Understanding Modules

Objectives

After completing this lesson, you should be able to:

- Understand Java modular design principles
- Define module dependencies
- Expose module content to other modules

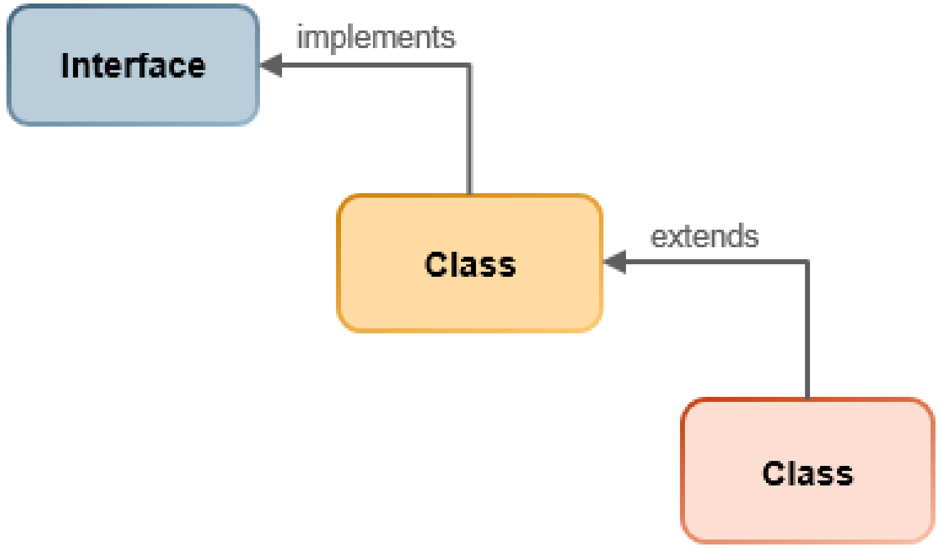
Topics

- Module system: Overview
- JARs
- Module dependencies
- Modular JDK

Reusing Code in Java: Classes

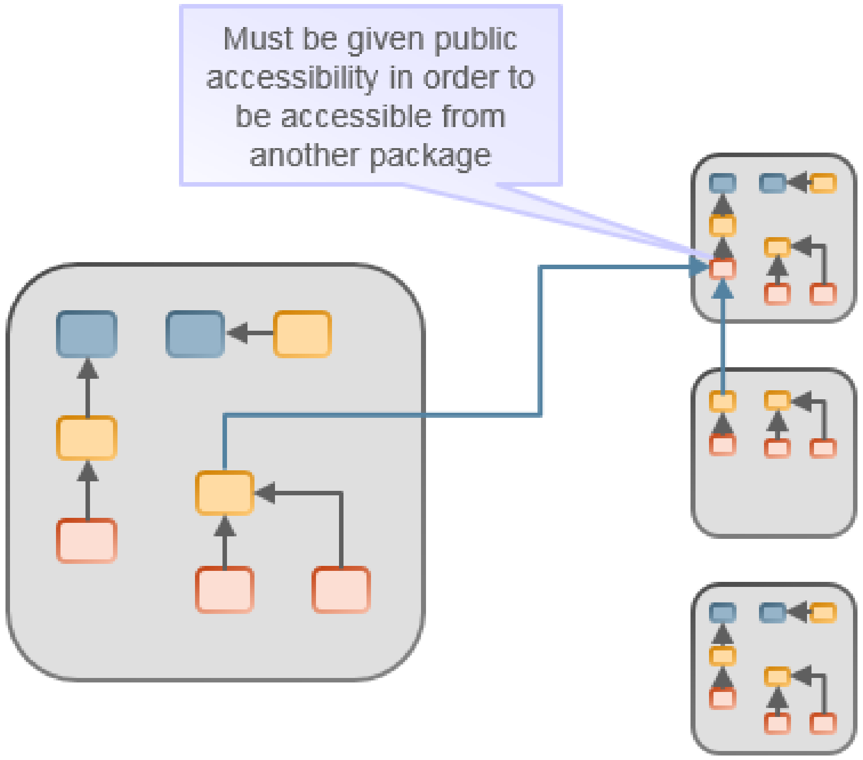
One of the core features of any programming language is its ability to reuse code.

- So that “large” programs can be built from “small” programs.
- In Java the basic unit of reuse has been a class, that is, **programs are classes.**
- Java has good mechanisms for promoting reuse of a class:
  - Inheritance for reusing behavior
  - Interfaces for reusing abstractions



Reusing Code in Java: Packages

- Java also has packages for grouping together similar classes, that is, **programs are packages**
- Packages are grouped in JARs, and JARs are the unit of distribution.


