Paremus

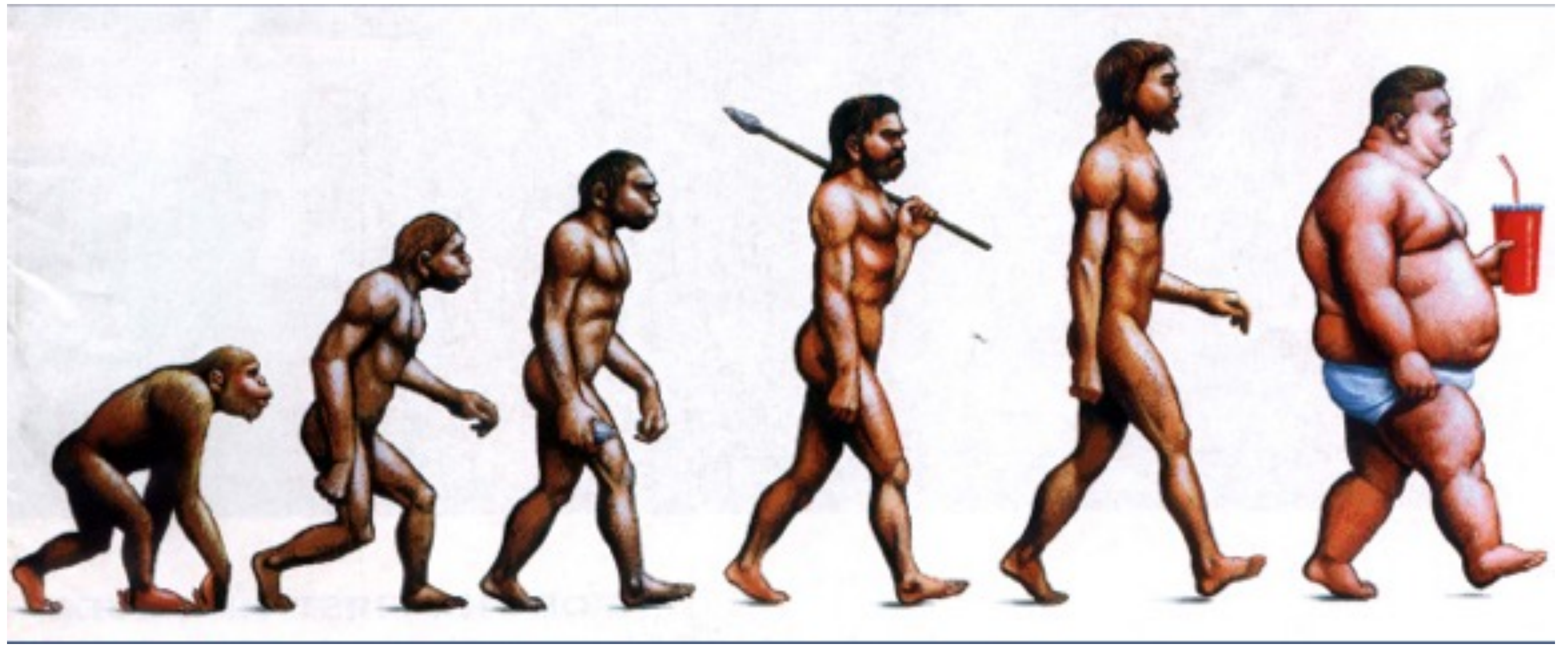
OSGi, Java 9 and the Future of Modularity

Neil Bartlett

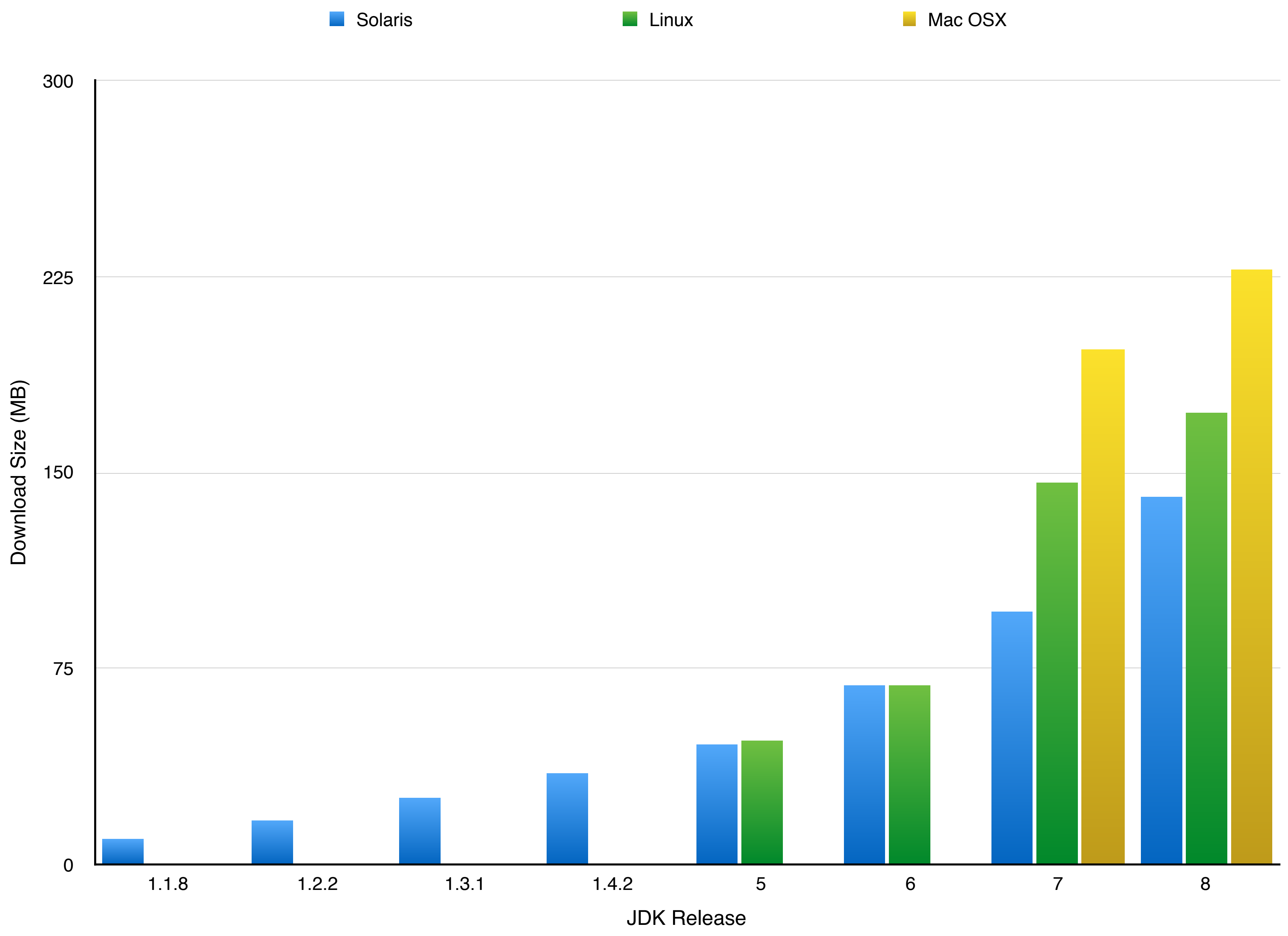
http://www.paremus.com
info@paremus.com

Introduction

Java has a Problem…



Java’s Getting Larger



Who Really Cares?

- **Not Enterprise.**
- Disk and memory are cheap.
- Competition is mostly .Net.

IoT Cares…

- Limited memory, CPU.
- Little or no storage!
- Economies of scale… 100,000s of devices * $5 saving per device?

- Java's competition is no longer (just) .Net.
- Includes NodeJS, Go, Rust…

Java Modularity

- OSGi: Started in late 1990’s with JSR 8.
- OSGi Alliance formed and Release 1 published in 2000.
- Now on Release 6, working towards Release 7.
- … but we never could modularise the JDK!

- Reasons include political, commercial and technical concerns.
- Let’s just focus on the technical.

JDK Modularity

- Sun (later Oracle) led several aborted projects in mid 2000s.
- JSR 294 “Superpackages” – dead
- JSR 277 “Java Module System” – dead
- JSR 376 “Java Platform Module System” (**JPMS**).
- Jigsaw prototype — targeting Java 7 in 2011.
- Slipped to Java 8.
- Slipped again to Java 9.
- Java 9 delayed by a year (so far).
- Clearly not a trivial problem!

