

Project Jigsaw: Under The Hood

Alex Buckley

Java Platform Group, Oracle

October 2015



The Modularity Landscape

- Java Platform Module System
 - JSR 376, started February 2015 – targeted for Java SE 9
- Java SE 9 Platform
 - JSR not started yet – will own the modularization of the Java SE API
- Project Jigsaw
 - Reference Implementation of JSR 376 in OpenJDK (JEP 261)
 - Modularization of the JDK (JEP 200, JEP 201, JEP 260)
 - New run-time image format (JEP 220)

Project Jigsaw: Under The Hood

Part I: Accessibility and Readability

Part II: Different Kinds of Modules

Part III: Loaders and Layers

Part I: Accessibility and Readability

Accessibility 1995-2015

- public
- protected
- <package>
- private

Accessibility 2015-

- *public to everyone*
- *public but only to specific modules*
- *public only within a module*
- protected
- <package>
- private

‘public’ no longer means “accessible”.

The result:

[GLASSFISH-21428] JDK9 - ...

https://java.net/jira/browse/GLASSFISH-21428

JIRA Dashboards Projects Issues Agile Quick Search Log In

glassfish / GLASSFISH-21428

JDK9 - REFERENCES TO JDK INTERNAL API IN main/appserver/security/core-ee/src/main/java/com/sun/enterprise/security/provider/PolicyWrapper.java

Agile Board Export

Details

Type:	Bug	Status:	OPEN
Priority:	Major	Resolution:	Unresolved
Affects Version/s:	4.1	Fix Version/s:	None
Component/s:	security		
Labels:	jdk9-int		
Tags:	jdk9-int		

People

Assignee: Arindam Bandyopadhyay

Reporter: Arindam Bandyopadhyay

Votes: 0 Vote for this issue

Watchers: 1 Start watching this issue

Dates

Created: 29/Sep/15 6:26 AM

Updated: Today 10:58 AM

Description

There is a reference to jdk internal api in main/appserver/security/coreee/src/main/java/com/sun/enterprise/security/provider/PolicyWrapper.java. We are getting the following exception in time of server start up with jdk9-jigsaw build.

ava.lang.IllegalAccessException: class com.sun.enterprise.security.provider.PolicyWrapper (in module: Unnamed Module) cannot access class sun.security.provider.PolicyFile (in module: java.base), sun.security.provider is not exported to Unnamed Module

at com.sun.enterprise.security.provider.PolicyWrapper.getNewPolicy(PolicyWrapper.java:75)

at com.sun.enterprise.security.provider.BasePolicyWrapper.<init>(BasePolicyWrapper.java:148)

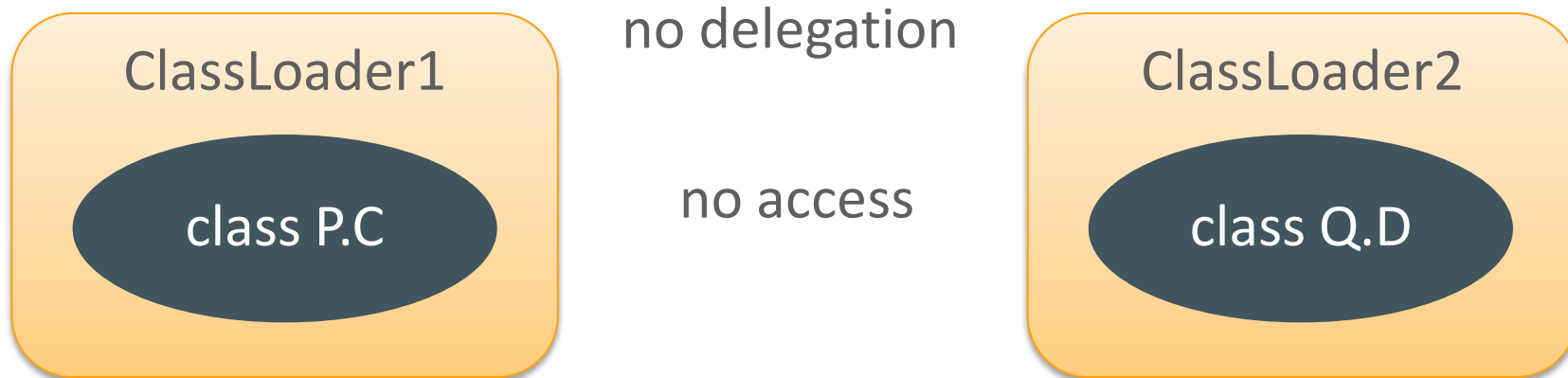
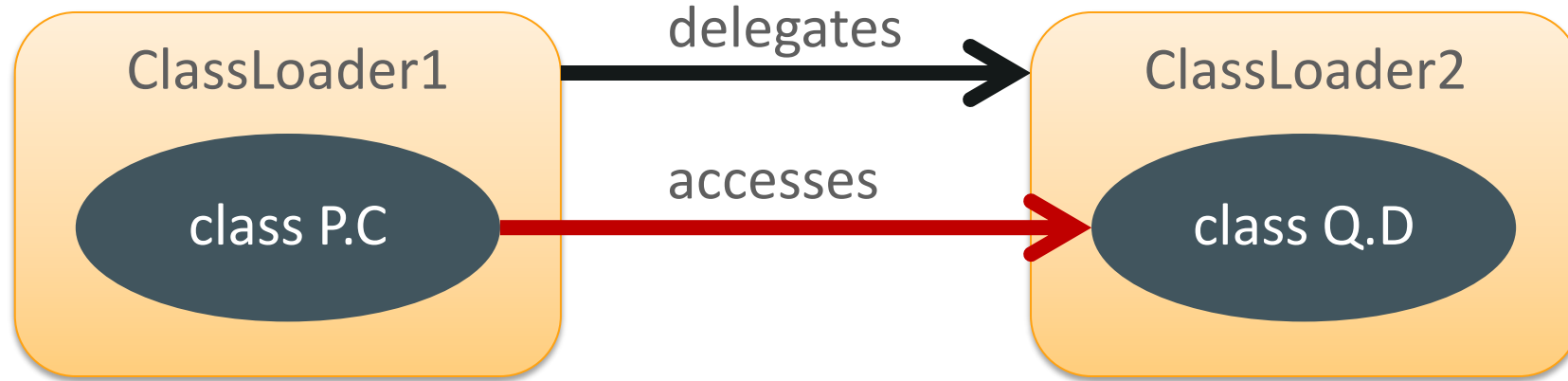
at com.sun.enterprise.security.provider.PolicyWrapper.<init>(PolicyWrapper.java:67)

at sun.reflect.NativeConstructorAccessorImpl.newInstance0(java.base@9.0/Native Method)

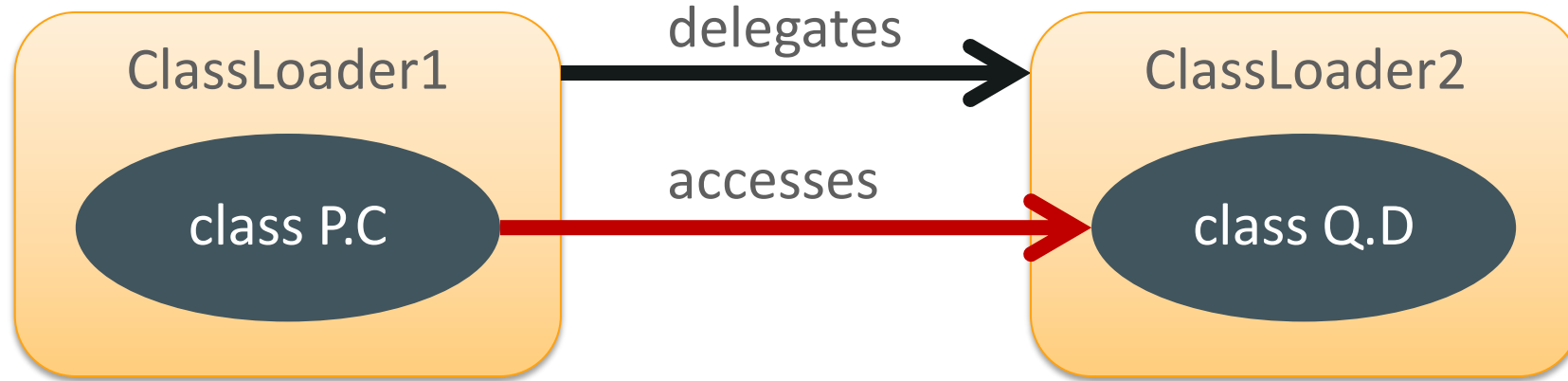
Accessibility and Module Declarations

```
// src/java.sql/module-info.java
module java.sql {
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
}
```