Must be given public accessibility in order to be accessible from another package

Reusing Code in Java: Programming in the Large

- In a large Java codebase, when the application uses several packages and is distributed in many JARs, then it becomes difficult to:
  - Control which classes and interfaces are re-using the code
- The only way to share code between packages is through the `public` modifier. But then the code is shared with everyone.
- Packages are a great way to organize classes, but is there a way to organize packages?

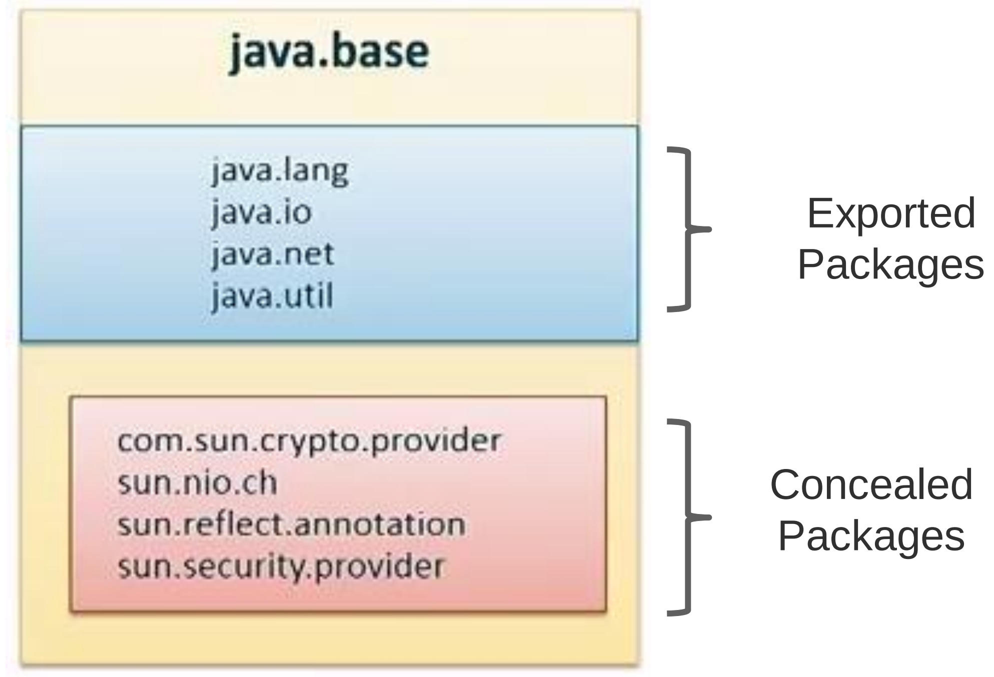
What Is a Module?

A module is a set of packages that is designed for reuse.

- Modularity was introduced in JDK 9.
- Modularity adds a higher level of aggregation above packages.
- A module is a reusable group of related packages, as well as resources (such as images and XML files) and a module descriptor, that is, **programs are modules**.
- In a module, some of the packages are:
  - Exported packages: Intended for use by code outside the module
  - Concealed packages: Internal to the module; they can be used by code inside the module but not by code outside the module

Example: `java.base` Module

- A module is a set of exported packages and concealed packages.
- This is strong encapsulation.

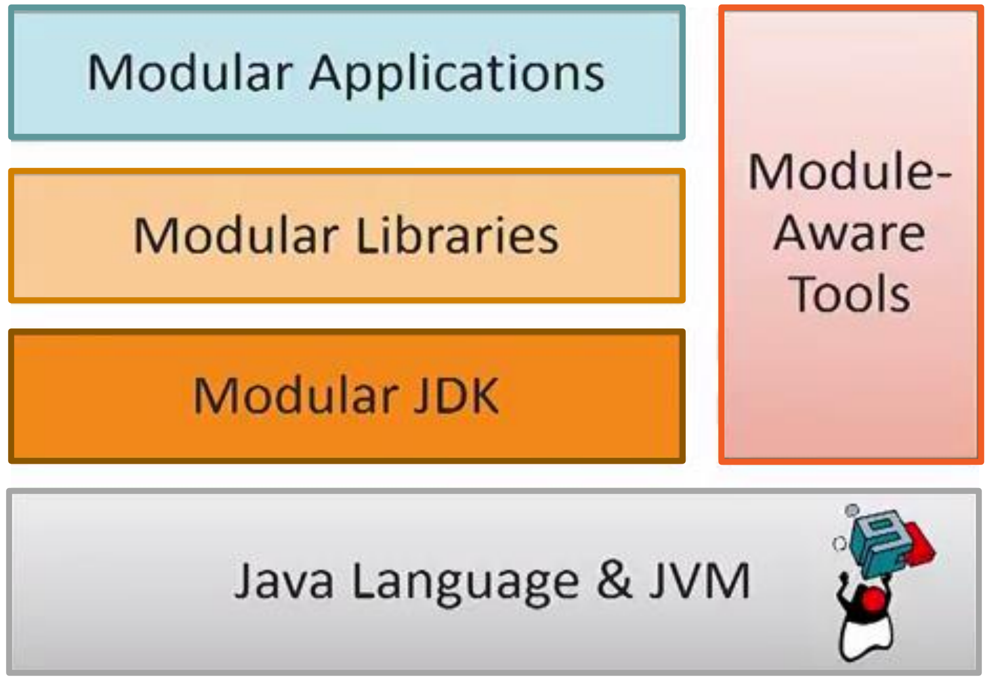


Module System

- The module system:
  - Is usable at all levels:
    - Applications
    - Libraries
    - The JDK itself
  - Addresses reliability, maintainability, and security
  - Supports creation of applications that can be scaled for small computing devices