JPMS and OSGi

- JPMS's primary goal is to modularise the **JDK**.
- It can also be used by libraries and applications.
- So how does this affect OSGi and its users?

Basics

What’s a Module?

“A unit of encapsulation that communicates with other modules through agreed contracts.”

What’s a Module?

“A unit of **encapsulation** that **communicates** with other modules through agreed **contracts**.”

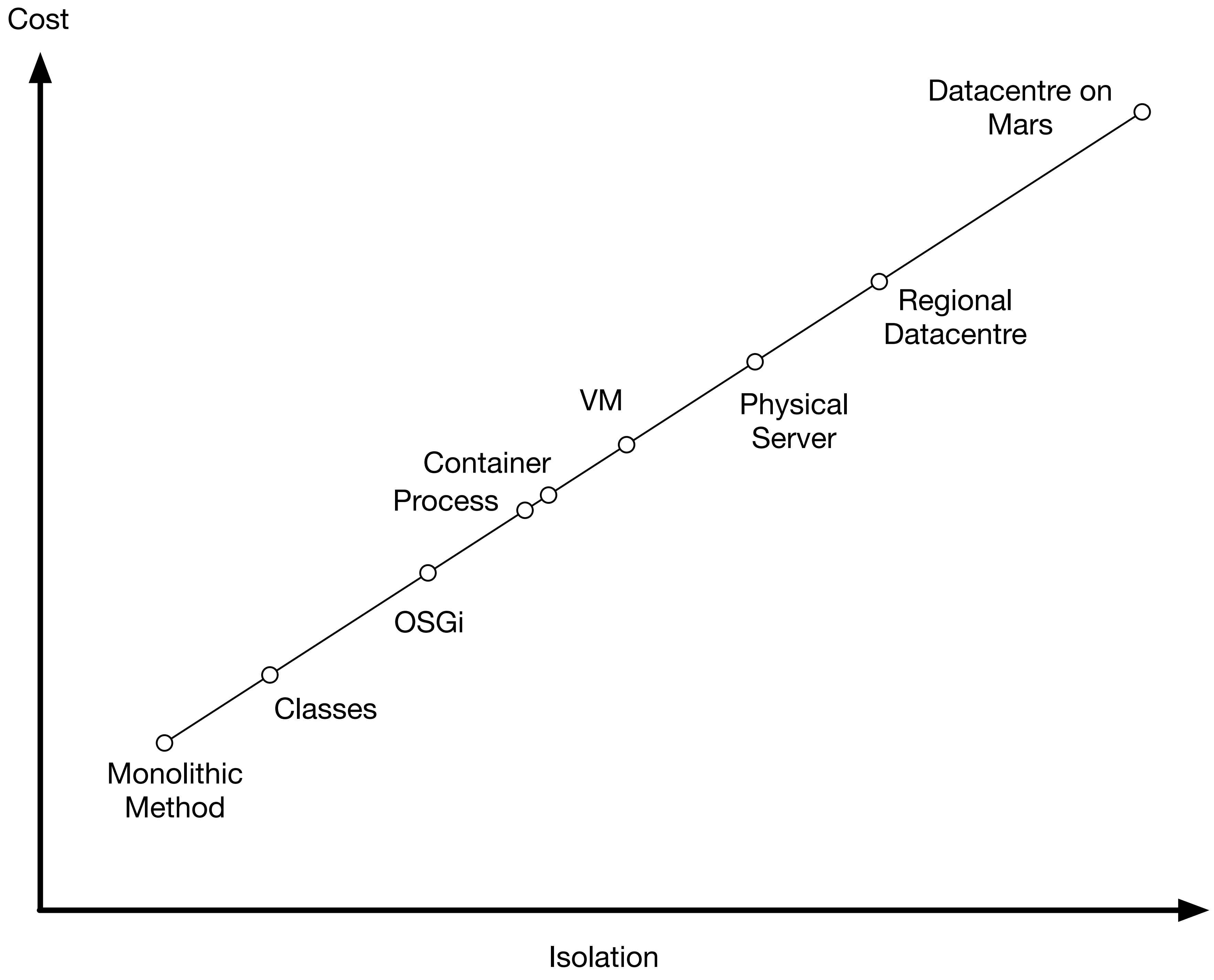
What’s NOT a Module?

- **Monolithic** Java applications (classpath).
- No encapsulation – **everything** can interact with **anything**.
- Communication is ad hoc.
- Contracts **may** be used.

Encapsulation / Isolation

Isolation

- Isolation is a **continuum**.
- Principle – Freedom from Interference
- Isolation has a cost!

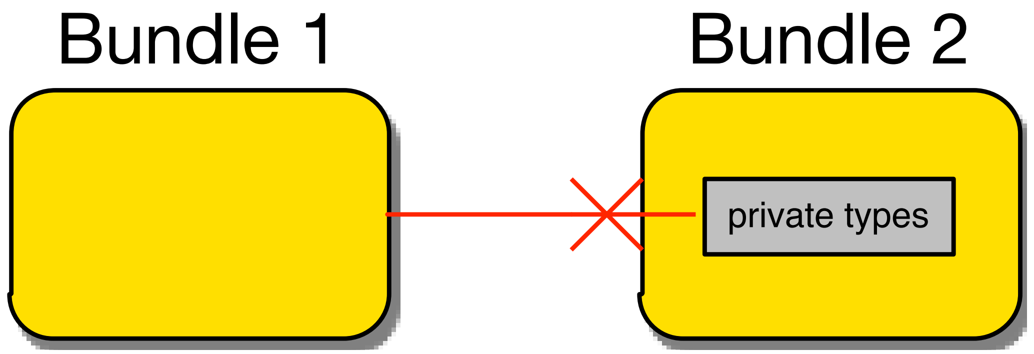


Isolation

- OSGi and JPMS provide code-level isolation.
- **Neither** prevents modules from:
  - Consuming all available memory;
  - Creating 1000s of threads;
  - Calling `System.exit()`!
- **Both do** prevent:
  - Accessing “internal” types from outside a module.

OSGi Isolation: Visibility

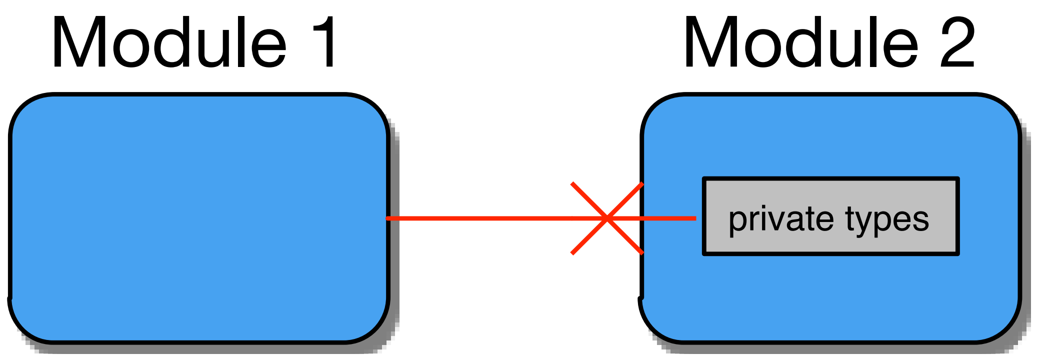
- OSGi creates a ClassLoader per bundle
- Each bundle has a *Class Space*: the set of classes visible to it.
- Equal to the private contents of the bundle + explicitly imported types.



- Bundle 1 cannot **see** Bundle 2’s private types.
- As if they **don’t exist**.

JPMS Isolation: Access

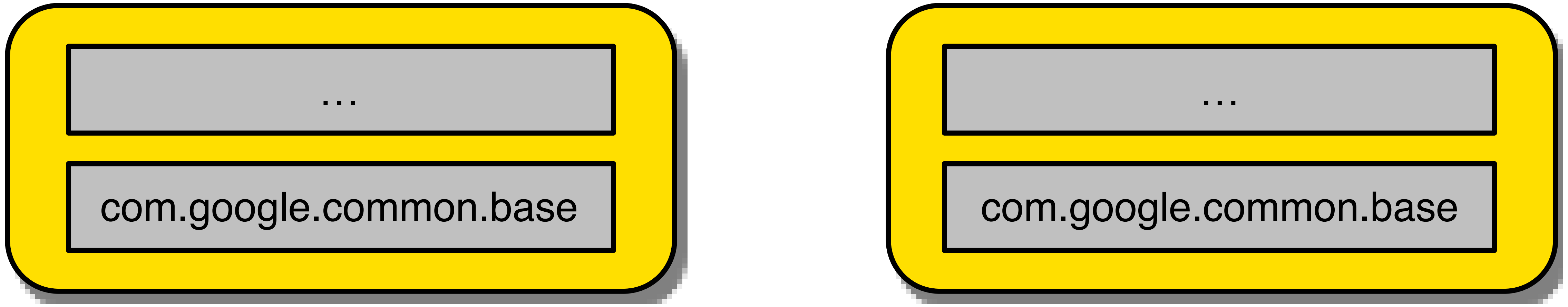
- Modules on the module path live in a **single** ClassLoader
- Modules can see but **not access** another module’s private types.