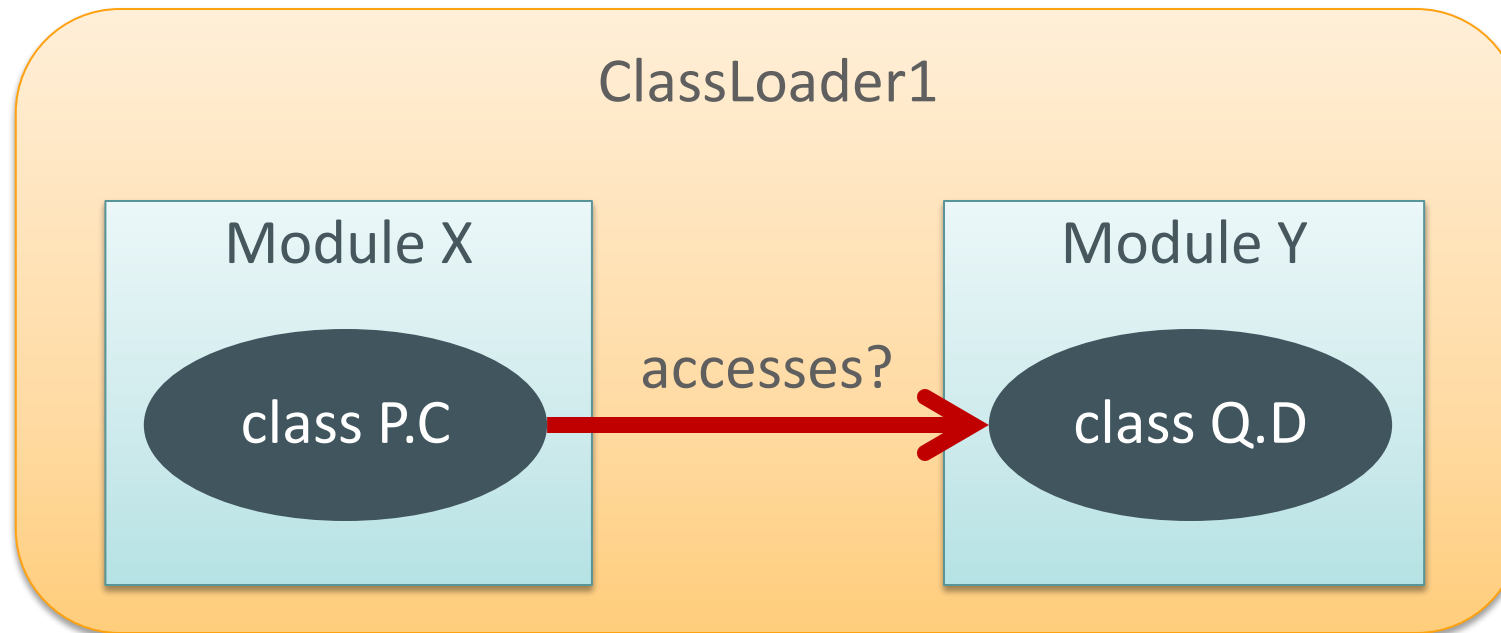
Accessibility and Class Loaders



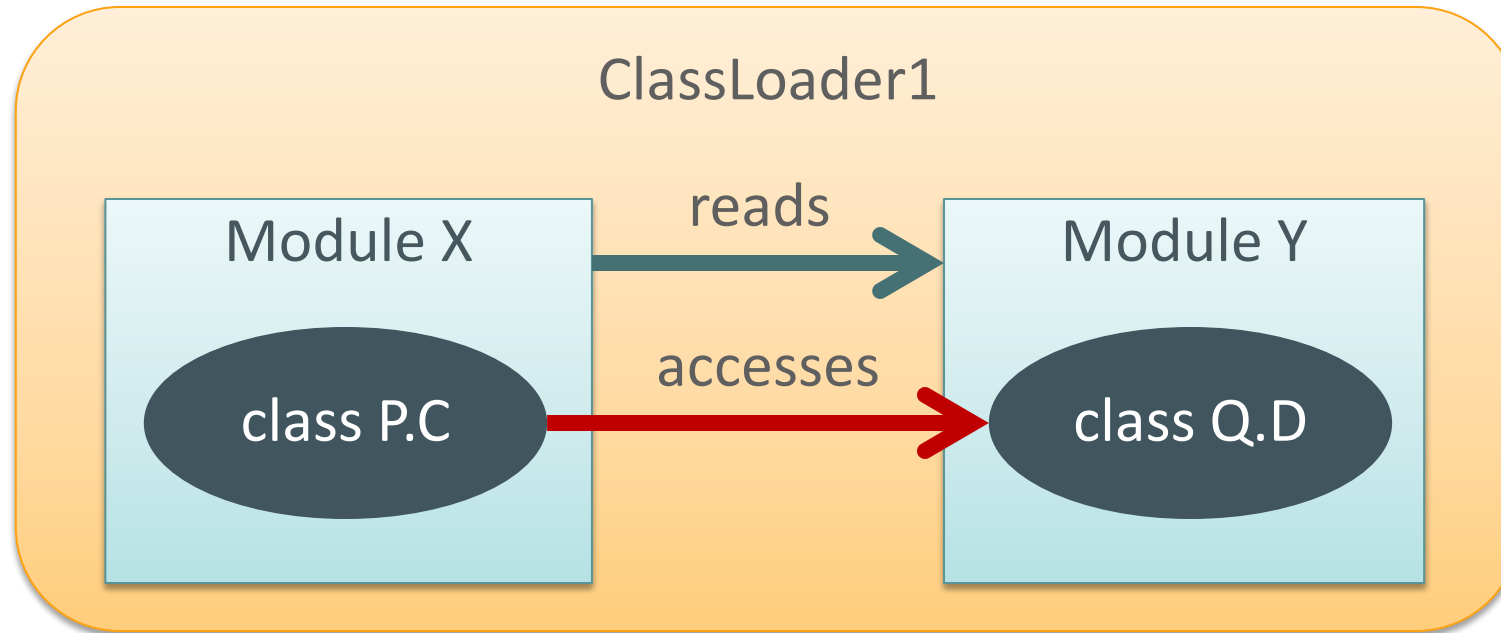
Accessibility and Class Loaders



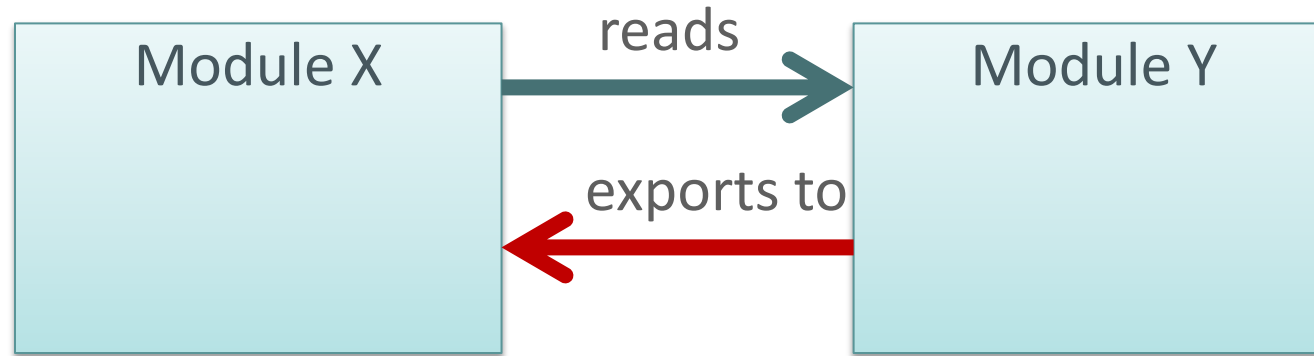
One Class Loader, Many Modules



The Role of Readability



The Role of Readability



```
module X {  
    requires Y;  
}
```

```
module Y {  
    exports Q;  
}
```

Readability in the Java SE module graph

```
module java.sql {  
    requires java.logging;  
    exports java.sql;  
}
```

```
package java.sql;  
import java.util.logging.Logger;  
public class DriverManager {  
    new Logger() {..}  
}
```

```
module java.logging {  
    exports java.util.logging;  
}
```

```
package java.util.logging;  
public class Logger {  
    ...  
}
```

Readability in the Java SE module graph

```
module java.sql {  
    requires java.logging;  
    exports java.sql;  
}
```

```
package java.sql;  
import java.util.logging.Logger;  
public interface Driver {  
    Logger getParentLogger();  
}
```

```
module java.logging {  
    exports java.util.logging;  
}
```

```
package java.util.logging;  
public class Logger {  
    ...  
}
```


Readability in the Java SE module graph

```
module myApp {  
    requires java.sql;  
    requires java.logging; ☹  
}
```

```
module java.sql {  
    requires java.logging;  
    exports java.sql;  
}
```

```
module java.logging {  
    exports java.util.logging;  
}
```

Readability in the Java SE module graph

```
module myApp {  
    requires java.sql;  
requires java.logging; ☺  
}
```

```
module java.sql {  
    requires public java.logging;  
    exports java.sql;  
}
```

```
module java.logging {  
    exports java.util.logging;  
}
```

Readability in the Java SE module graph

```
module myApp {  
    requires java.sql;  
}
```

```
module java.sql {  
    requires public java.logging;  
    requires public java.sql.time;  
}
```

```
module java.logging {  
    exports java.util.logging;  
}  
  
module java.sql.time {  
    exports java.sql.time;  
}
```

Direct and implied readability

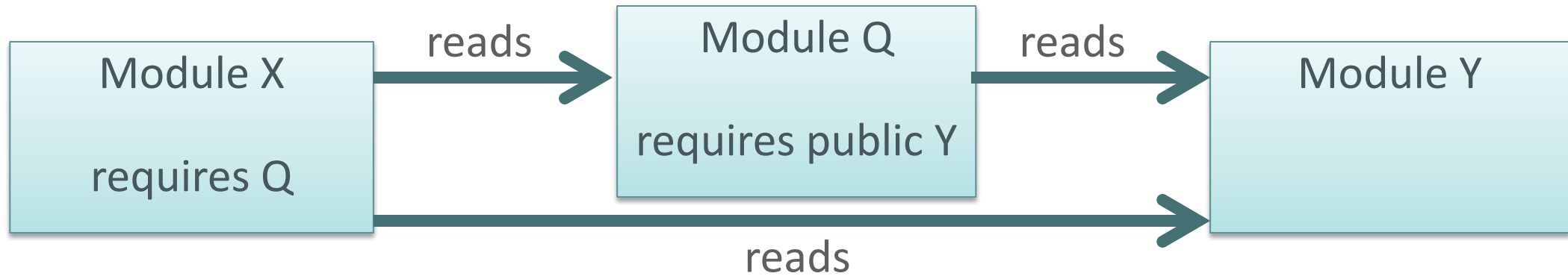
- X reads Y if:

- X requires Y

or



- X reads Q, and Q requires public Y



Core Reflection

```
void doSomething(Class<?> c) {  
    Method[] ms = c.getDeclaredMethods();  
    ms[0].invoke(...);  
}
```

Core Reflection

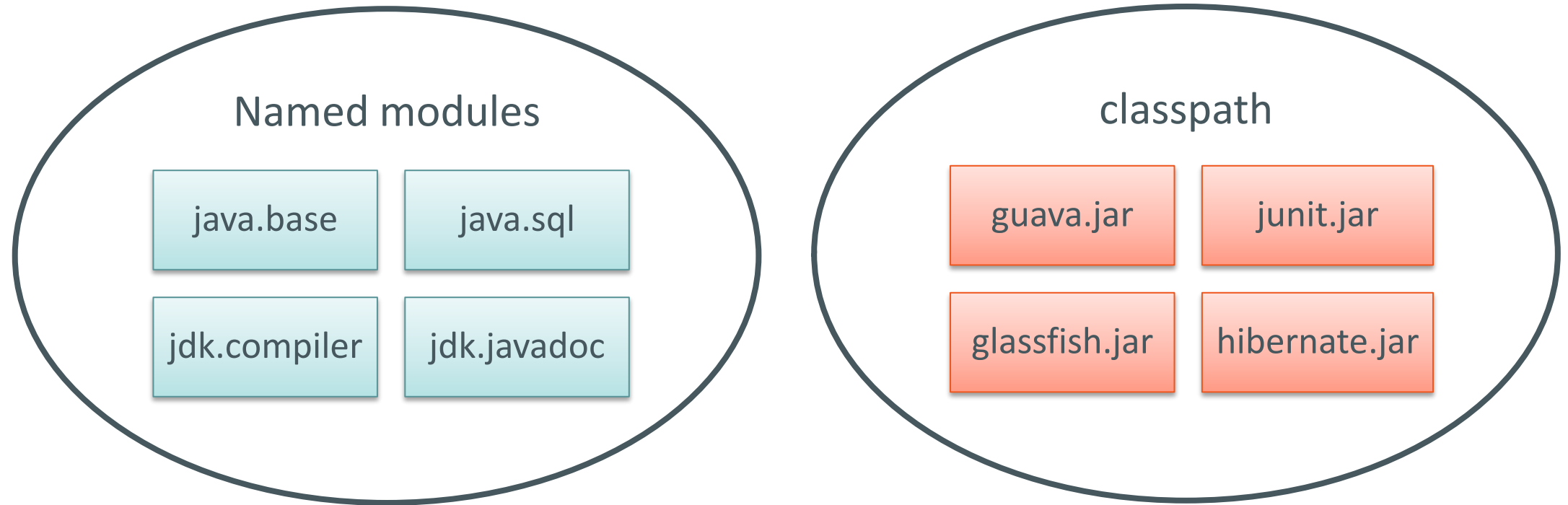
`setAccessible(true)`

Summary of Part I: Accessibility and Readability

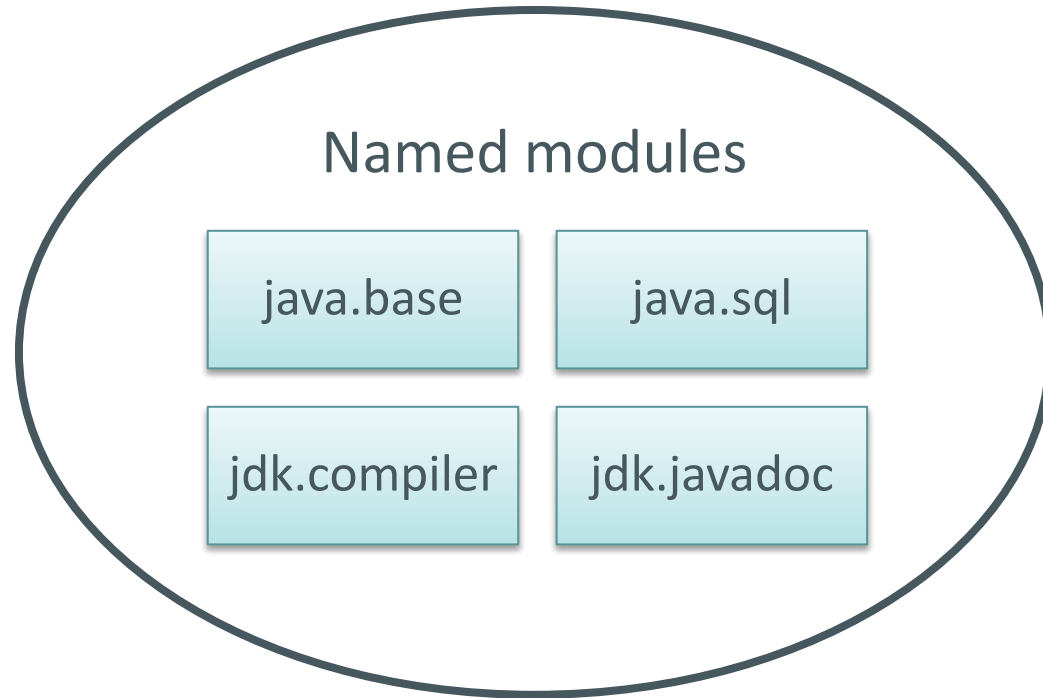
- Accessibility used to be a simple check for ‘public’ or “same package”.
- In Java SE 9, accessibility strongly encapsulates module internals.
- Accessibility relies on readability, which can be direct or implied.
- **Accessibility is enforced by the compiler, VM, and Core Reflection.**

