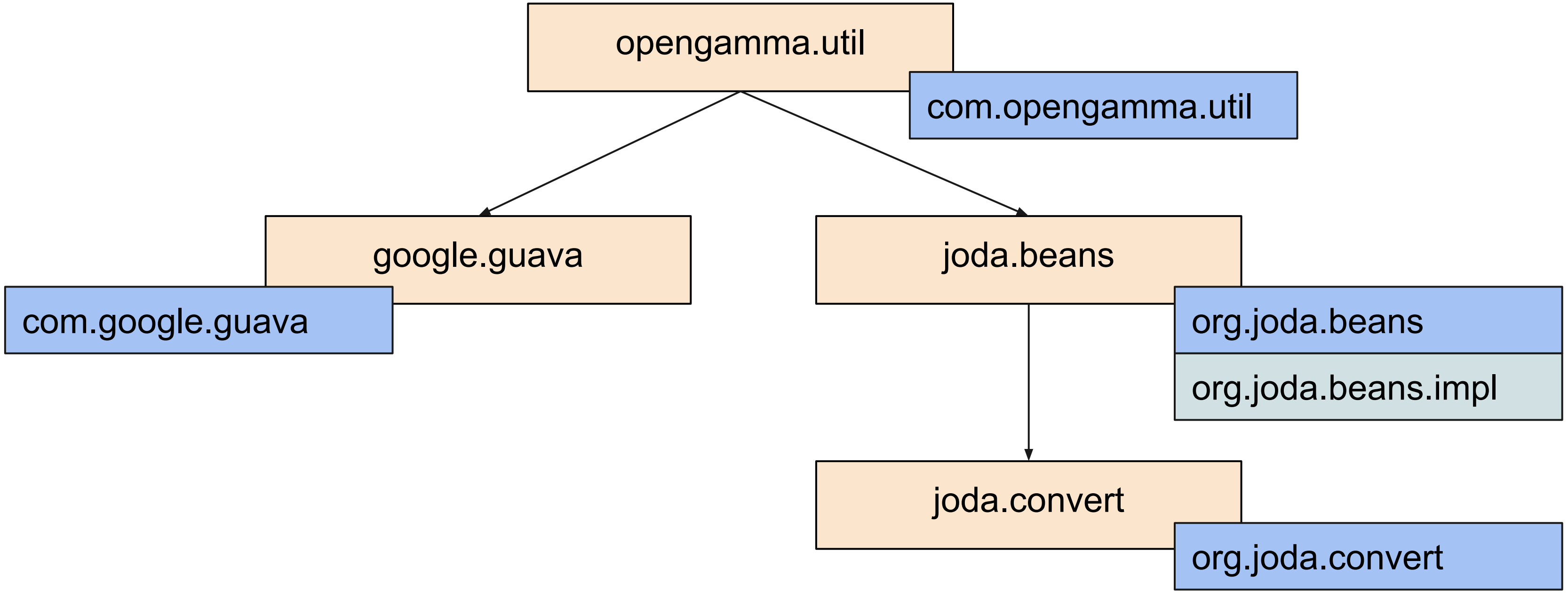
Calling code in another module

https://www.flickr.com/photos/distillated/4019168958/

Hiding packages

- Given two packages
  - `org.joda.beans`
  - `org.joda.beans.impl`
- It is possible to completely hide the internal package
  - export `org.joda.beans`
  - do not export `org.joda.beans.impl`