- `Class.forName()` … **works**!
- `clazz.newInstance()` … **fails** with IllegalAccessException.
- ICYWW, `setAccessible(true)` also fails.

Implications for OSGi

- In OSGi this works:

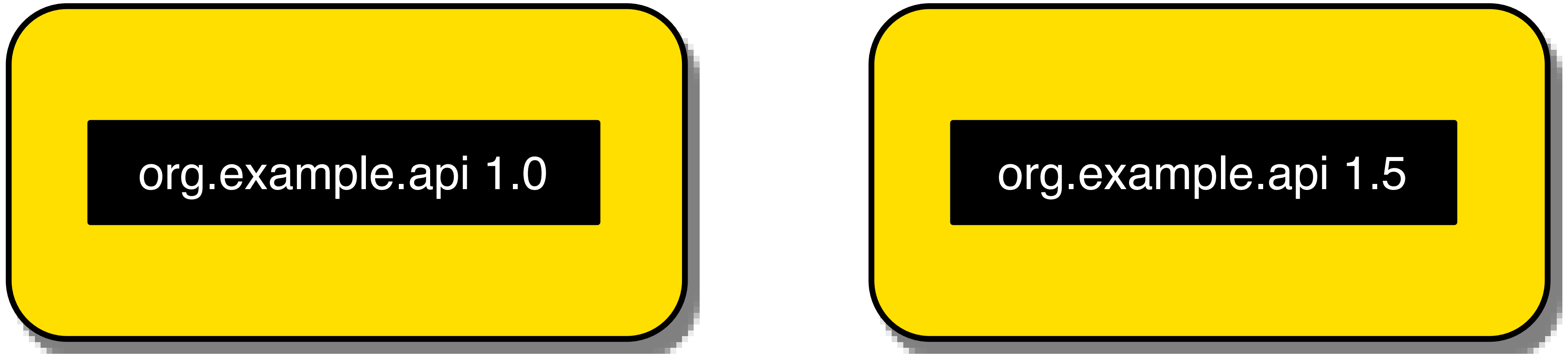


- Why would you do this?
  - Static linking is a useful technique;
  - Avoid external dependency;
  - Avoid versioning issues.

Legend:

Implications for OSGi

- In OSGi this also works:

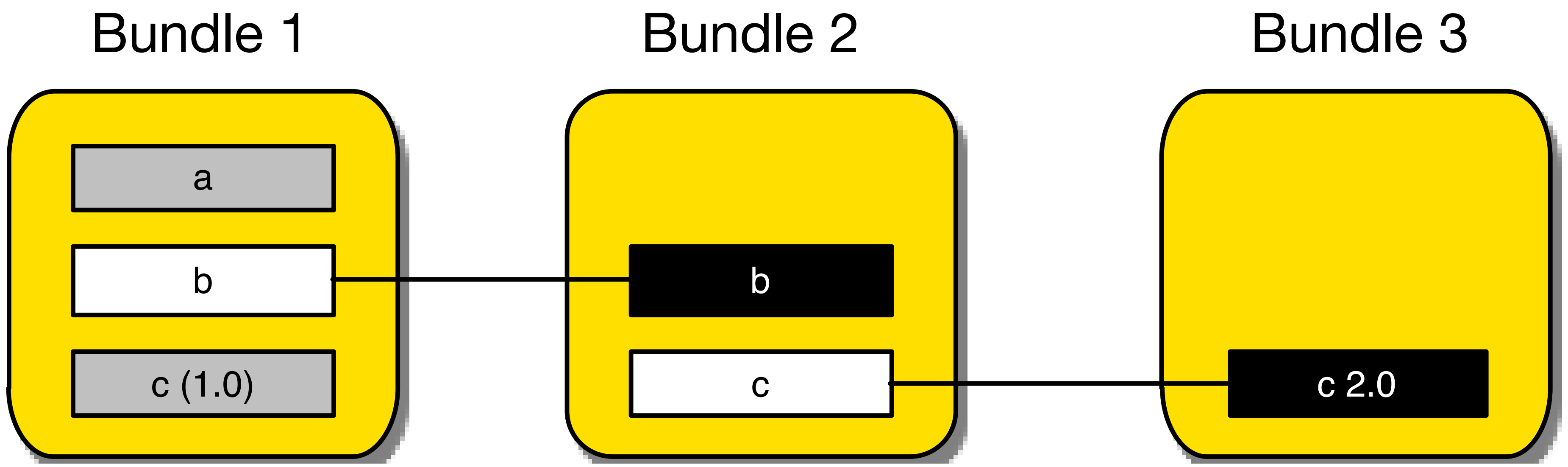


- Does add some complexity, but manageable.
- Sometimes our dependencies cannot be reduced to a single version of every API.

Legend: exported package

Implications for OSGi

- Even this works

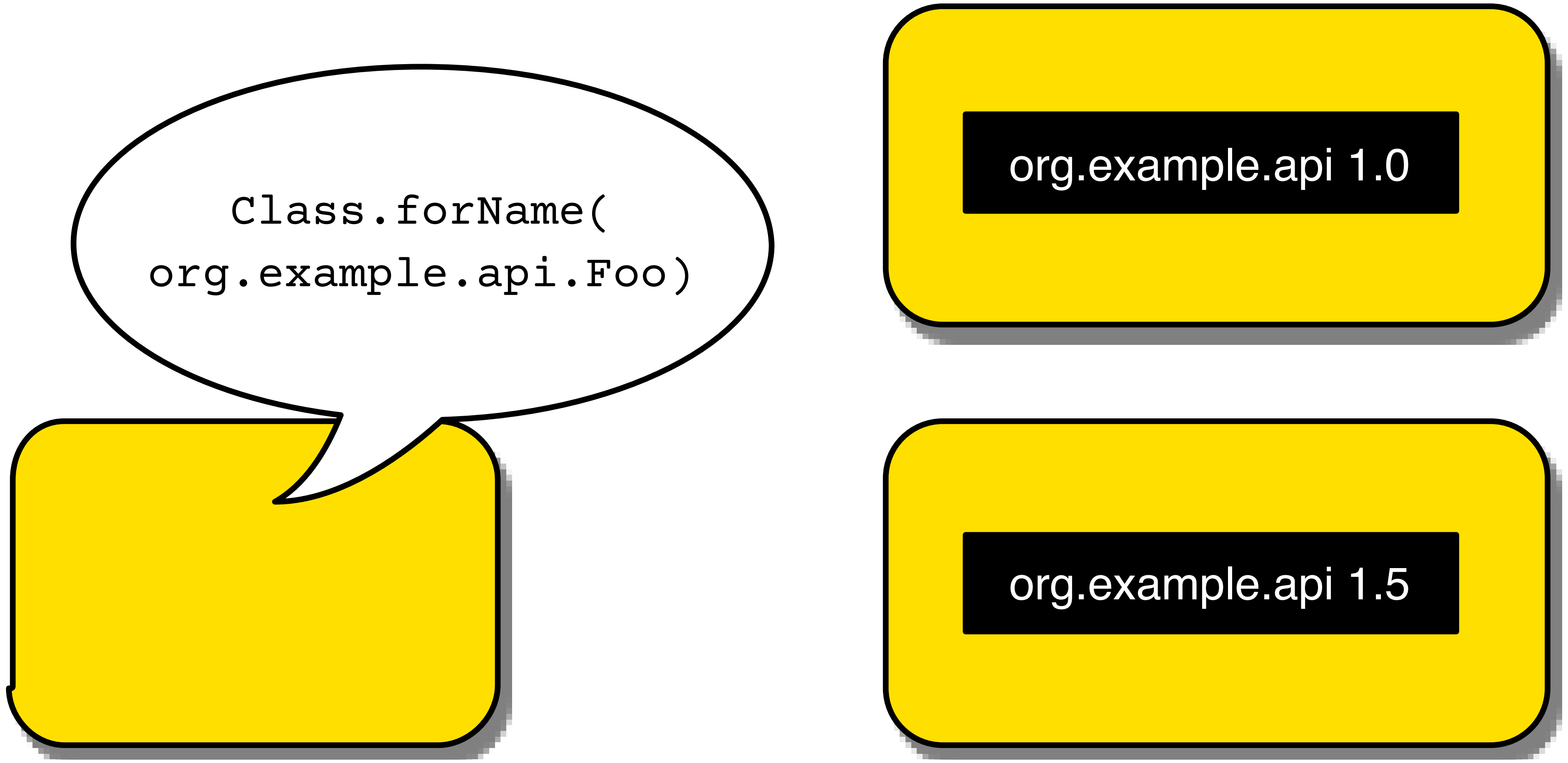


- B1 sees packages a, b and c(1.0).
- B2 sees packages b and c (2.0).
- B3 sees package c (2.0).
- Package c in B1 and B3 can be entirely different.