Part II: Different Kinds of Modules

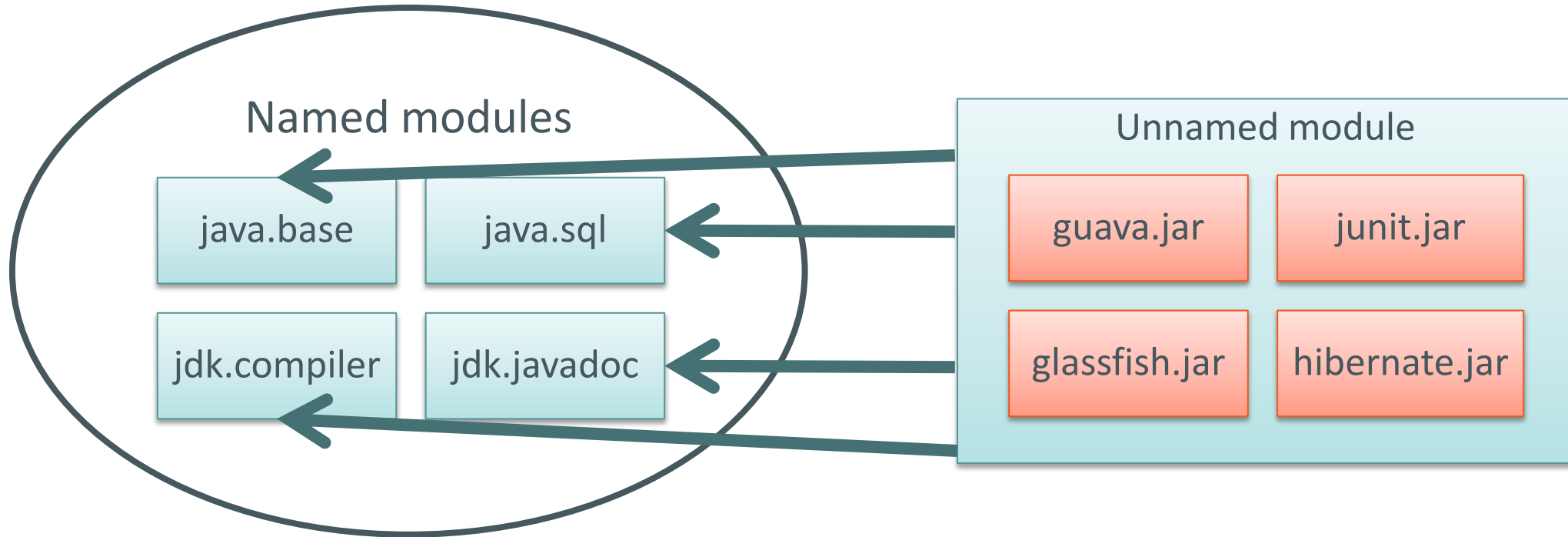
Named Modules



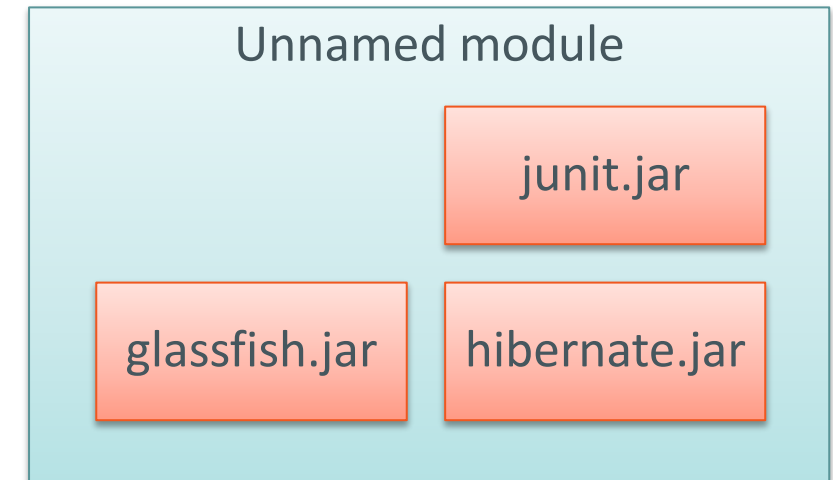
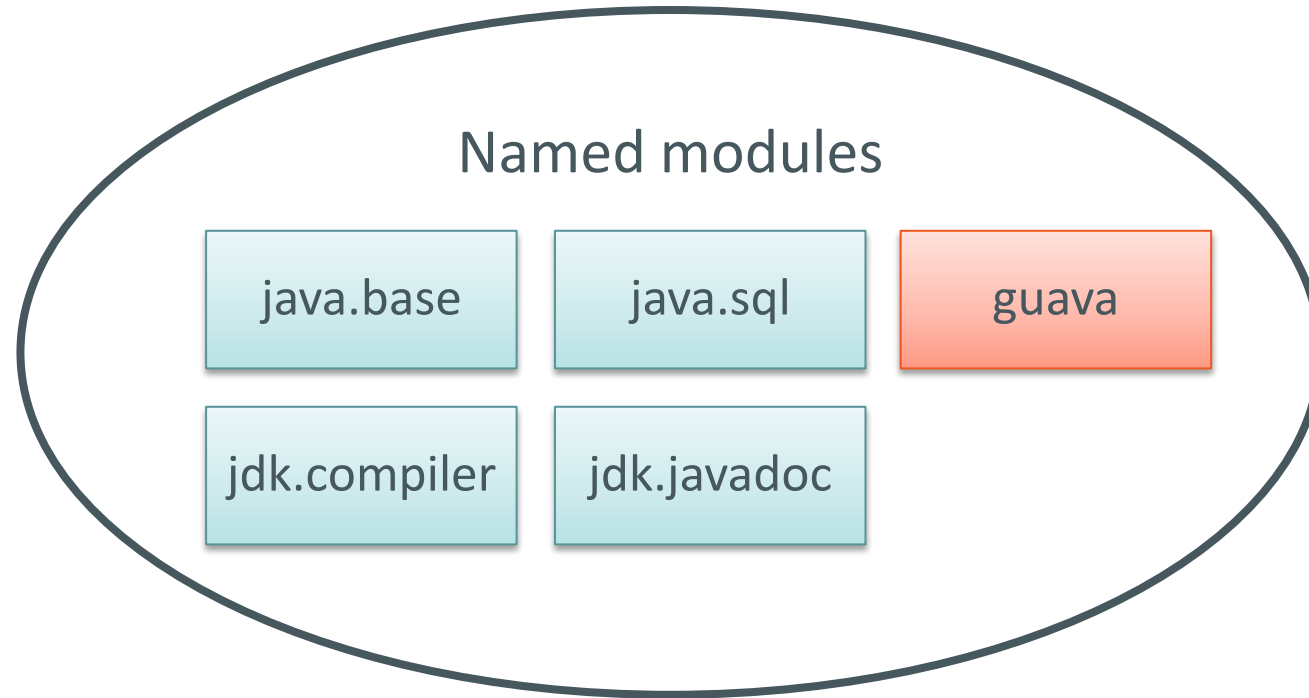
The Unnamed Module



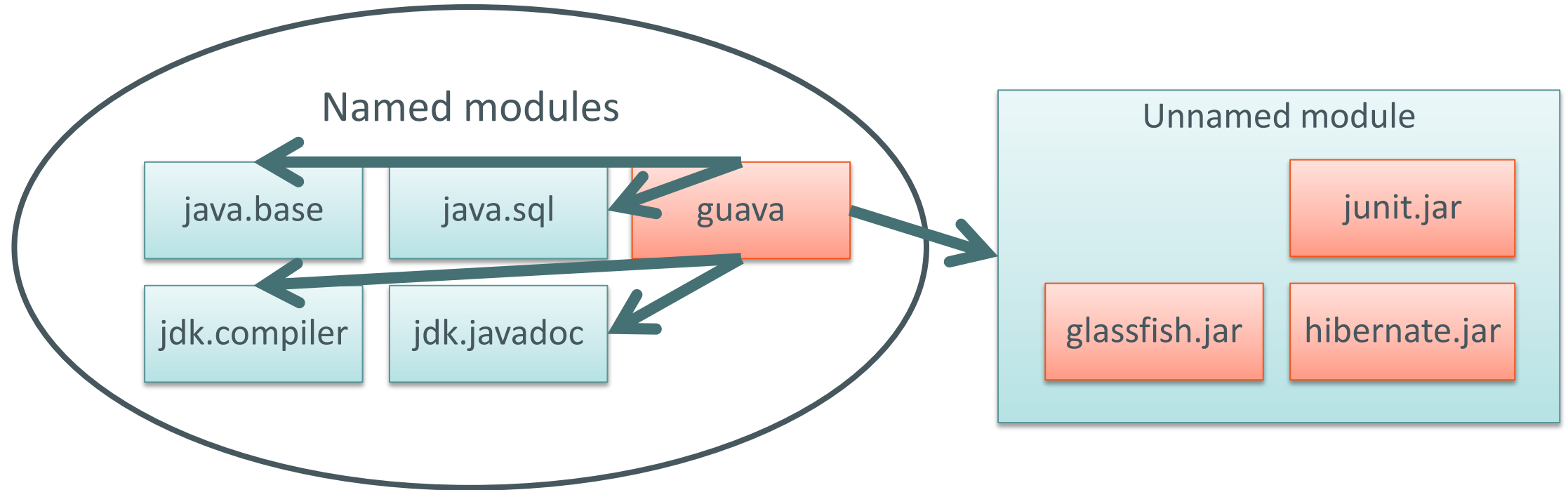
The Unnamed Module



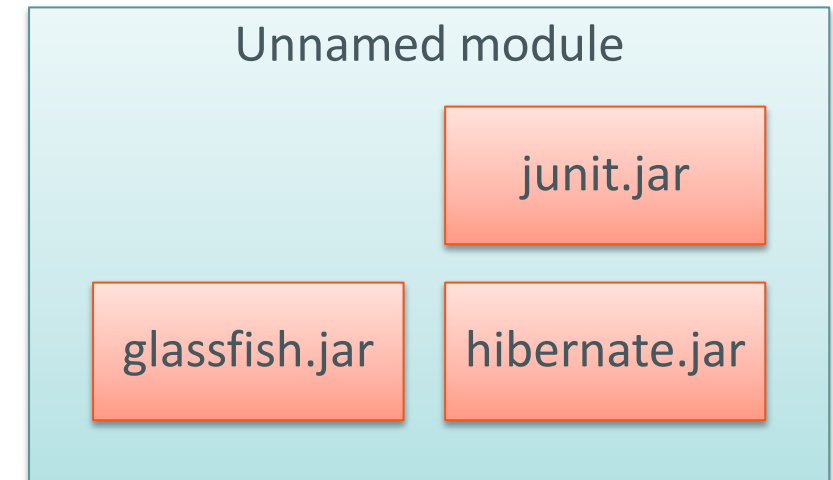
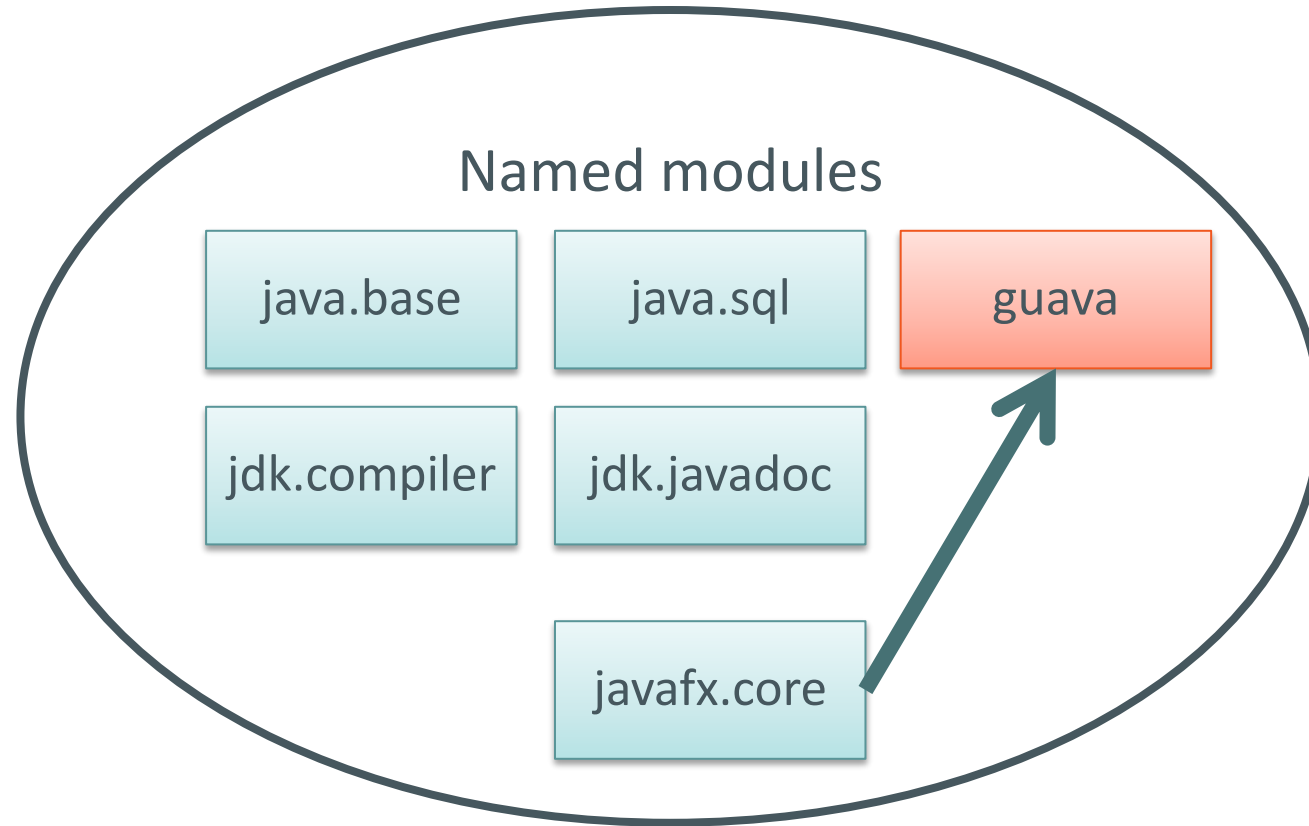
Automatic Modules