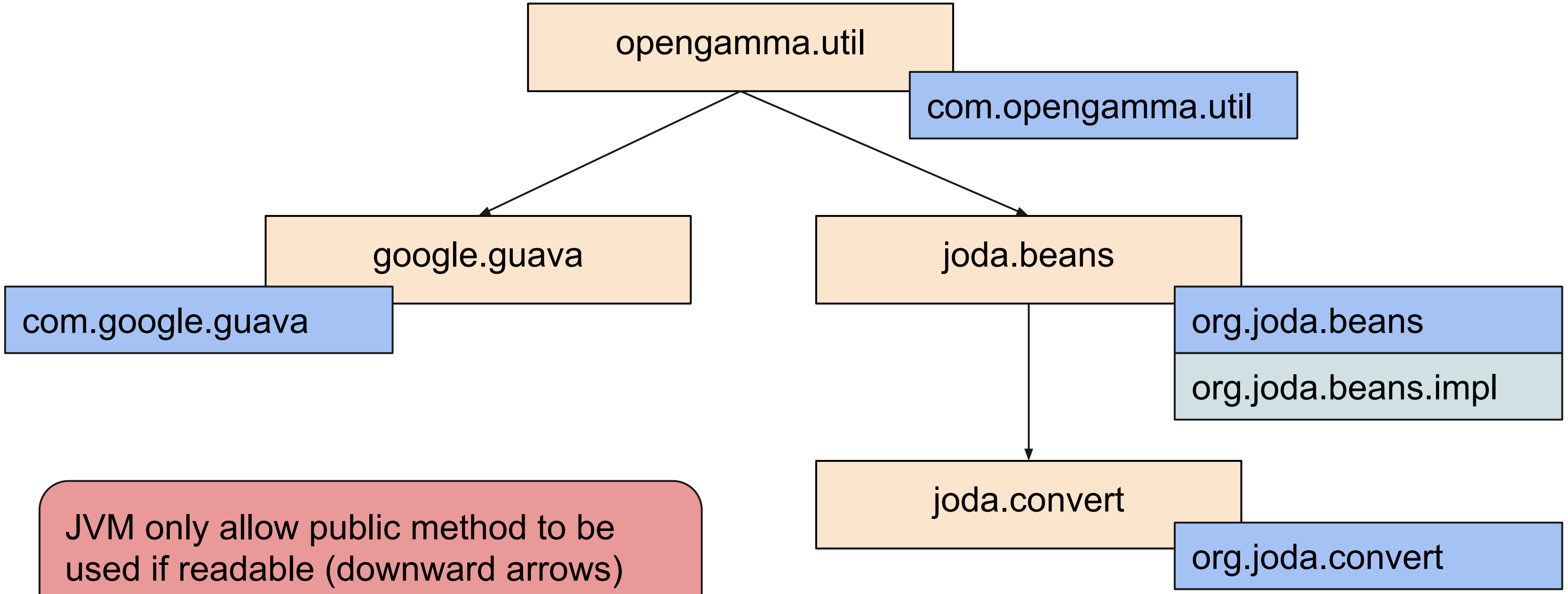
Hiding a package



Access

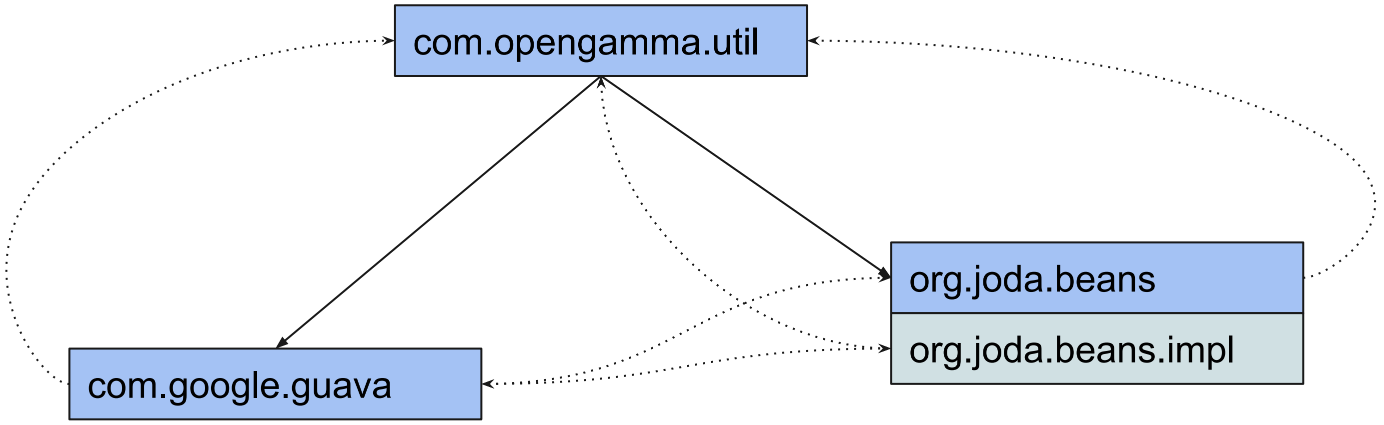
- Only exported packages are visible to other modules
- All other packages are private to the module
- **"public" no longer means "public"**
- Other modules can only see and use code if:
  - public
  - package is exported by target module
  - package is readable from this module
- Combination referred to as "accessibility"