Legend: imported package

Implications for OSGi

- In OSGi this **doesn't** work:



Implications for OSGi

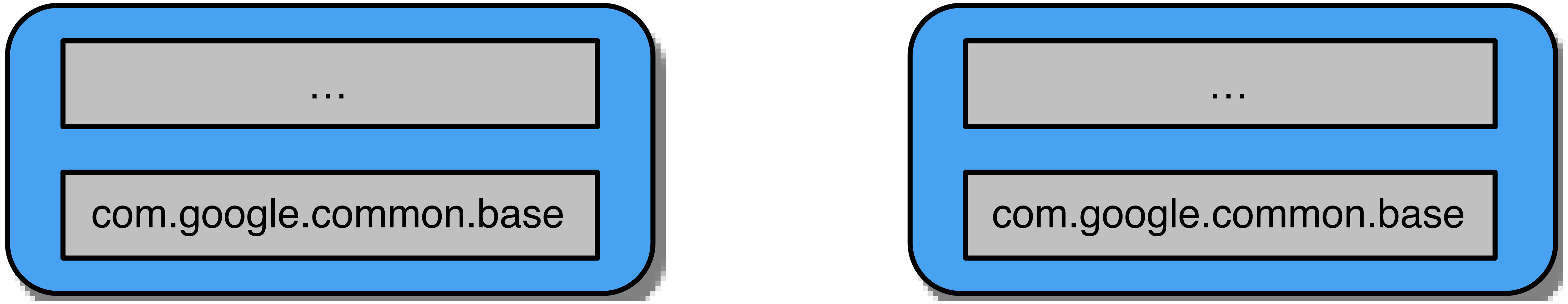
- Why not?
  - Type name is not unique.
  - If we specify a bundle we are fine!

- Why is this a problem?
  - Many libraries assume it “just works”.
  - OSGi ClassLoader is forced to guess. Sometimes it guesses wrong.

- Probably the biggest source of frustration for new OSGi users!
- But what are we to do??

Implications for JPMS

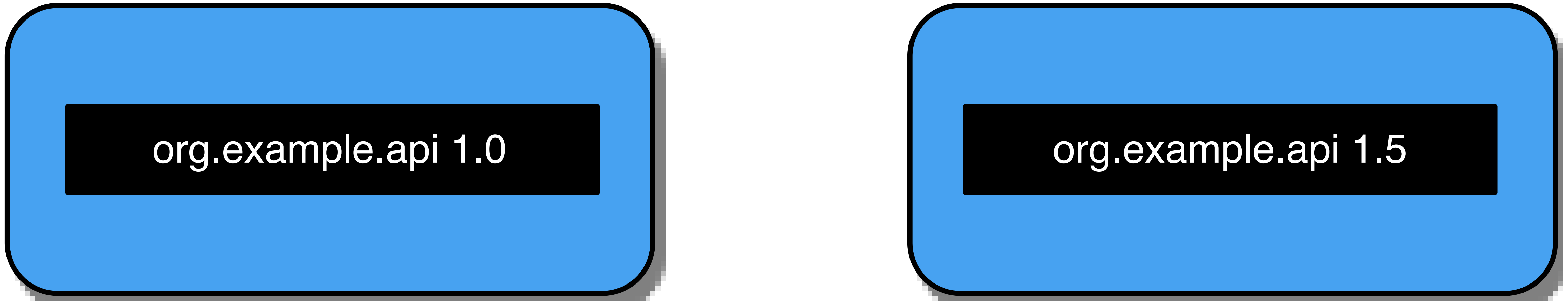
- In JPMS this **doesn’t** work:



- `java.lang.reflect.LayerInstantiationException: Package com.google.common.base in both module a and module b.`
- Unless we manage our own ClassLoaders.
- … like OSGi does!

Implications for JPMS

- And this doesn't work:



- Unless we manage our own ClassLoaders…

Contracts & Dependencies

Contracts & Dependencies

- **Any fool can build a wall.**
- Working together is harder.
- How do we reintroduce connections in a **controlled** way?

Exports

- Both JPMS and OSGi share Java **packages**.
- In both cases, any non-exported packages are “private”.

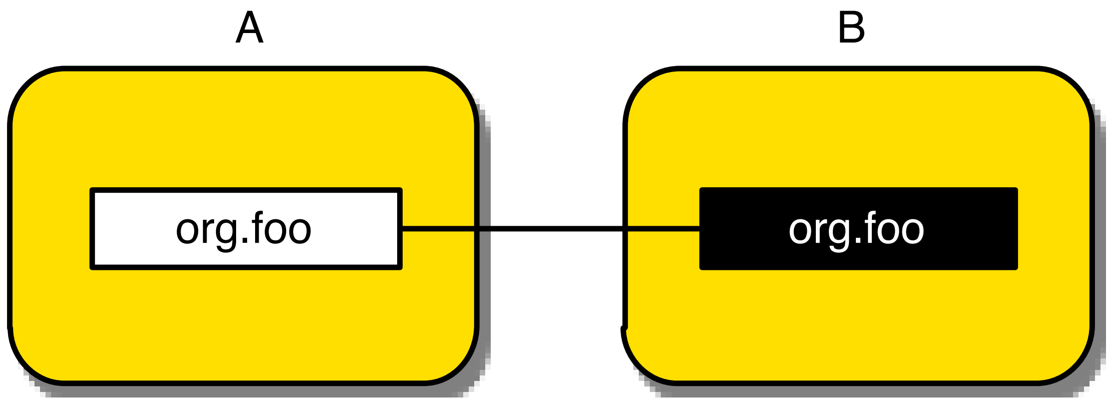
```
# OSGi
Export-Package: org.example.api


// JPMS
module A {
   exports org.example.api;
}
```

Imports

- In OSGi the complement of exporting packages is importing packages.
- This creates a “wire” across which class loading requests can be delegated.