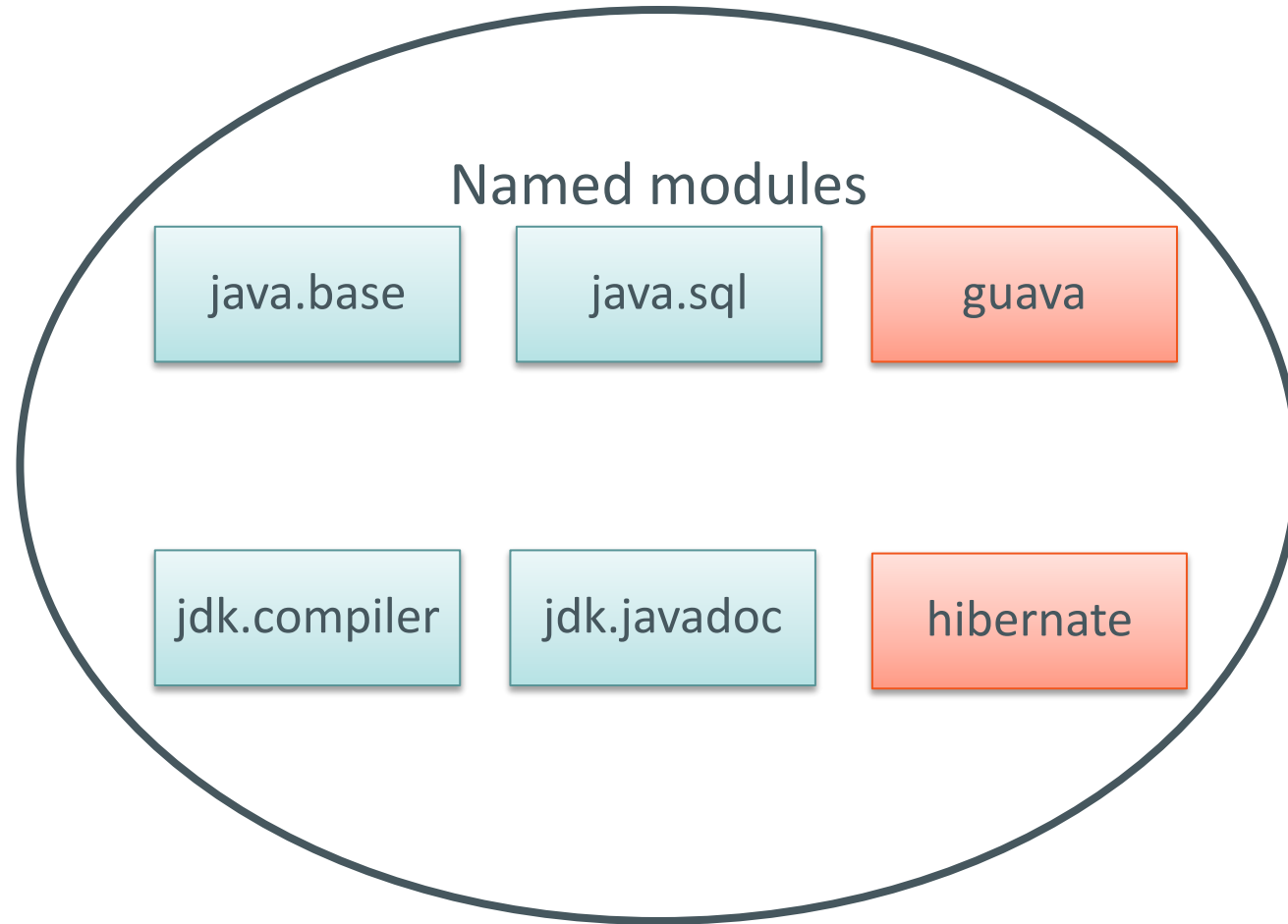
Automatic Modules



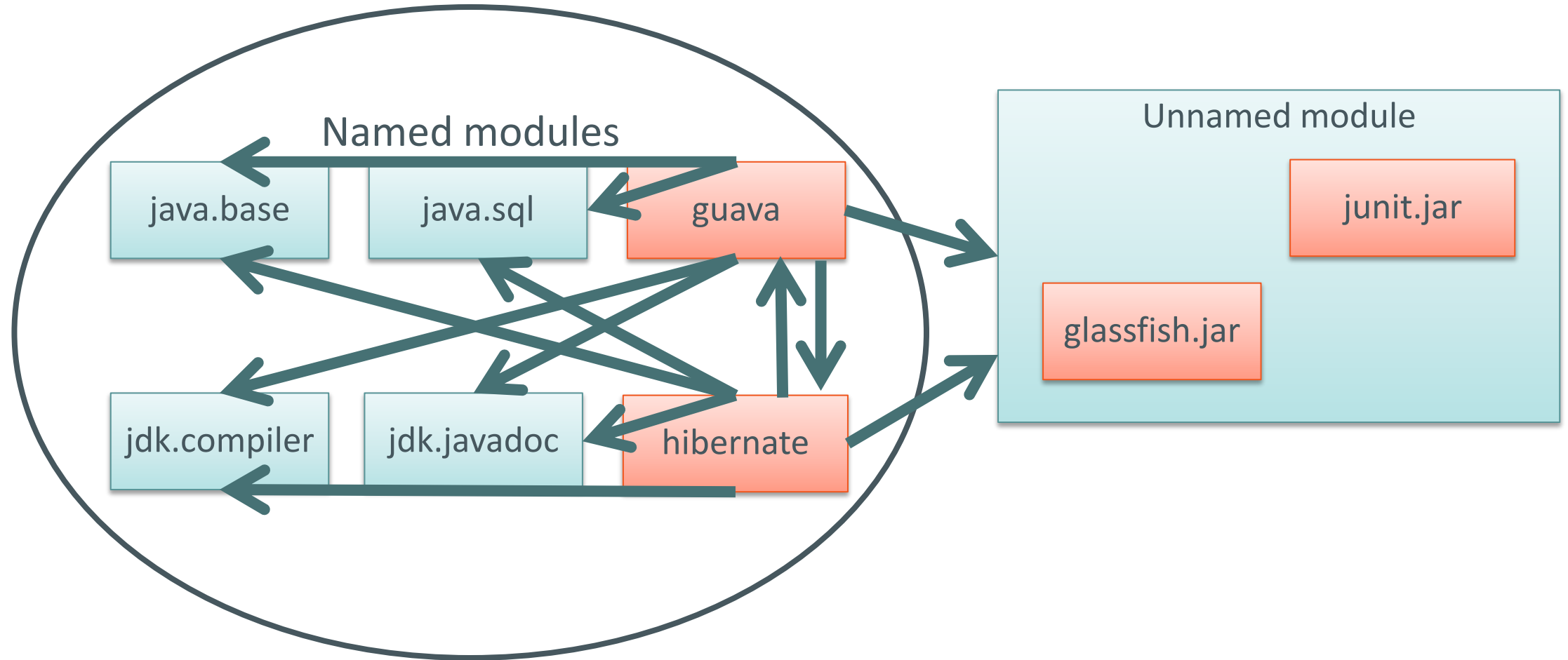
Automatic Modules



Multiple Automatic Modules



Multiple Automatic Modules



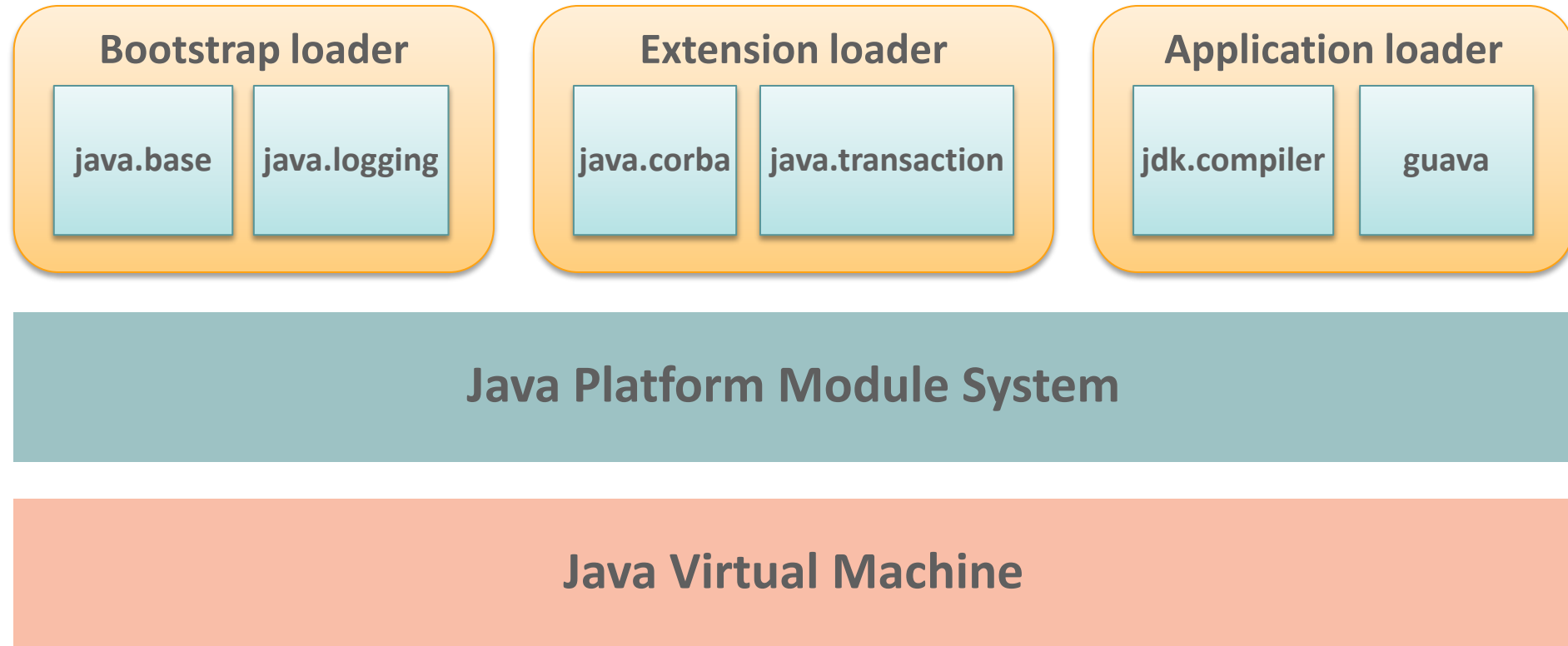
Summary of Part II: Different Kinds of Modules

- Explicit named modules (java.sql)
- Automatic named modules (guava)
- Unnamed module (a.k.a. classpath)
- **Lots of readability “for free” to help with migration.**

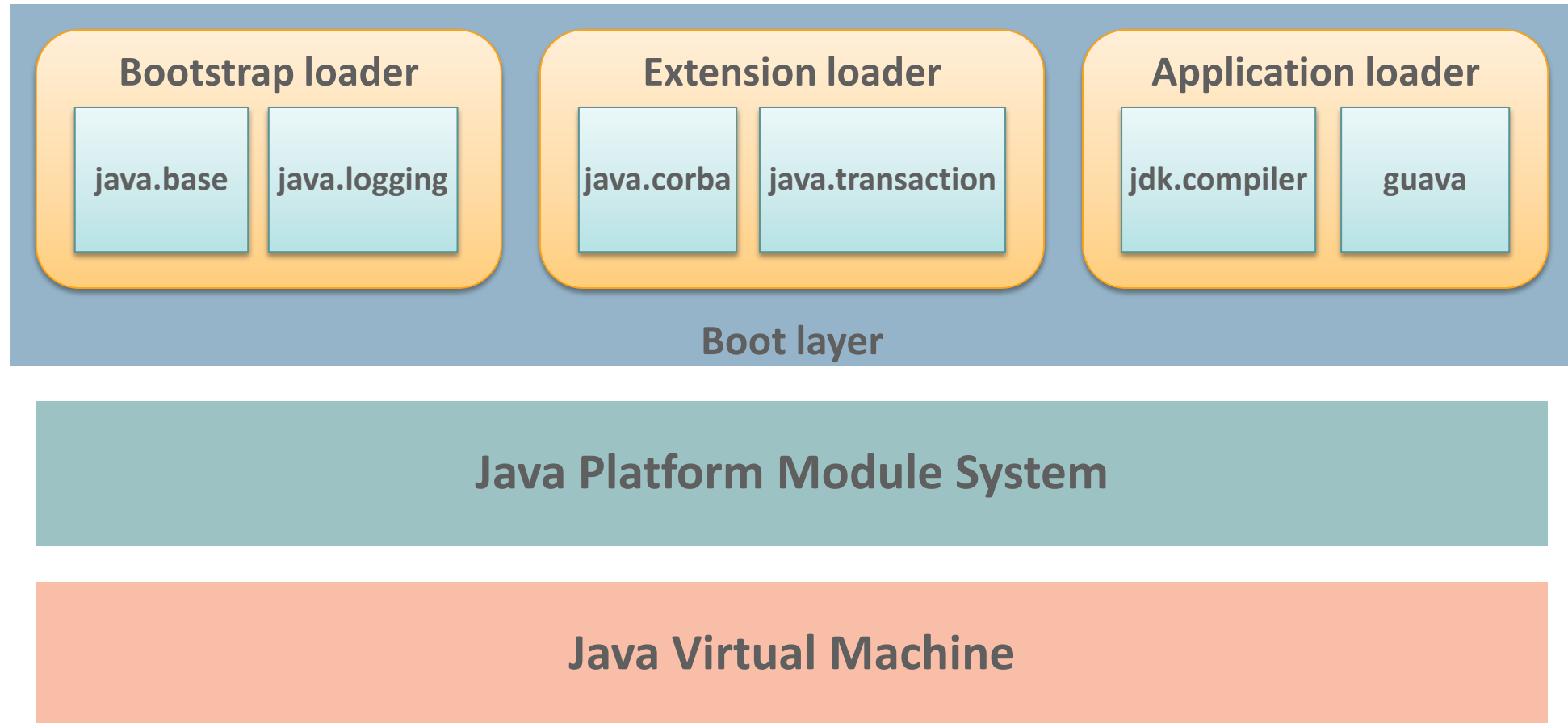
Part III: Loaders and Layers

Class loading doesn't change.