Access



JVM only allow public method to be used if readable (downward arrows) and package is exported (dark blue)

Access



JVM prevents calls along dotted lines

Targetted exports

- Package can be exported to a specific module:
  - `exports org.joda.beans.impl to joda.special`
- Effectively a kind of "friend" access
- Usually suggests poor package structure

Lockdown

- Ability to lockdown packages is powerful
- However, it only works on the modulepath
- Paranoid code could refuse to work on classpath

Reflection

https://www.flickr.com/photos/distillated/4019168958/

Reflection

- Powerful mechanism to access code
- Access package and private scope
  - use **`setAccessible()`**
- Vital to frameworks, particularly for bean creation

Reflection in 9

- Can access public elements of exported packages
- Cannot access package/private types/members
- Cannot access non-exported packages
- Much stronger barrier than previously
  - **`setAccessible()`** does not help

Reflection

- Modules can selectively allow reflection
- Use "opens", also enables setAccessible()

```
module joda.beans {
  // other modules this module depends on
  requires transitive joda.convert;
  // packages this module exports to other modules
  exports org.joda.beans;
  opens org.joda.beans.impl;
}
```

Reflection

- Modules can open all packages if desired

```
open module joda.beans {
  // other modules this module depends on
  requires transitive joda.convert;
  // packages this module exports to other modules
  exports org.joda.beans;
}
```

Reflection on the JDK

- JDK is fully modular
- JDK does not have "open" packages
- Thus, cannot reflect on non-public JDK
- Big security boost
- Breaks many existing libraries
- Can use command line flags to force JDK open
    - useful in the short term until the libraries are fixed

Reflecting on modules

- Module information is available to reflection
  - new **`Module`** class and **`getModule()`** method
  - allows the JVM module graph to be accessed
- See also the **`ModuleLayer`** concept
  - intended for advanced use cases

Migration

https://www.flickr.com/photos/distillated/4019168958/

Migration

- Take existing code and migrate it
- New rules make this tricky in some cases
    - eg. SLF4J will need major rework

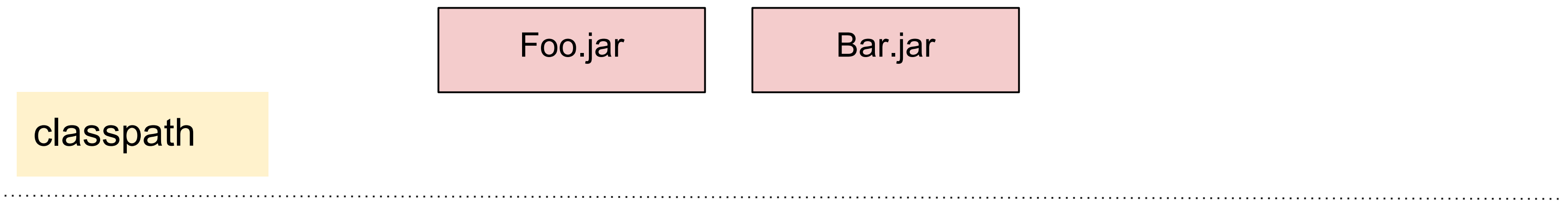
Use classpath

- Most existing code will run on Java SE 9 classpath
- If it uses internal JDK API, fix or use a flag:
  - `--add-exports`, `--add-opens`, `--permit-illegal-access`
- If it uses removed JDK module, fix or use a flag:
  - `--add-modules java.xml.bind`