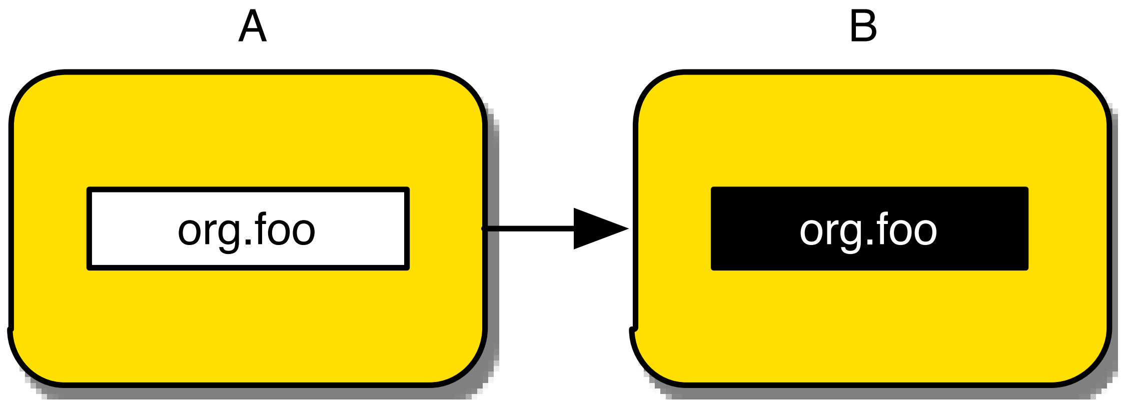
```
Import-Package: org.example.api
```



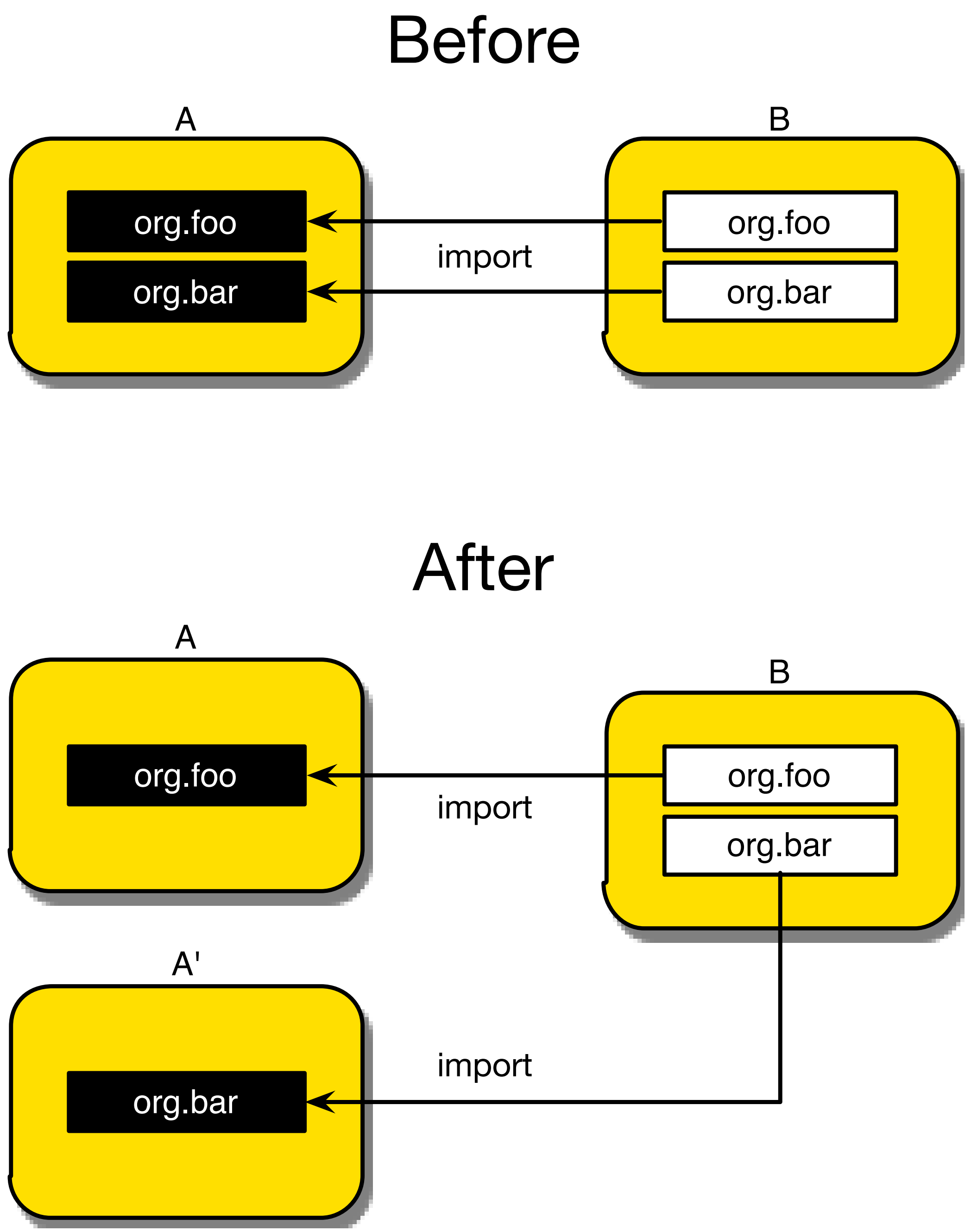
Requires

- OSGi also supports Require-Bundle. Imports all exported packages of a bundle.
- Require-Bundle is **deprecated**. Eclipse developers: this includes you!