Class Loaders in the JDK

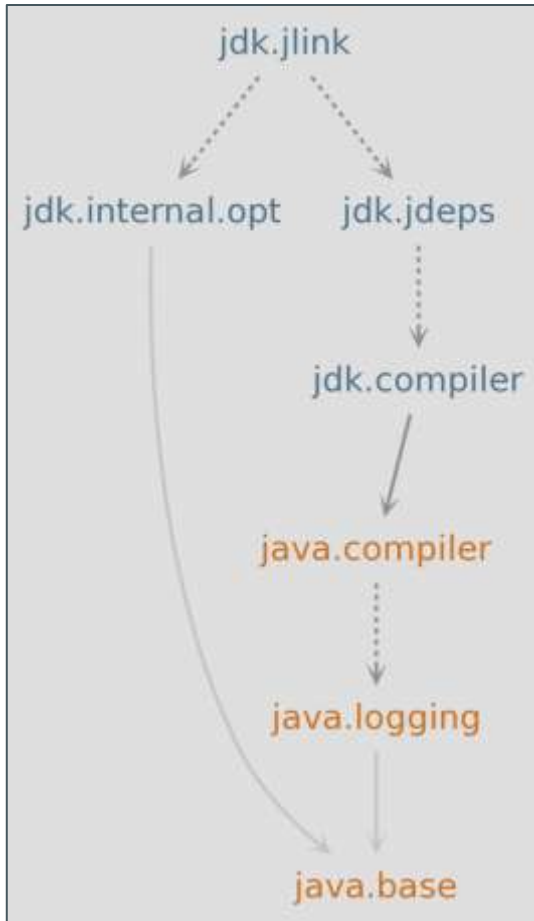


Layers



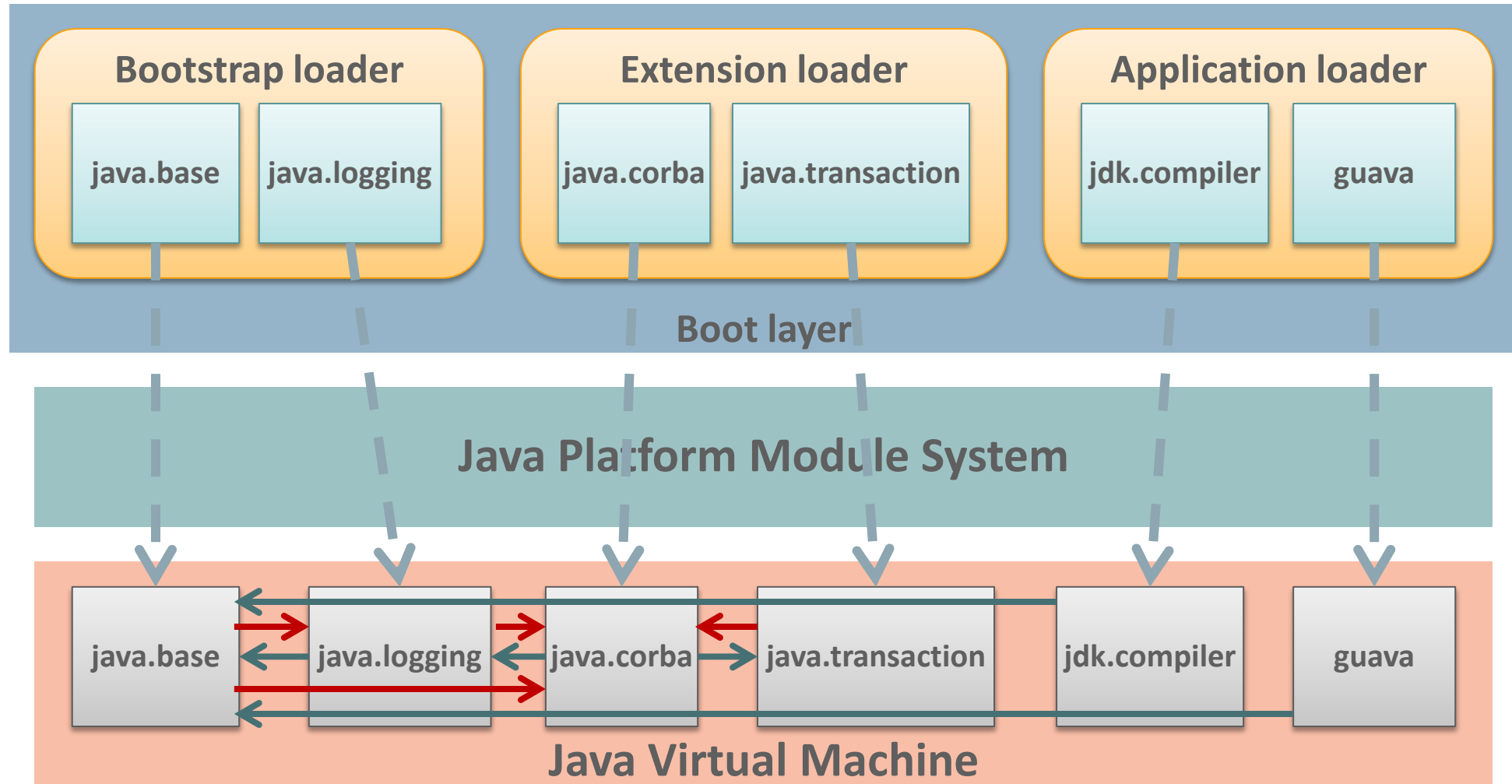
Layer creation

(1)

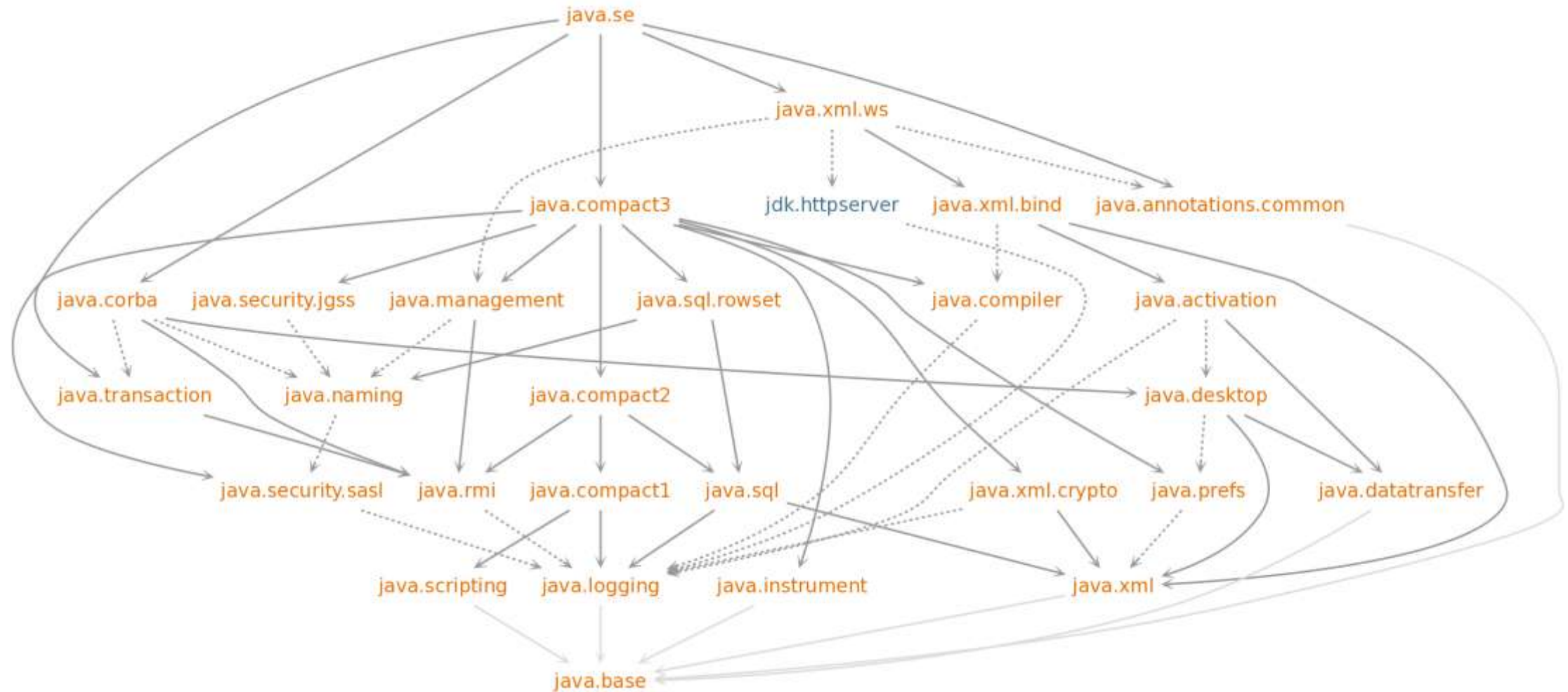


```
(2) String moduleName -> {  
    switch (moduleName) {  
        case "java.base":  
        case "java.logging":  
            return BOOTSTRAP_LDR;  
        default:  
            return APP_LDR;  
    }  
}
```

Layers and the VM

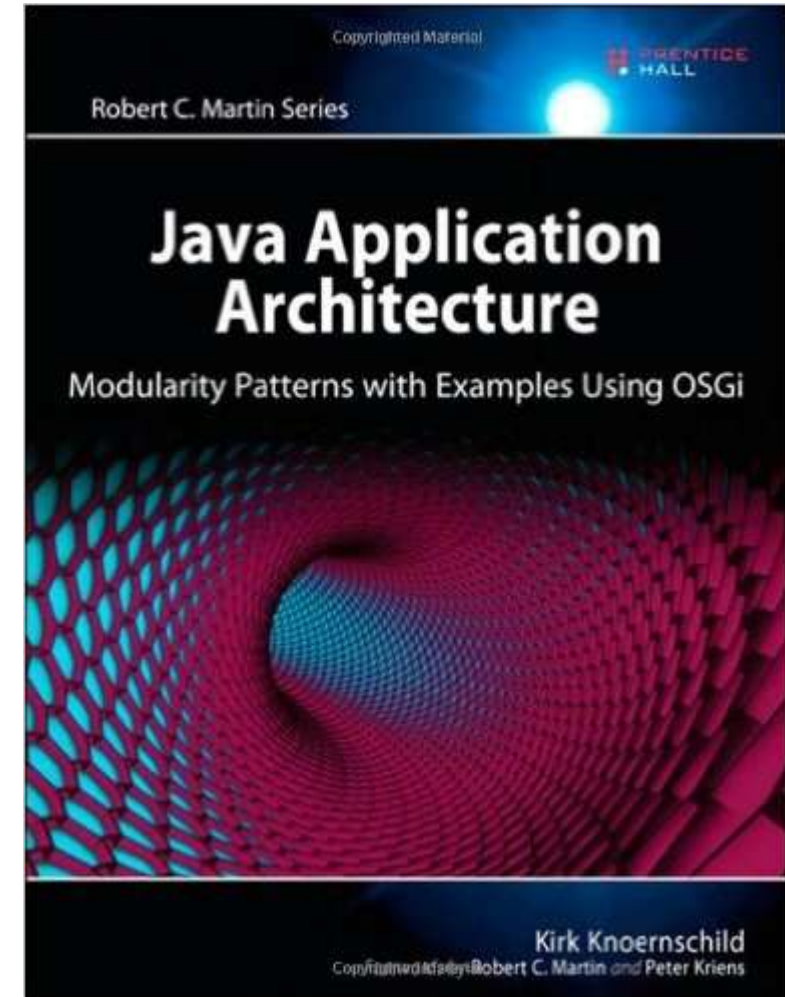


Well-formed graphs



Well-formed graphs

“Excessive dependencies are bad. But, cyclic dependencies are especially bad.”



Well-formed graphs

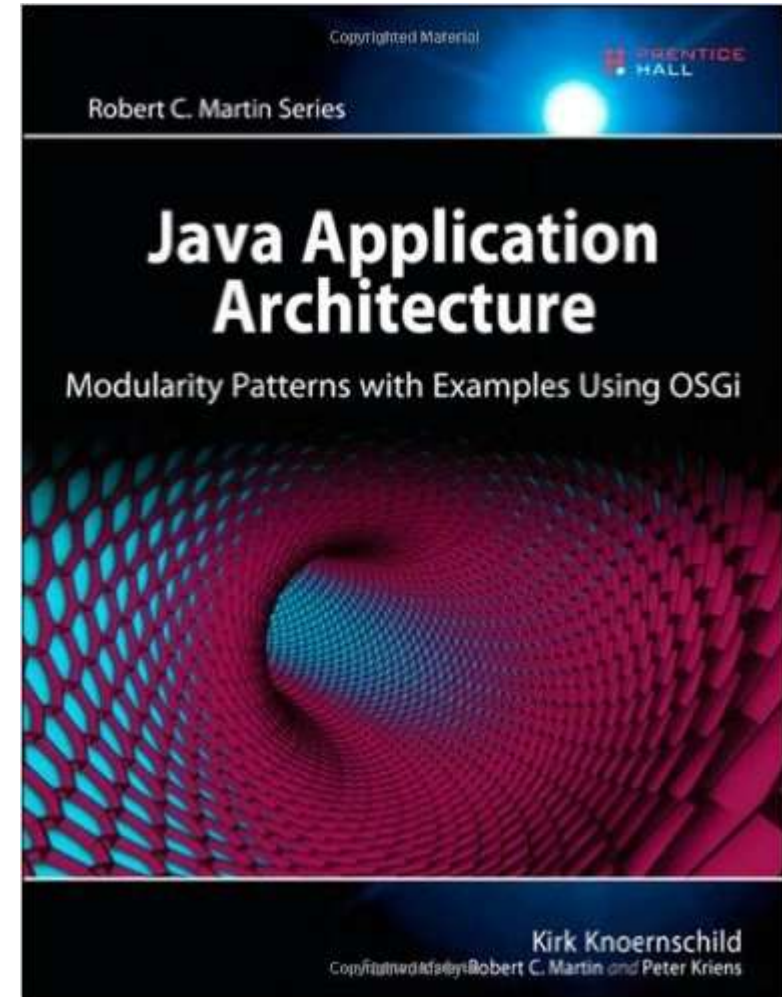
“Generally speaking, cycles are always bad!