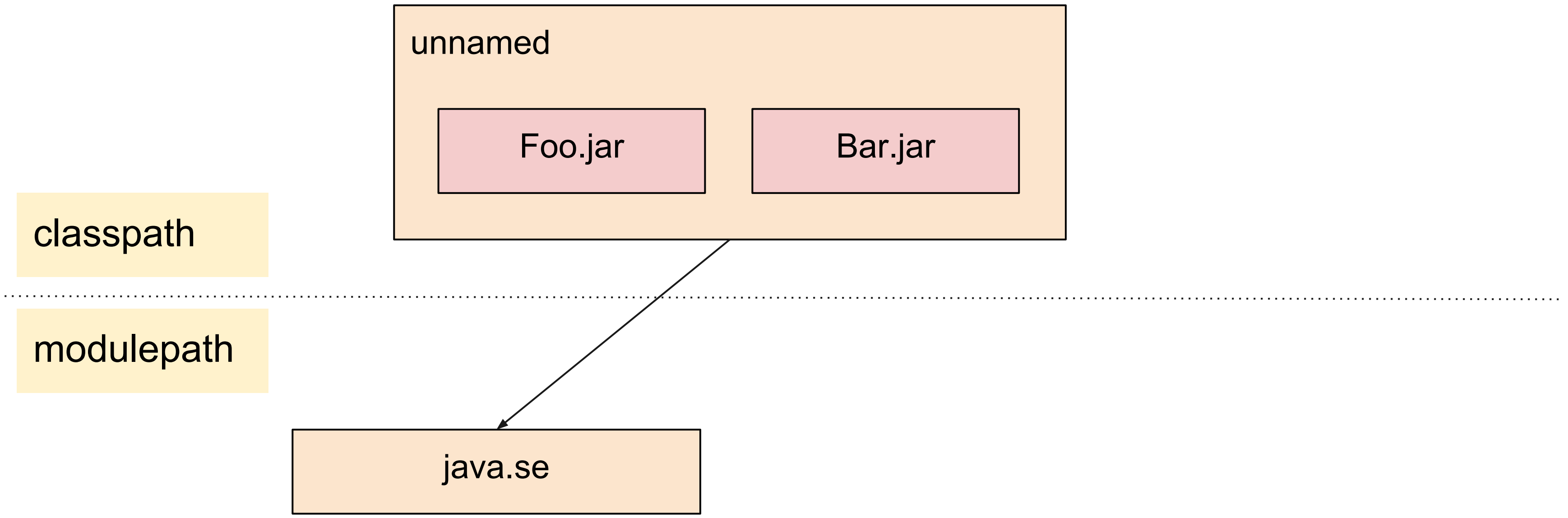
Migration

- Classpath mapped to the "unnamed module"
- Unnamed module reads all other modules
- Cannot access non-exported packages
- Most existing code should run fine in unnamed module
- Named modules cannot access the unnamed module