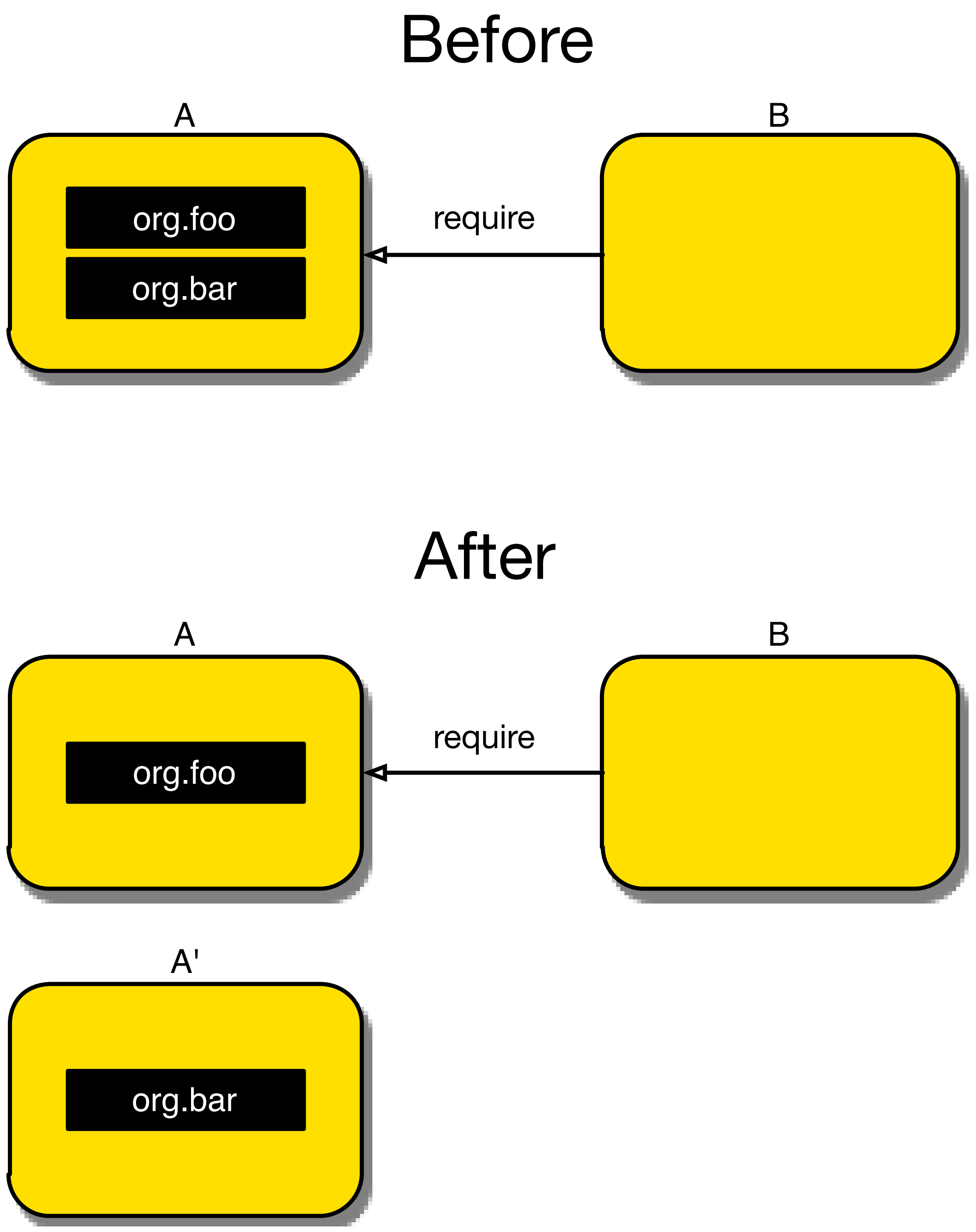
```
Require-Bundle: B
```



Refactoring with Import-Package



Refactoring with Require-Bundle



Requires

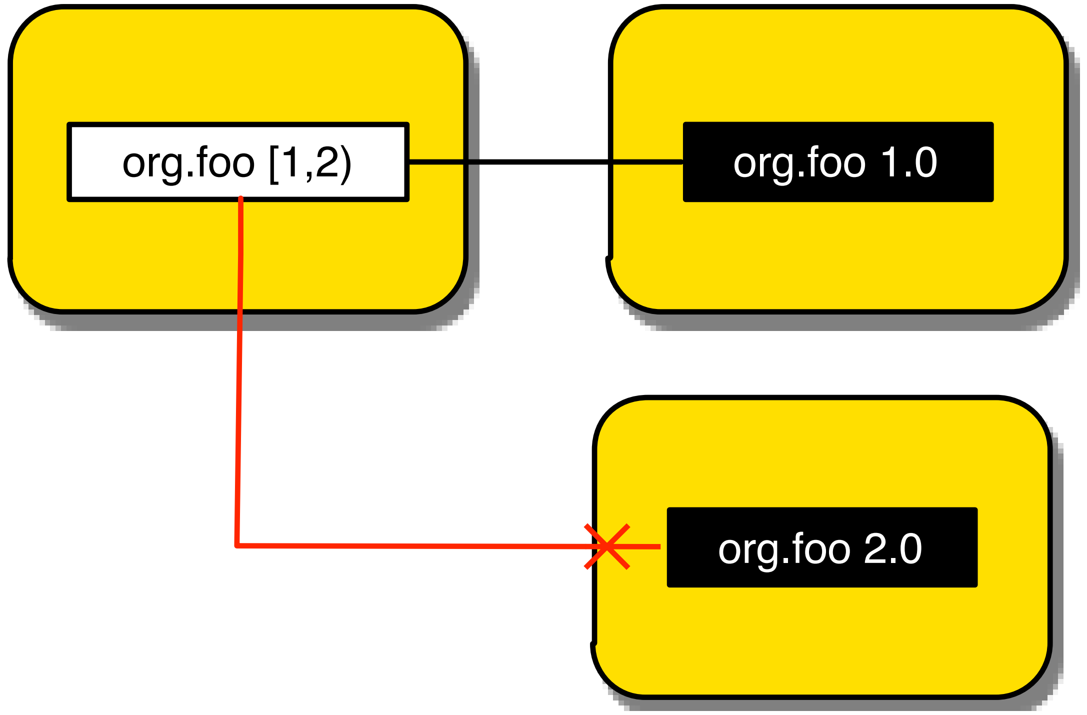
- JPMS only supports requires – equivalent to Require-Bundle
- Same problems!
- “requires” public means re-export

```
module B {
   requires org.example.api;
   requires public org.example.foo.api;
}
```

Versioning

Versioning

- OSGi supports versioning: of bundles and packages.
- Export packages with a version e.g. 1.0.1
- Import packages with a version e.g. [1, 2)
- Tooling detects how we use the package and generates the correct range.



Versioning

- JPMS **does not support versions** in module-info.java.
- A version can be supplied at build-time with a command-line param.
- … but it’s not used anywhere.
- Multiple versions of a module are **not supported**.
- Version selection by the module system is **out of scope**.

- Somebody (Maven?) has to create a set of modules that will work.
- That set must contain at most one version of each module.
  - Maven isn’t very good at this!

Dynamics

Dynamics

- OSGi bundles can be dynamically installed, updated, uninstalled.
- “But I’m an enterprise developer, I never want to do this”.
- Fair point! Nobody’s forcing you.

- OSGi originally designed for home gateways, similar to modern IoT.
- Installing and updating **minimal** set of dependencies is invaluable.
- Especially over low-speed, intermittent networks.

Dynamics

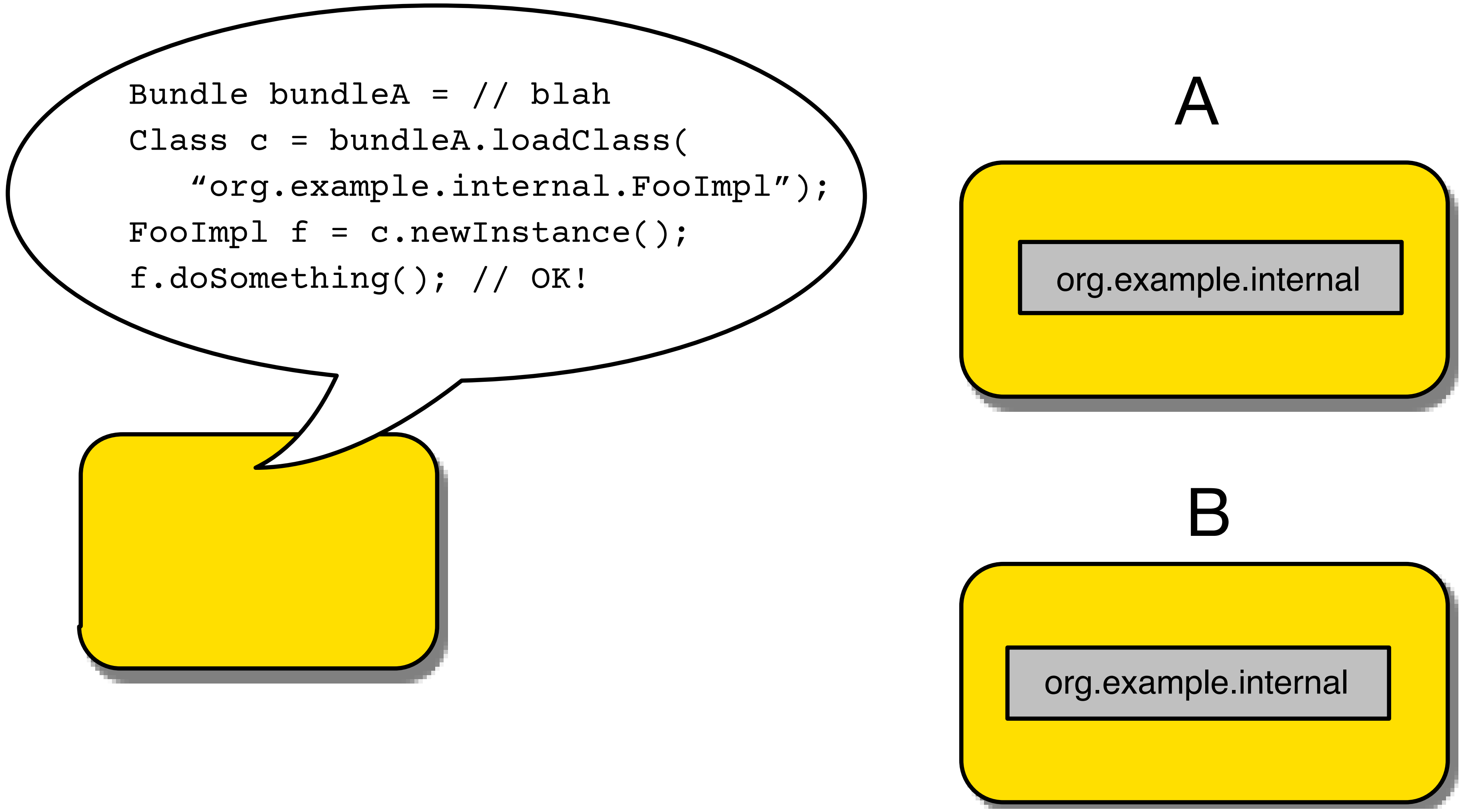
- So… dynamics are just for IoT?
- No! OSGi **Services** represent the change state of the world:
  - Remote Service availability (microservices!)
  - Network status
  - Market opening hours
  - …
- OSGi encourages a robust programming model that adapts to the real world

JPMS Dynamics

Reflection

Reflection

- This is the big one!
- First, OSGi:



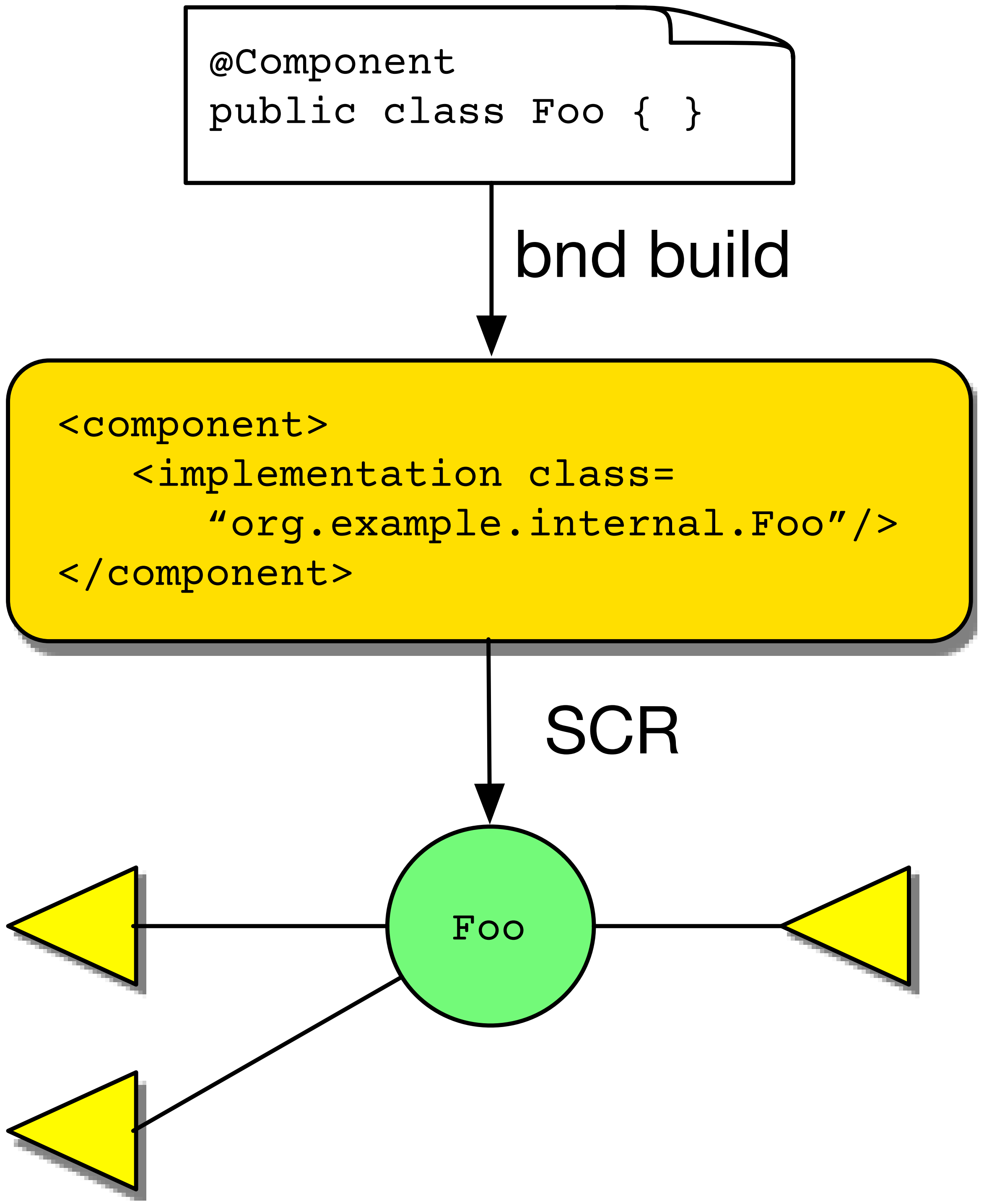
```
Bundle bundleA = // blah
Class c = bundleA.loadClass(
   “org.example.internal.FooImpl”);
FooImpl f = c.newInstance();
f.doSomething(); // OK!
```

Reflection

- Note:
  - No import of org.example.internal.
  - org.example.internal isn't exported.

- This is always possible if we know the origin bundle and type.
- Use case: Declarative Services

Declarative Services in a Nutshell



```
@Component
public class Foo { }
```

```
<component>
   <implementation class=
      “org.example.internal.Foo”/>
</component>
```

Declarative Services

- Doesn’t this break encapsulation? We can access any type!
- Debatable… but practically speaking, no.
- Origin bundle declares the type explicitly, otherwise it’s unknown.

- Reflection **could** be used to do bad things…
- … but you **can’t** claim you didn’t know what you were doing.

Other Use-Cases

- Dependency Injection frameworks (all of them).
- Object/Relational Mapping … Hibernate, JPA, etc.
- Serialization… JAXB, Protocol Buffers, etc.
- Eclipse Extension Registry (`plugin.xml`)

- In other words: most of our critical infrastructure!

The JPMS Approach

- JPMS automatically adds a “read edge” when we reflect on a module.
- But the private packages remain inaccessible, period.
- Probably the biggest area of contention in JSR 376.
- See http://bit.ly/jpms-reflect *

* http://openjdk.java.net/projects/jigsaw/spec/issues/#ReflectiveAccessToNonExportedTypes

Non-Solution: Services

- ServiceLoader has privileged access to named types inside module private packages.
- No help. Services in J2SE are very, very limited.
- No lifecycle control, no dependency injection…
- DI module has to declare “uses” for each interface type.

```
module A {
  provides org.example.api.Foo
     with org.example.internal.FooImpl;
}
```

* http://openjdk.java.net/projects/jigsaw/spec/issues/#ReflectiveAccessToNonExportedTypes

Possible Solution: Export and be Damned!

- We could just export all the packages!
- Inadvisable… now all internal packages are public API.
- Encapsulation is gone, both at build and run time.

```
module A {
  exports org.example.internal;
  // I hope nobody depends on this!
}
```

Possible Solution: Qualified Export

- Exports can be **qualified**: only accessible to specific, named modules.
- Problem: we need to know all the possible requirers in advance!
- Doesn’t work for specifications with multiple implementations, like JPA.

```
module A {
  exports org.example.internal to
         org.hibernate;
         // Bad luck EclipseLink!
}
```

Possible Solution: Dynamic Export

- Proposed and implemented by Oracle … then killed just last month.
- Idea: exports that are effective at runtime but not build time.
- Have to explicitly list every package to be treated this way.
- Weakens “fidelity across all phases” but this is closest to OSGi.

```
module A {
  exports dynamic org.example.internal;
  exports dynamic org.example.impl.a;
  exports dynamic org.example.impl.b;
  // Hope I got them all!
}
```

Possible Solution: Weak Modules

- Oracle’s ~~current~~ proposal (killed last week!)
- Idea: weak modules have no private packages, everything is exported.
- Envisioned as a transitional step to “strong” modules.
- Question: if we use DI, ORM, serialization etc, can we **ever** get rid of “weak” modules??

```
weak module A {
  // The jokes write themselves…
}
```

Possible Solution: Open Modules

- Oracle’s even more current proposal (since Thursday, around tea-time).
- Like “dynamic” exports, i.e. open for reflection but not at compile-time.
- In an open module, all packages are open.
- Normal modules can open specific packages.

```
open module A {
  // Can still explicitly export…
  exports org.example.api;
}
module B {
  opens org.example.impl;
}
```

Possible Solution: Privileged Modules

- Community proposal from Nikita Lipsky.
- Idea: bless certain modules as “privileged”. They, and **only** they, can access private packages of any module.
- Perhaps a command line switch to permit privileged modules?
- Attraction: only a small number of modules ever need this.
- Why should only ServiceLoader be allowed to do this?

Interoperability,

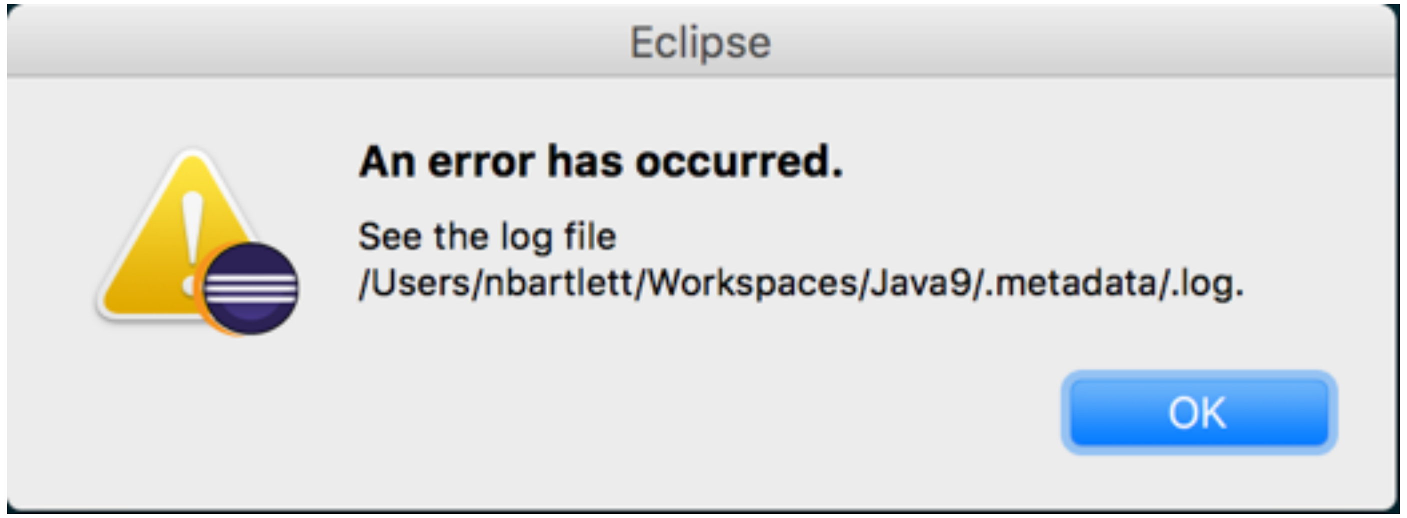
OR,

Can Dogs and Cats Live Together??