However, some cycles are worse than others.

Cycles among classes are tolerable, assuming they don't cause cycles among the packages or modules containing them.

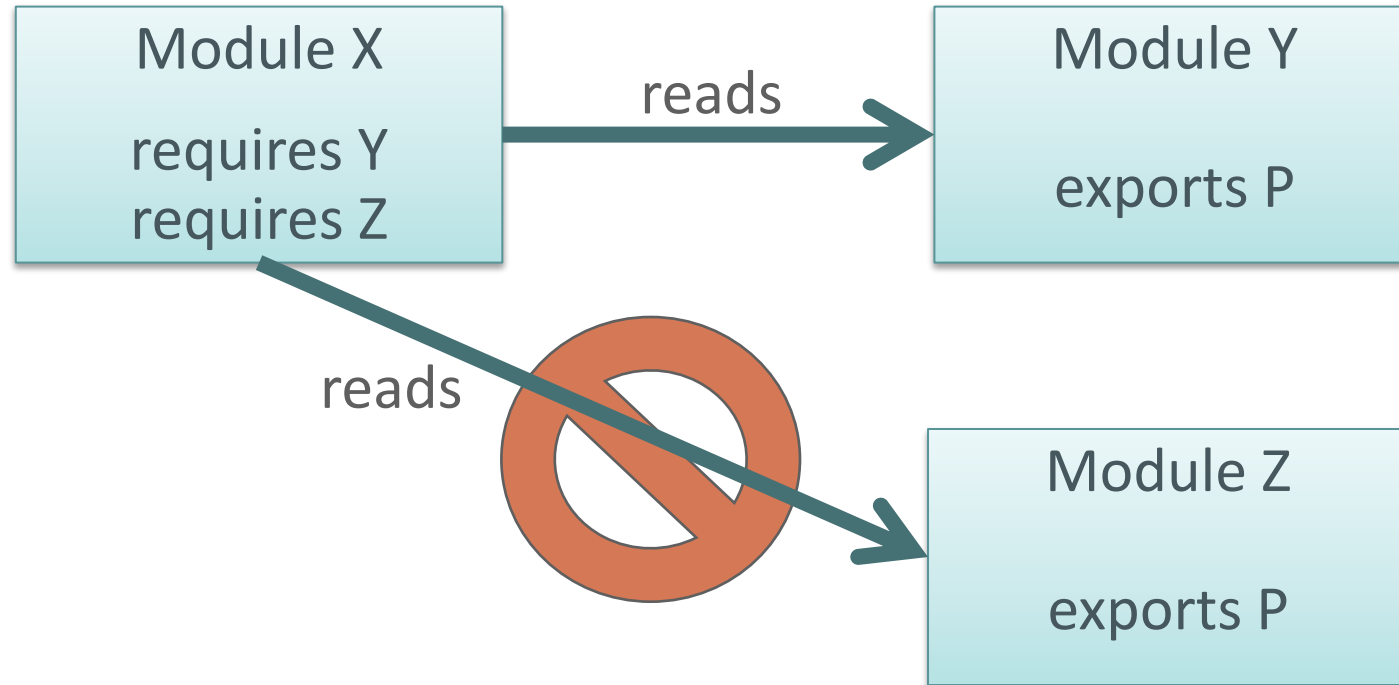
Cycles among packages may also be tolerable, assuming they don't cause cycles among the modules containing them.

Module relationships must never be cyclic.”



Well-formed graphs

- A module may read at most one module that exports a package called P.

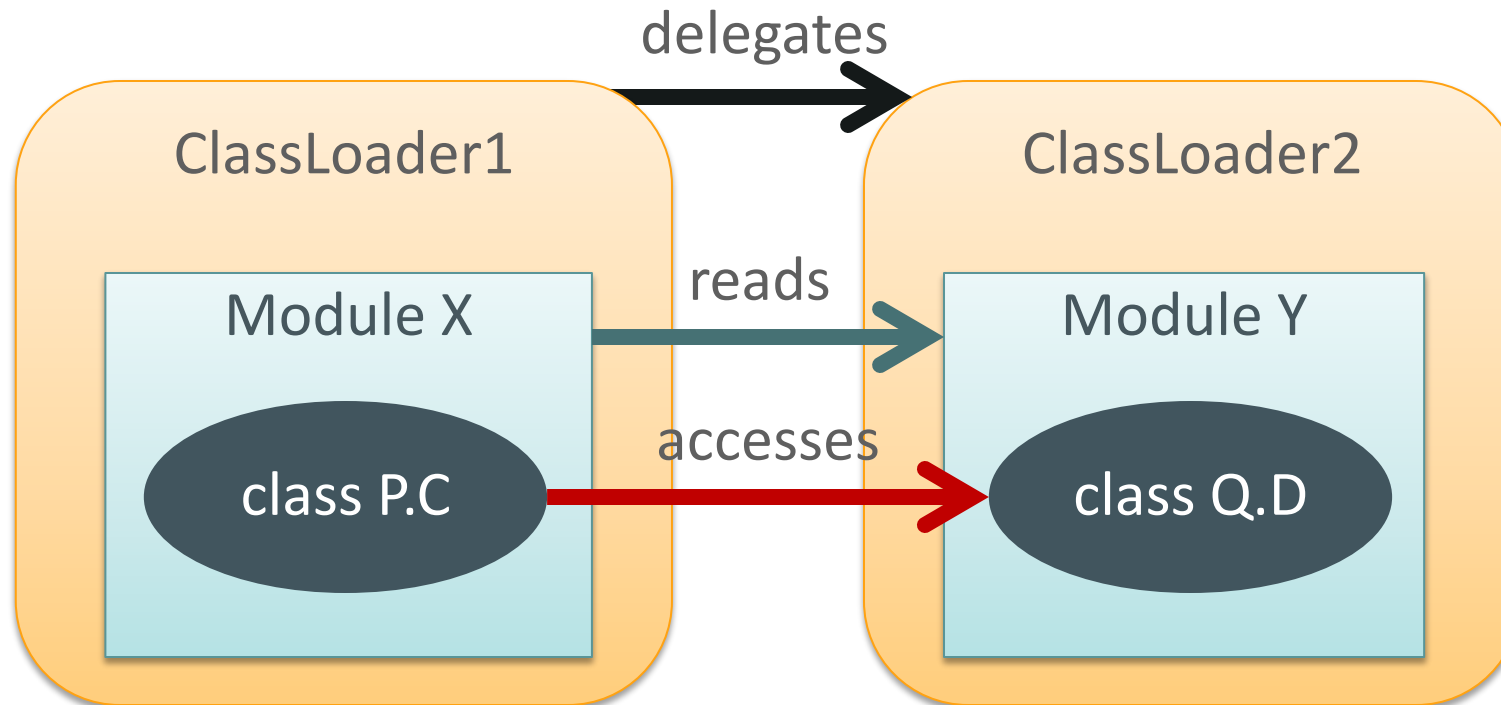


Well-formed maps

```
String moduleName -> {  
    switch (moduleName) {  
        case "java.base":  
        case "java.logging":  
            return BOOTSTRAP_LDR;  
        default:  
            return APP_LDR;  
    }  
}
```

- Different modules with the same package map to different loaders.
- (Loader delegation respects module readability.)