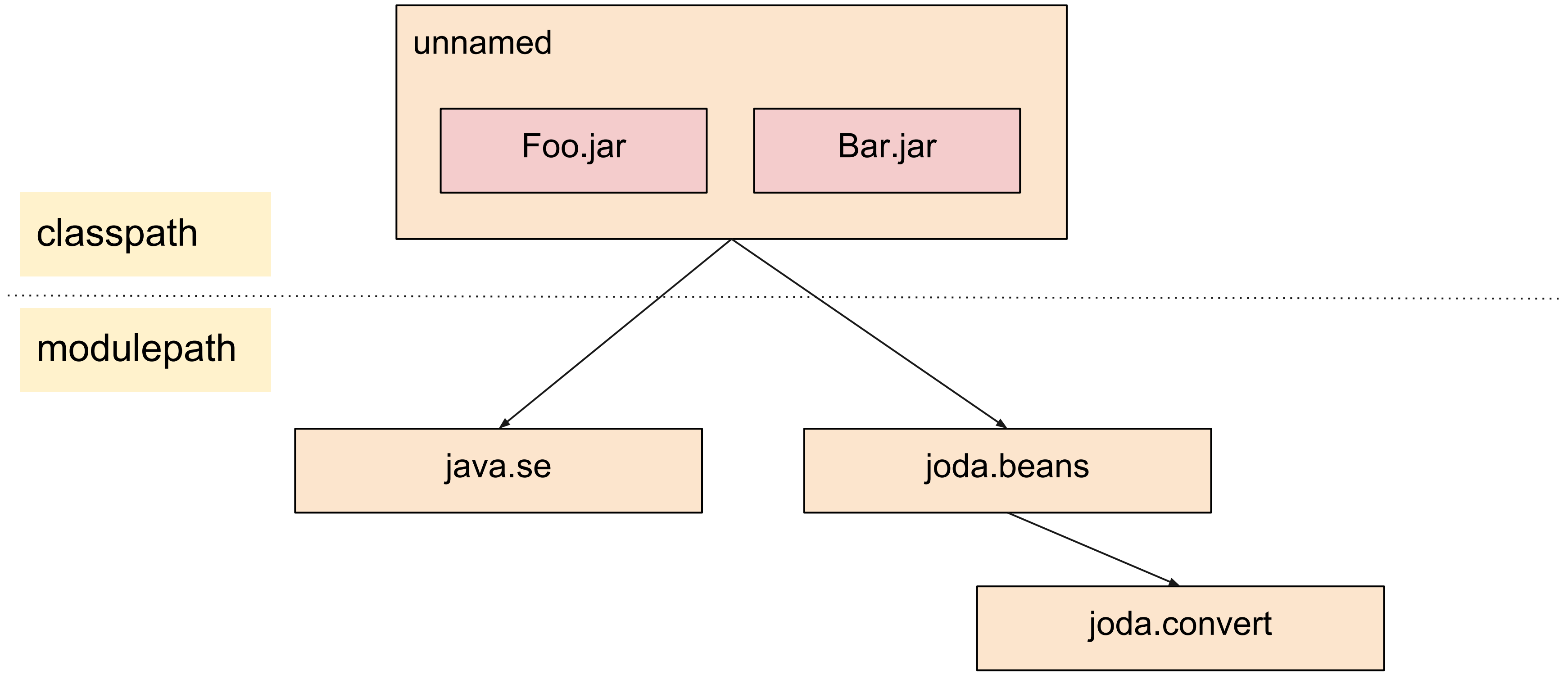
Classpath



Classpath



Classpath

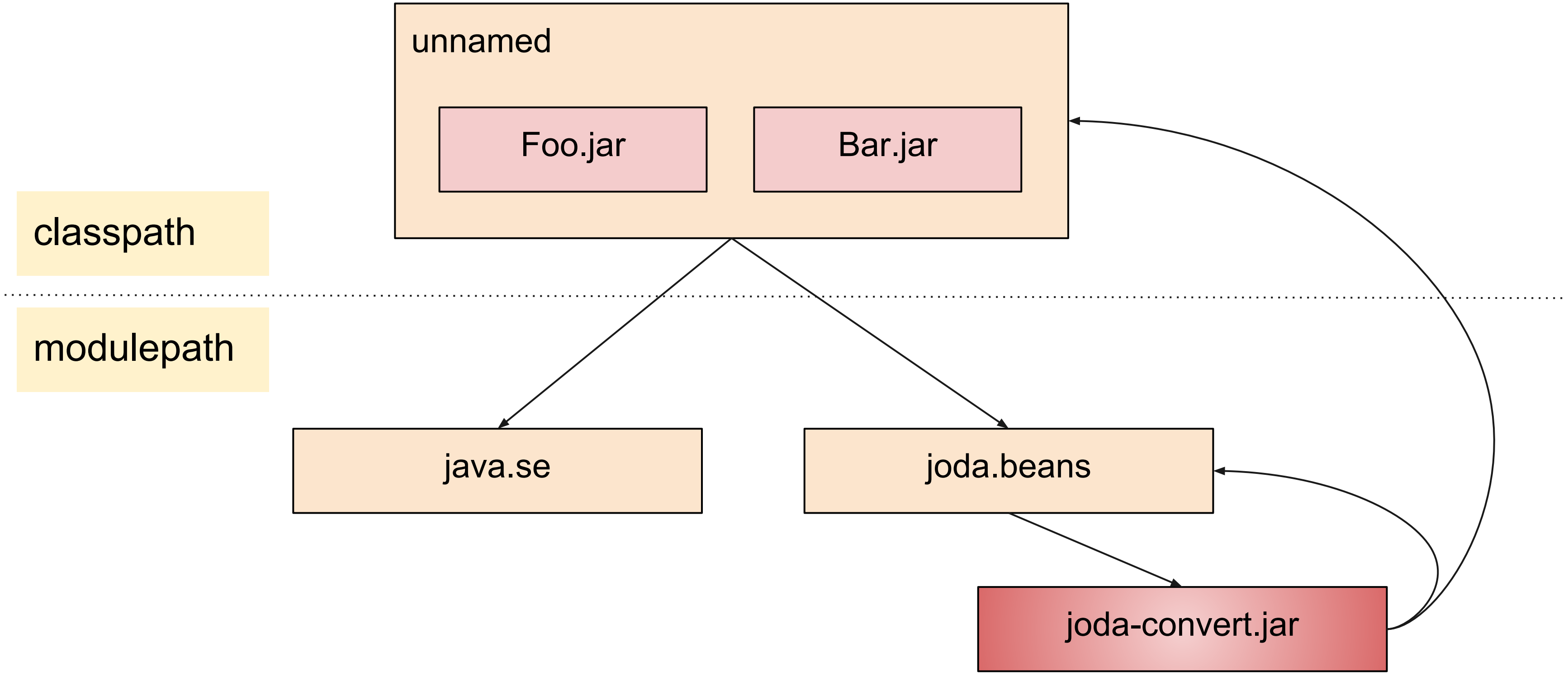

unnamed
Foo.jar
Bar.jar
classpath
modulepath
java.se
joda.beans
joda.convert

Migration

- Additional "automatic module" concept
- Jars without module-info on modulepath
- Exports all packages in jar
- Reads the all other modules, including unnamed
- Used to handle libraries that have not yet modularized

WARNING! There are serious concerns about using automatic modules in open source projects. Maven Central will likely ban projects that depend on an automatic module.

Classpath


unnamed
Foo.jar
Bar.jar
classpath
modulepath
java.se
joda.beans
joda-convert.jar

Module naming

- No agreement on module names yet
- Please wait until clear convention agreed
- Might be like package names
  - org.joda.convert
- Might be shorter, but this will lead to clashes
  - joda.convert

WARNING! There are still serious naming concerns from the open source community.

Modules and Maven

- Maven creates a tree of dependencies
- Often enforced by IDEs
- Not enforced at runtime
- Likely that each pom.xml will be a JPMS module
    - unclear whether module-info.java will be hand written
    - or derived from pom.xml (I suspect hand written, with IDE help)
- JPMS does not handle versions
- JPMS does not select jar files from Maven repo

Modules and OSGi

- OSGi is a module system based on the classpath
- It still works fine on the classpath
- Dynamic loading, lifecycle etc. not part of JPMS
- JPMS is not ClassLoader based
    - all classes in a module have the same ClassLoader
    - same ClassLoader may be used by many modules

Summary

https://www.flickr.com/photos/distillated/4019168958/

Summary

- Modules are a new JVM concept, exposed in Java
    - don't confuse with Maven, OSGi, JBoss modules
- Moving to modules is non-trivial
    - no reflection on JDK internals without command line flag
    - parts of the JDK removed by default and need command line flag
- Reflection is being restricted
    - command line flags allow this to be broken
- Don't rush to modularize!
    - wait for open source to do so first

Summary

- Project Jigsaw:
  - http://openjdk.java.net/projects/jigsaw/
  - Time to test Java SE 9 for bugs is NOW!
- Stephen Colebourne:
  - @jodastephen - feedback & questions
  - http://blog.joda.org
- OpenGamma
  - Strata - open source market risk analytics
  - Cloud-based metrics for derivatives trading and clearing