YES

Basic Interop

- Java 9 is backwards compatible, if you use only standard Java SE APIs.
- OSGi uses only standard Java SE APIs.
- Therefore OSGi runs unchanged on Java 9!
- Give or take the usual bugs, it's only Early Access:



An error has occurred.

See the log file
/Users/nbartlett/Workspaces/Java9/.metadata/.log.

Compatibility Issues

- Java 9 may **not** be backwards compatible for code that uses non-standard APIs.
- E.g.: `sun.misc.Unsafe`
- Same advice for OSGi developers as all other Java developers.
- As an OSGi dev you have a much better idea of your dependencies already!

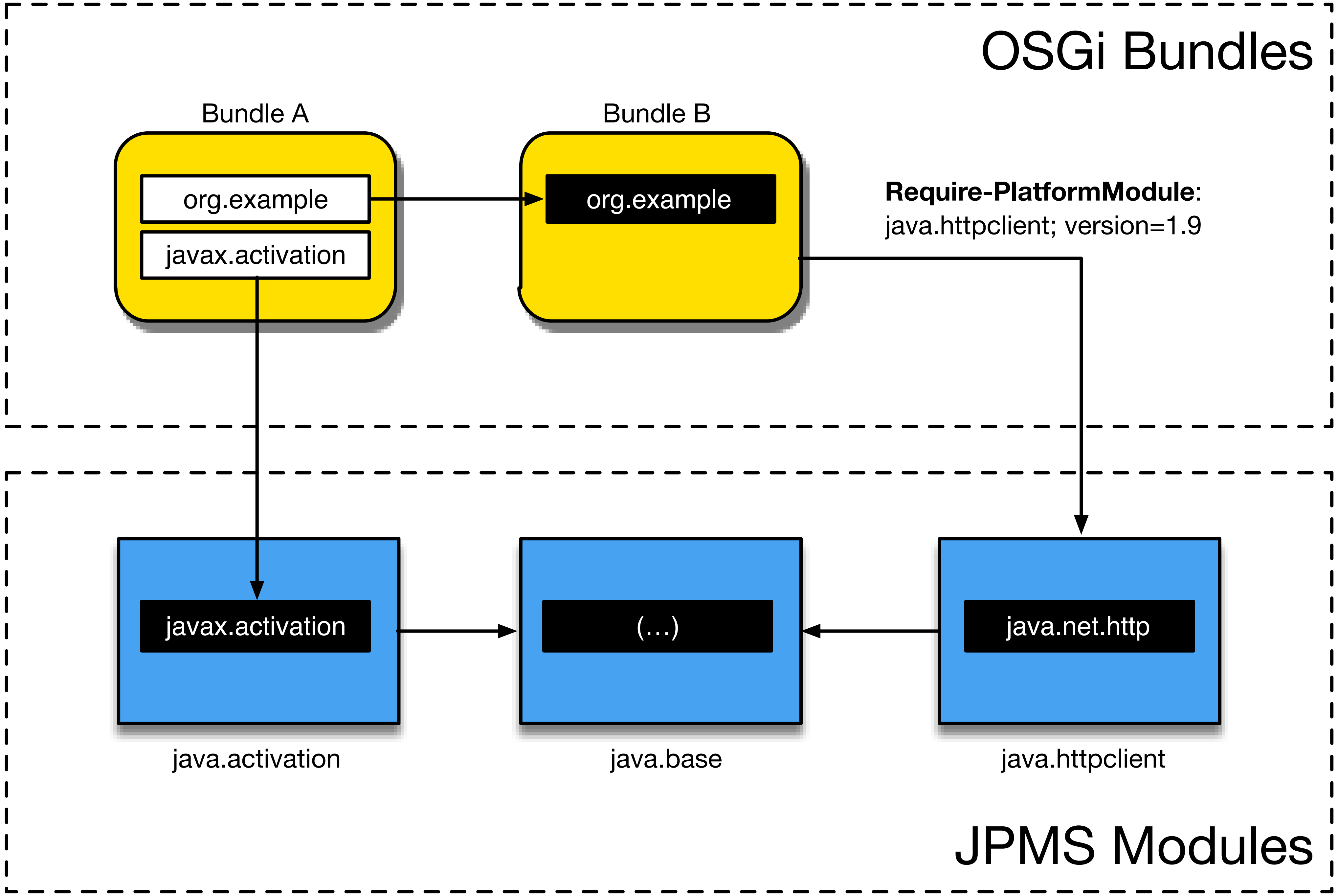
Can We Do Better?

- YES!
- Background: “platform” dependencies are handled 2 ways in OSGi:
- **1.** The *Execution Environment*: a capability published by the Framework.
  - Example: `Require-Capability: osgi.ee; filter:=“(&(osgi.ee=JavaSE)(version=1.8))"`
  - Generated by tooling (fortunately!)
  - Bundle now **only** resolves on Java 8, can access APIs e.g. `java.util.function`.
- **2.** Import-Package for everything outside `java.*`.

Can We Do Better?

- Import-Package still works great.
- The *execution environment* concept is probably obsolete.
- Can no longer talk about monolithic platform with a single version.
- Need to depend on platform modules… i.e. JPMS.
- Proposal: Require-PlatformModule
- Example: `Require-PlatformModule: java.httpclient; version=1.9`
- Bundle should no longer resolve on a platform that lacks the `java.httpclient` module.

Can We Do Better?



Proof of Concept

- Iterate modules in current platform.
- Provide a capability for each module from the Framework.
- Export packages into OSGi for each module-exported package.
- Launch OSGi bundles in the “unnamed” JPMS module.
- Simulate Require-PlatformModule with a capability:
    - `Require-Capability: jmodule; filter:=“(jmodule=java.httpclient)”`
- Works! `github.com/njbartlett/osgi_jigsaw`

DEMO

Next Steps

- Use `jdeps` to calculate minimal platform dependencies for a set of bundles.
- Use `jlink` to create a complete runtime: JVM, JPMS modules, OSGi bundles.
- Integrate these tools into bnd and Bndtools.

Can We Do Even Better?

- That was **unidirectional** dependence.
- All OSGi bundles in a single JPMS module.
- OSGi can’t use JPMS encapsulation (assuming we want to?).
- Can we map bundles directly to modules? One-to-one?
- **Maybe**… but it’s complicated.
- JPMS not dynamic, no overlapping private packages, no cycles.
- Would require multiple module *Layers*, not strictly hierarchical, with Layer creator controlling module wiring.
- Requires changes in JPMS/Jigsaw that may or may not happen.
- Tom Watson (Equinox project lead) has done great work here.

JSR Membership

- I am now a member of the JSR 376 Expert Group.

- I want to represent the OSGi community…
  - (within the constraints set by the Spec Lead).

- Talk to me if there's something you think I should raise on the EG.

Conclusion

JPMS Biggest Problem

- “Adding” modularity to a 20-year-old product is **hard**.
- Best way to modularise? **Refactor!**
- **Not an option for the JRE.**

- JPMS did what was necessary to modularise the JRE without refactoring.
- OSGi **couldn’t** have done this job.
  - (It was tried – Apache Harmony came close)

- The choices made by JPMS have unfortunate consequences.
- Why suffer those consequences outside the JRE?

Takeway Message:

JPMS for the JDK.

OSGi for Everything Else.

Afterword

What does JPMS remind me of…?

BReXiT

JPMS ~= BREXIT

- I didn’t vote for it!
- Nobody knows what it will look like.
- Huge distraction from actual important stuff.
- Can’t be stopped now …or can it? ;-)
- Try to make the best of it?

Paremus

www.paremus.com @Paremus info@paremus.com