Loaders and Readability



Example: DOM APIs

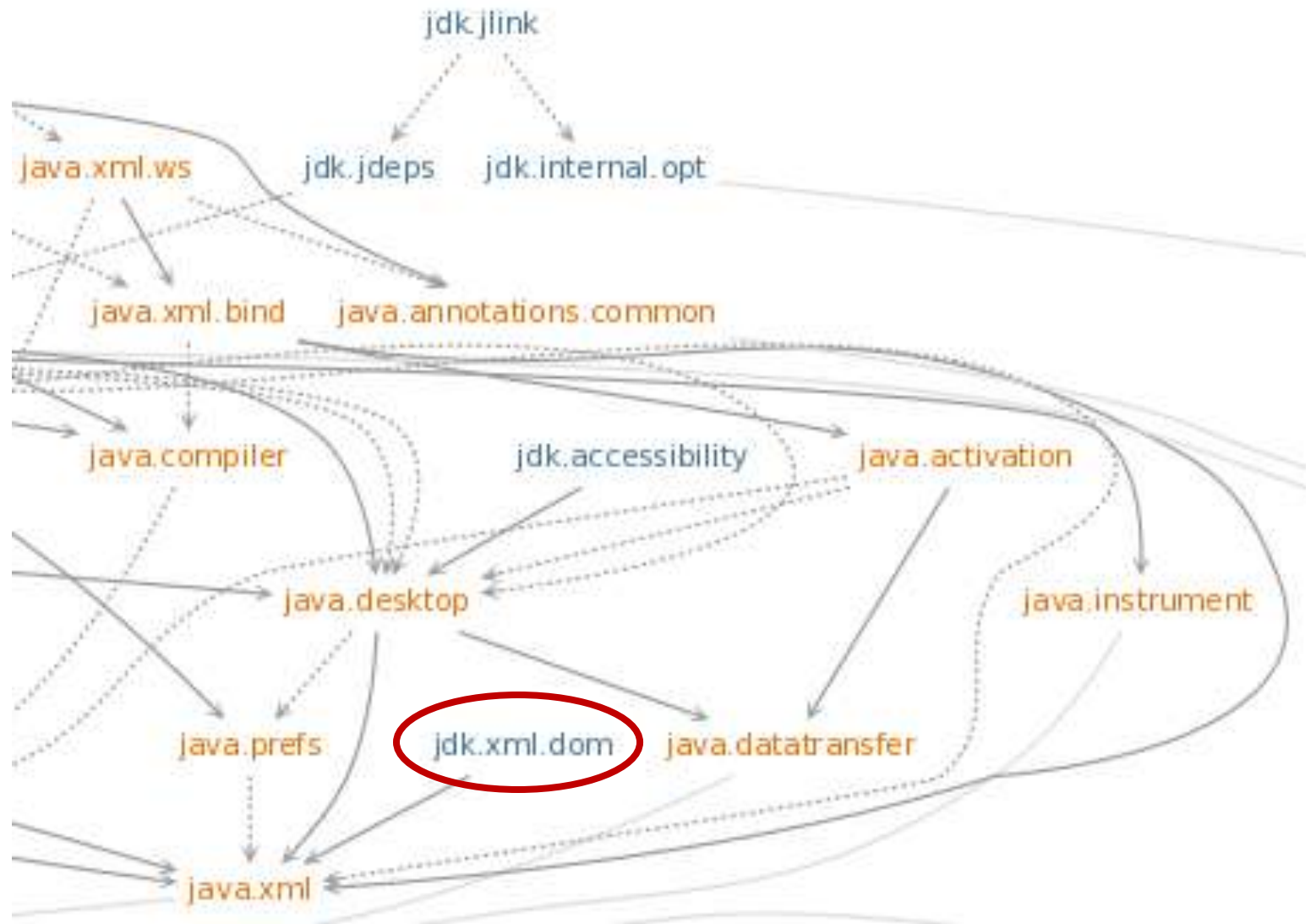
`org.w3c.dom.css`

`org.w3c.dom.html`

`org.w3c.dom.stylesheets`

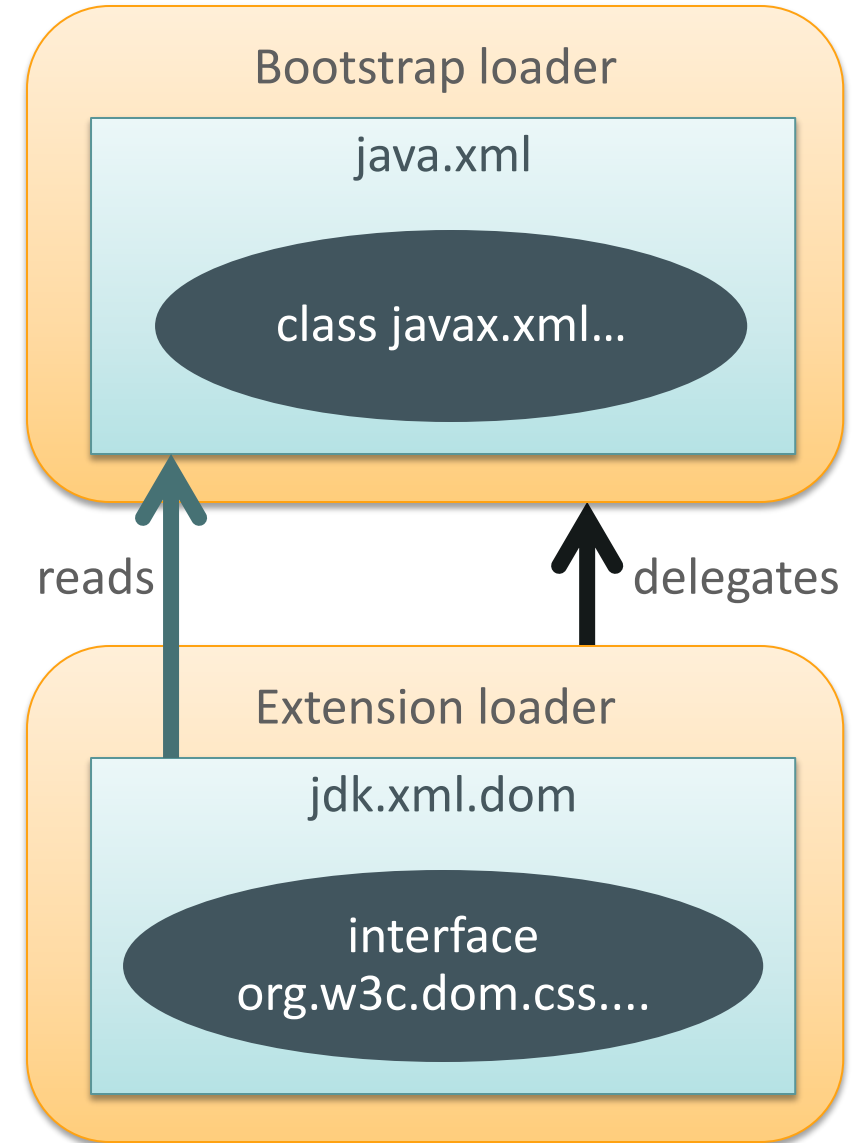
`org.w3c.dom.xpath`

Example: DOM APIs

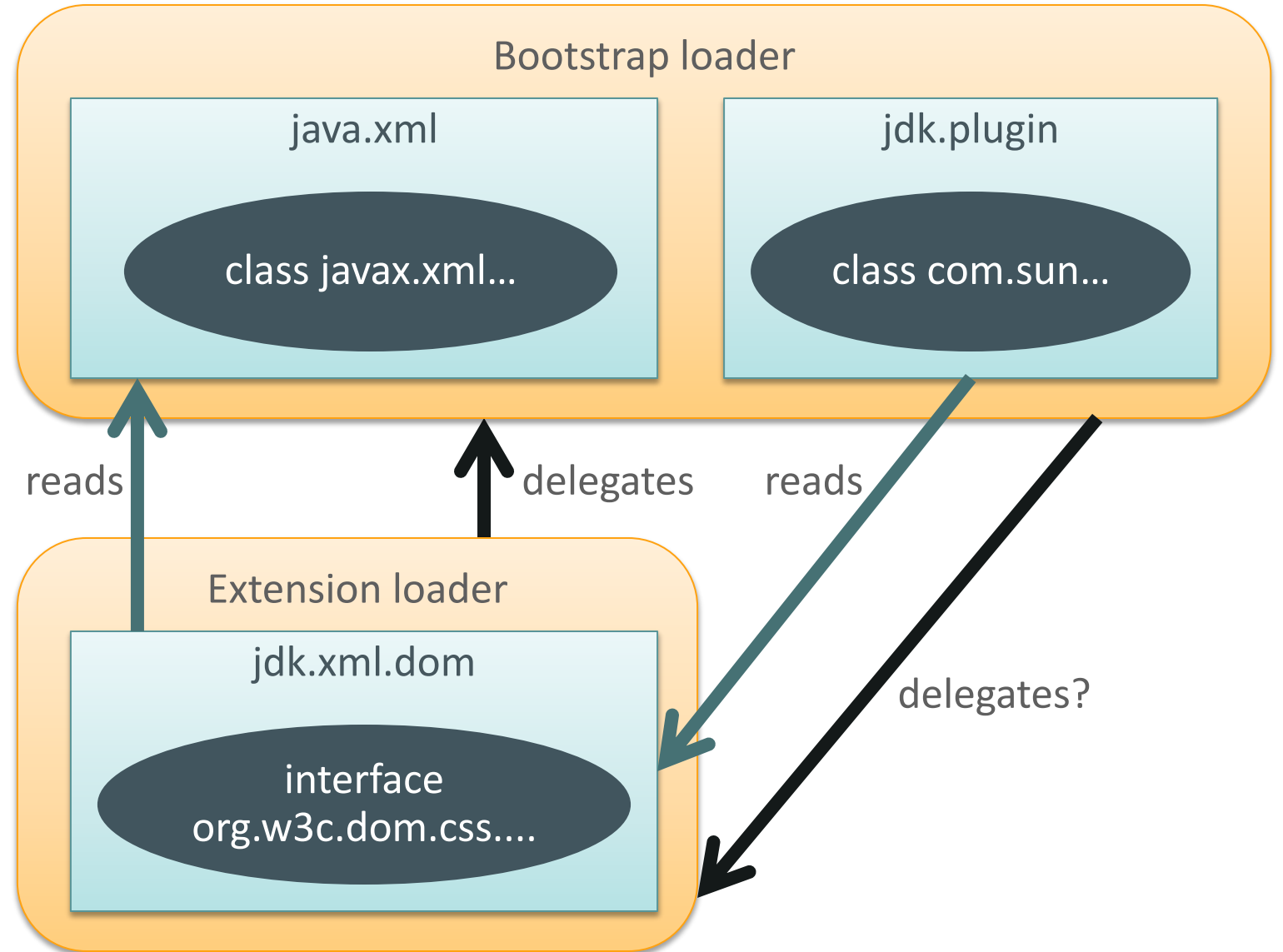


Example: DOM APIs

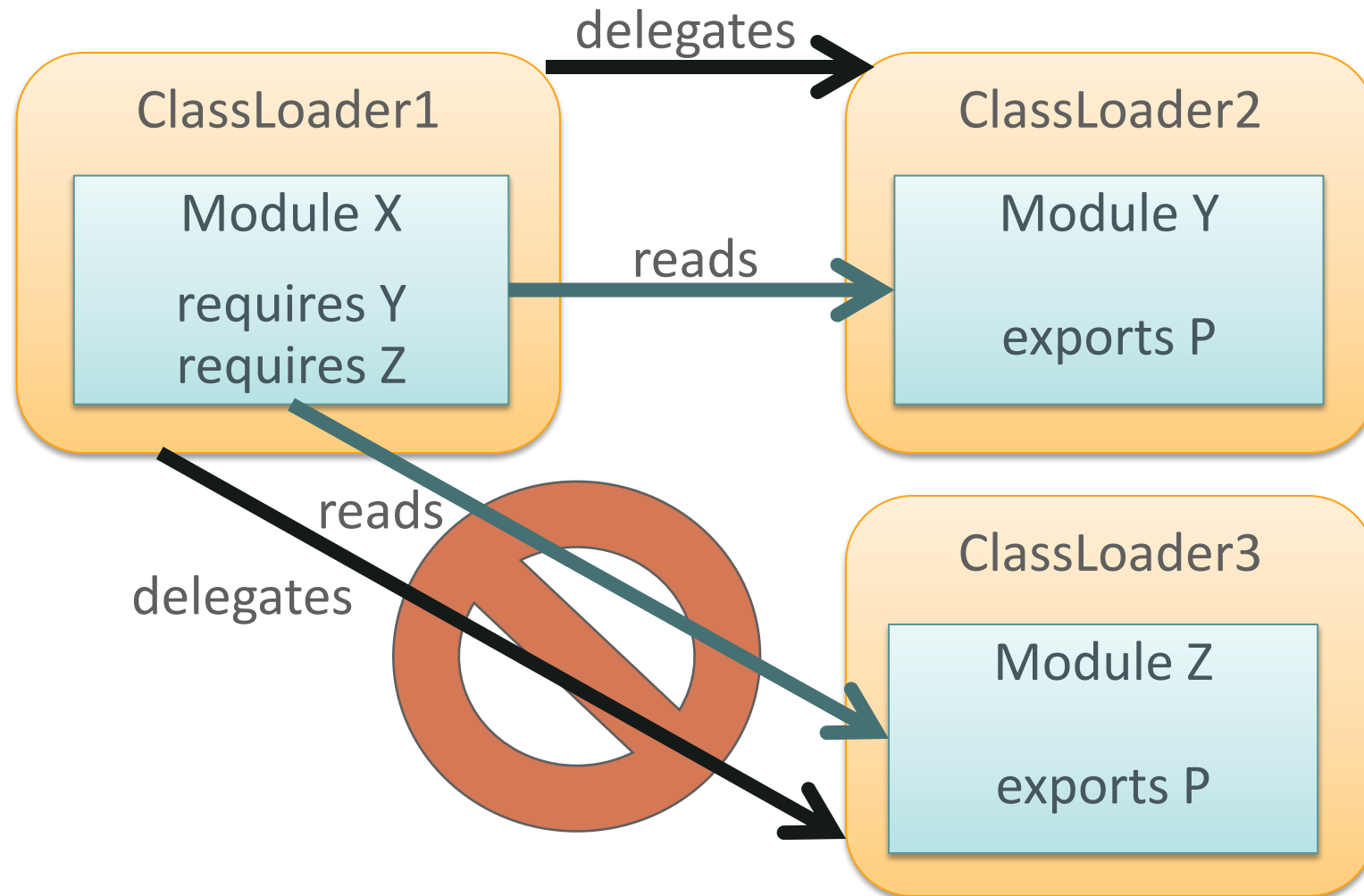
```
module jdk.xml.dom {  
    requires public java.xml;  
  
    exports org.w3c.dom.css;  
    exports org.w3c.dom.html;  
    exports org.w3c.dom.stylesheets;  
    exports org.w3c.dom.xpath;  
}
```



Example: DOM APIs



Loader delegation respects module readability



Split packages (missing class)

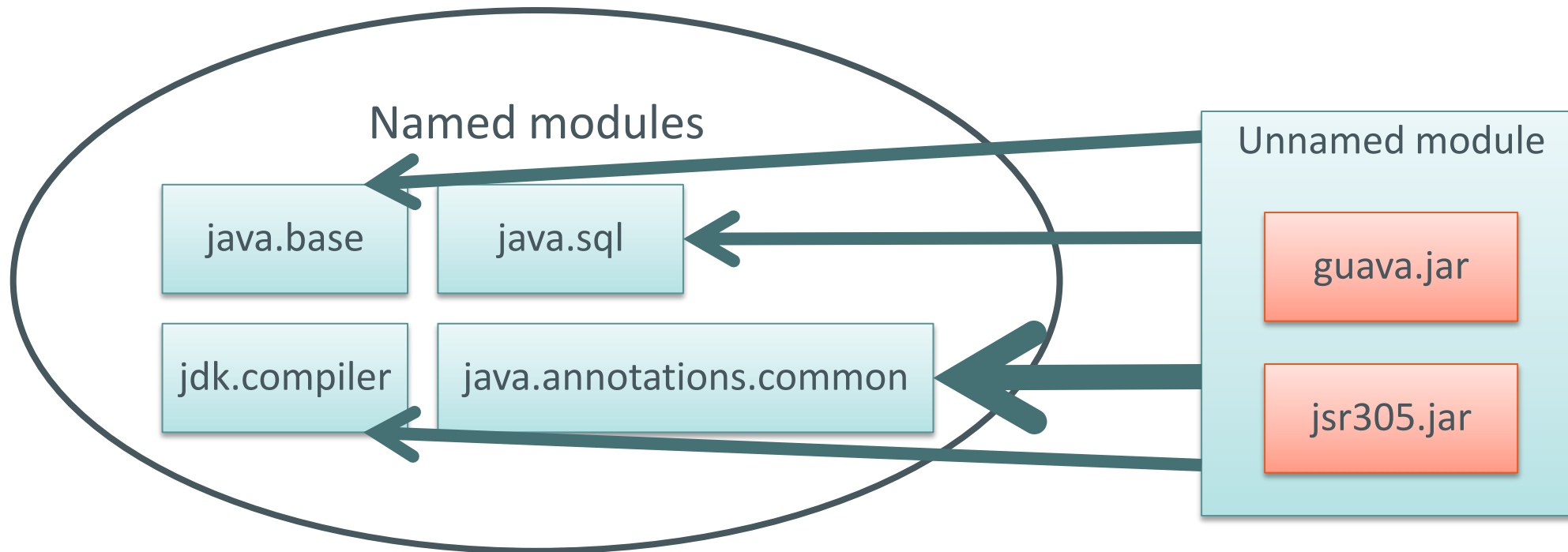
Module java.X

JAR file Y

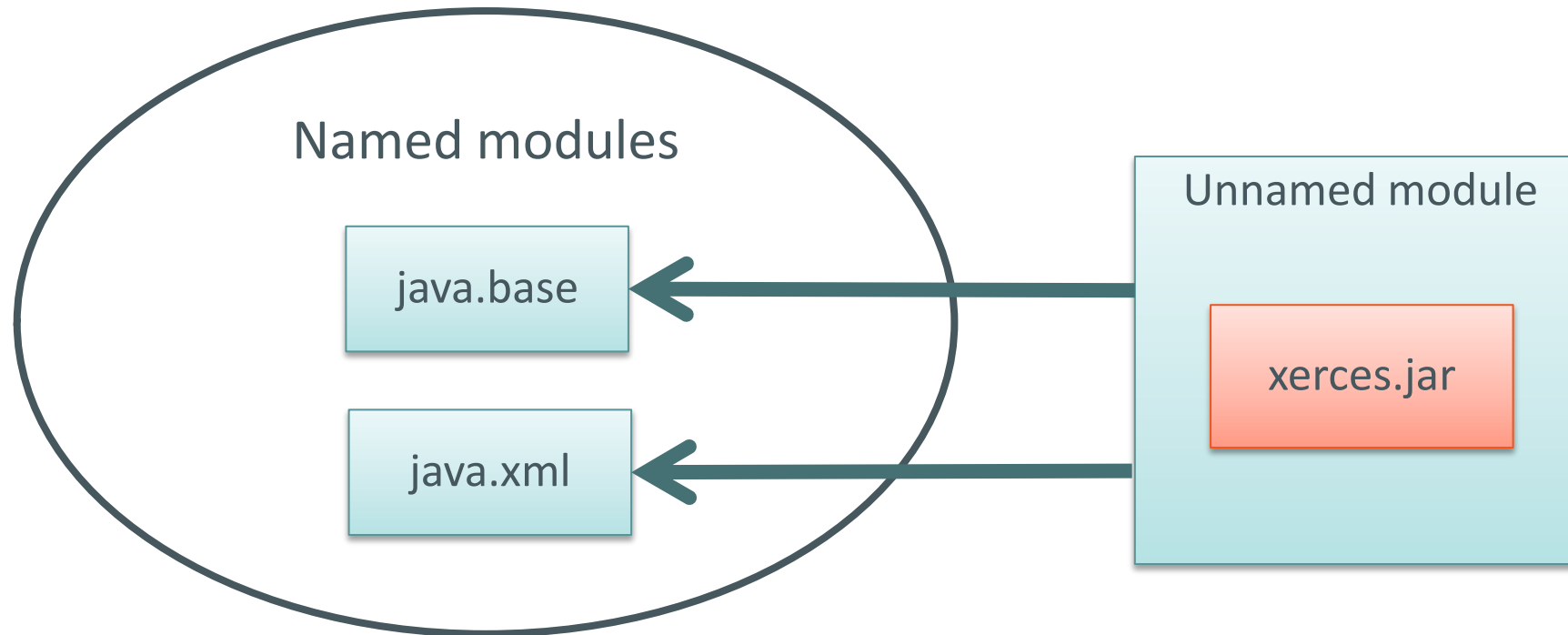
```
module java.X {  
    exports javax.annotation;  
}
```

```
javax/annotation/MyAnno1.class  
javax/annotation/MyAnno2.class
```

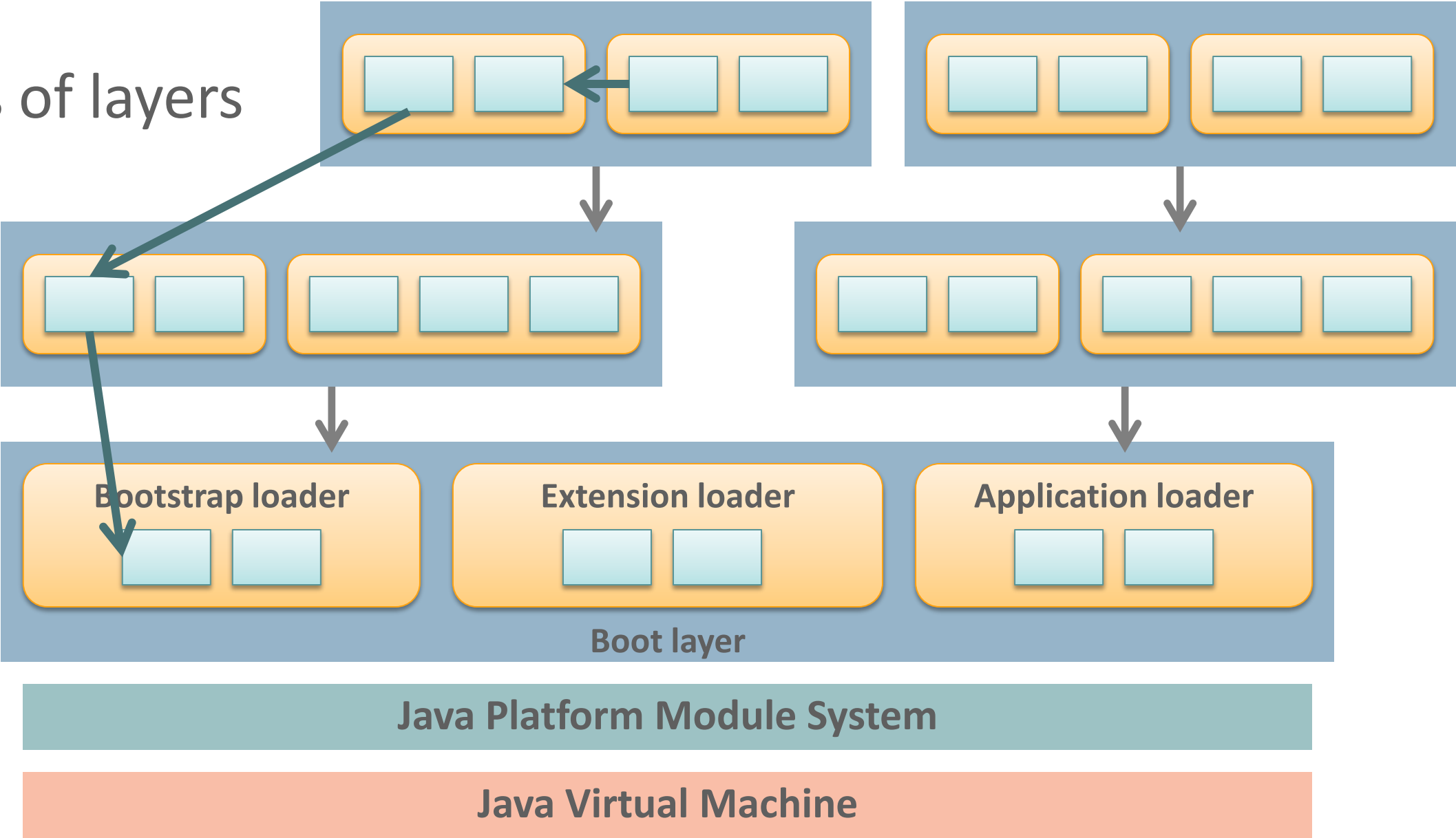
Split packages (missing class)



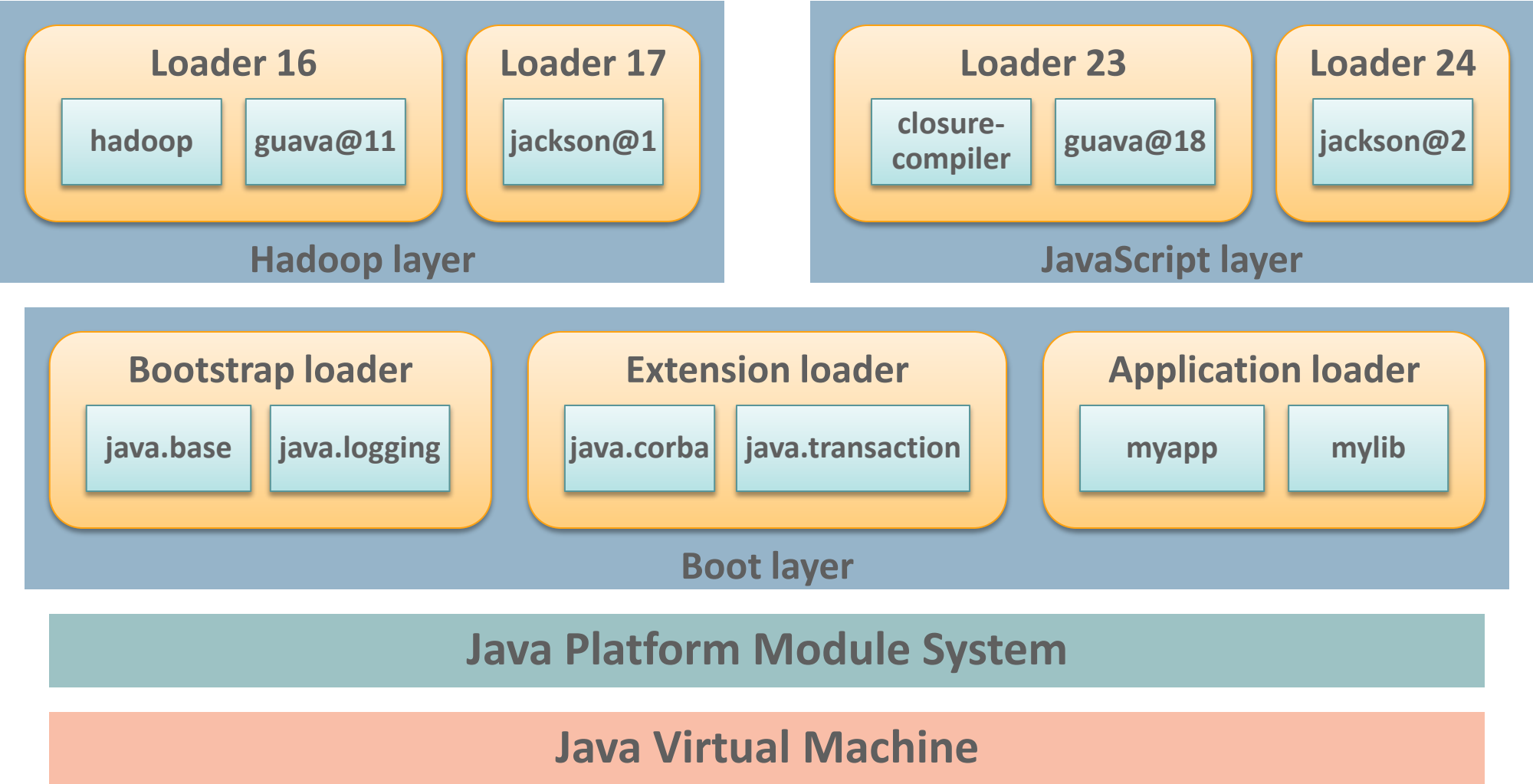
Split packages (duplicate class)



Layers of layers



Layers and Versions



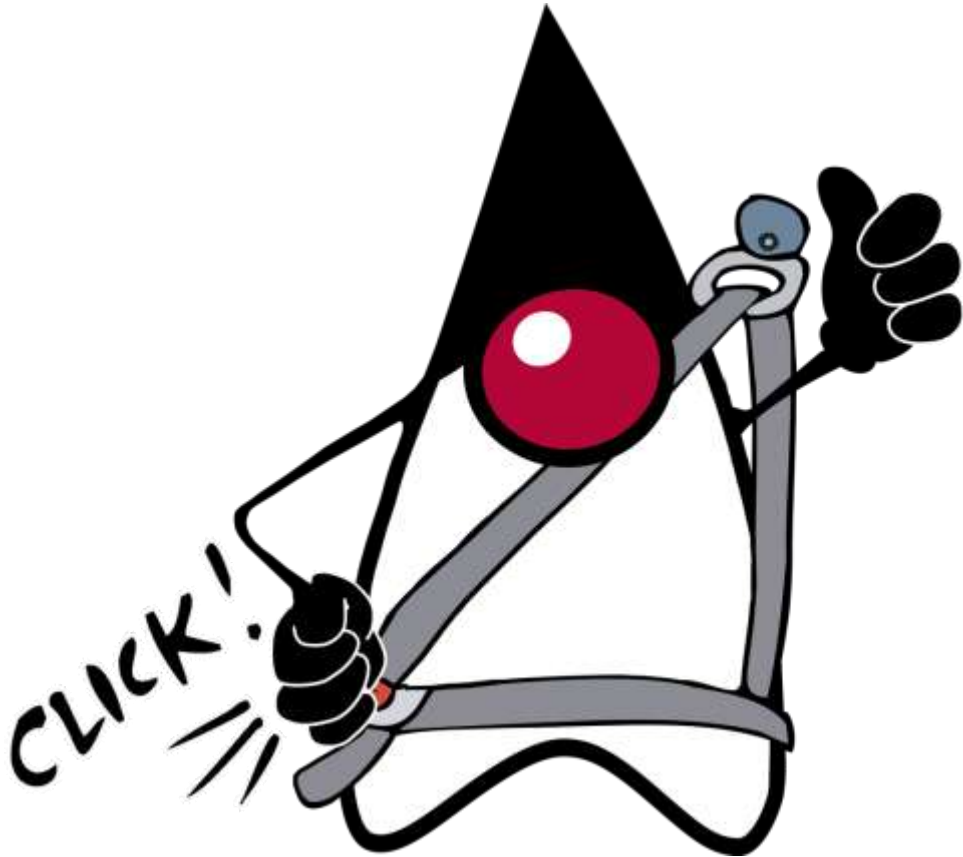
Summary of Part III: Loaders and Layers

- Modules do a better job of encapsulation than class loaders, but class loaders are still necessary.
- Layers control the relationship between modules and class loaders.
- **Assuming class loaders respect the module graph, the system is safe by construction – no cycles or split packages.**

Summary of Summaries

- **Strong encapsulation of modules by the compiler, VM, Core Reflection.**
- **Unnamed and automatic modules help with migration.**
- **The system is safe by construction – no cycles or split packages.**

The module system: a seat belt, not a jetpack



Meta



Personal photo of speaker, France, 2001

What can you do to prepare for JDK 9?

- Try JDK 9 with Jigsaw – jdk9.java.net/jigsaw
- Run `jdeps` on your code and on your classpath.
- Subscribe to `jigsaw-dev` @ OpenJDK to see common problems + solutions.

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

ORACLE®