Modular Development in JDK 9

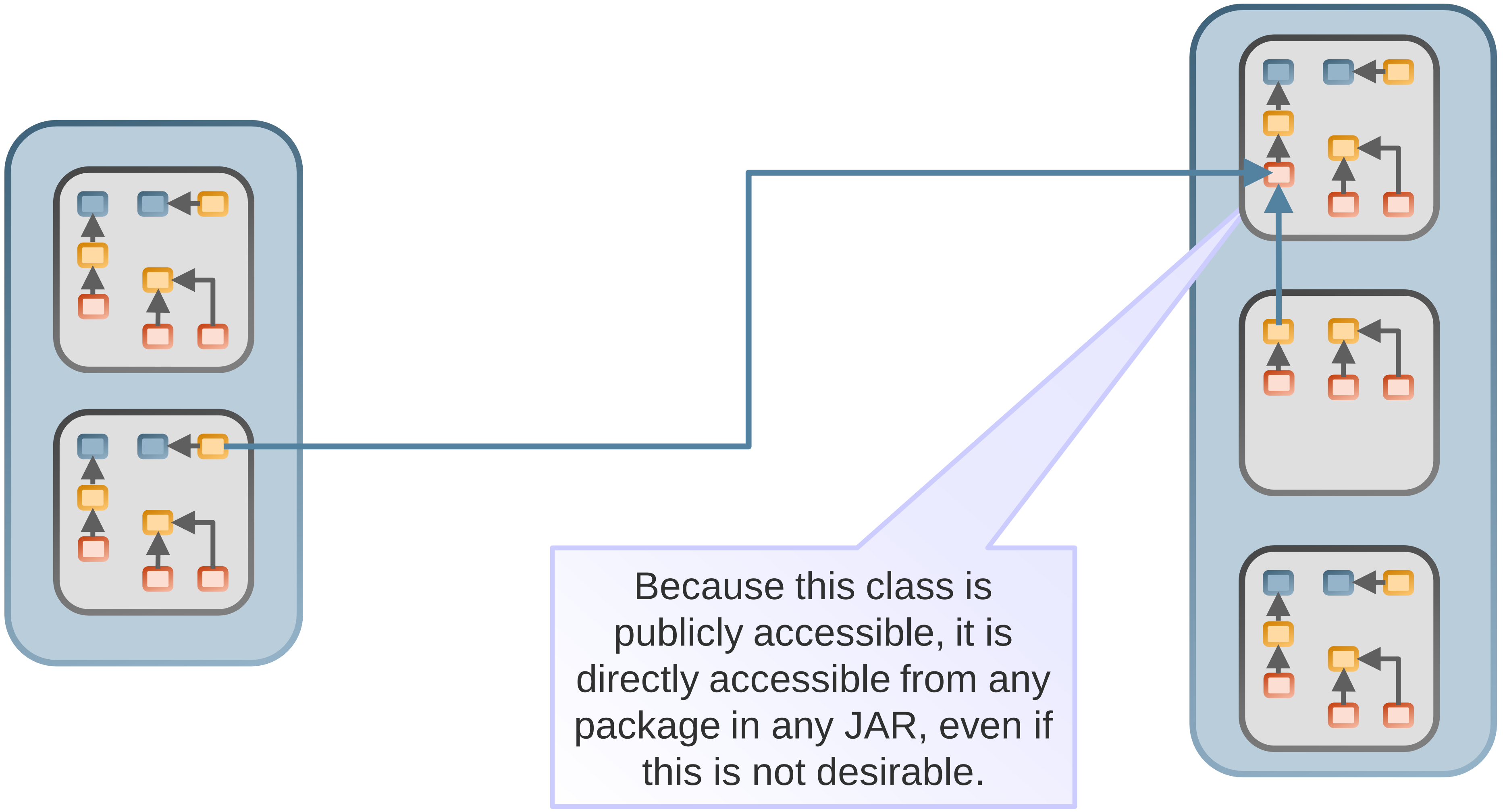
JDK 9 enables modular development all the way down.



Topics

- Module system: Overview
- JARs
- Module declarations
- Modular JDK

JARs



Because this class is publicly accessible, it is directly accessible from any package in any JAR, even if this is not desirable.

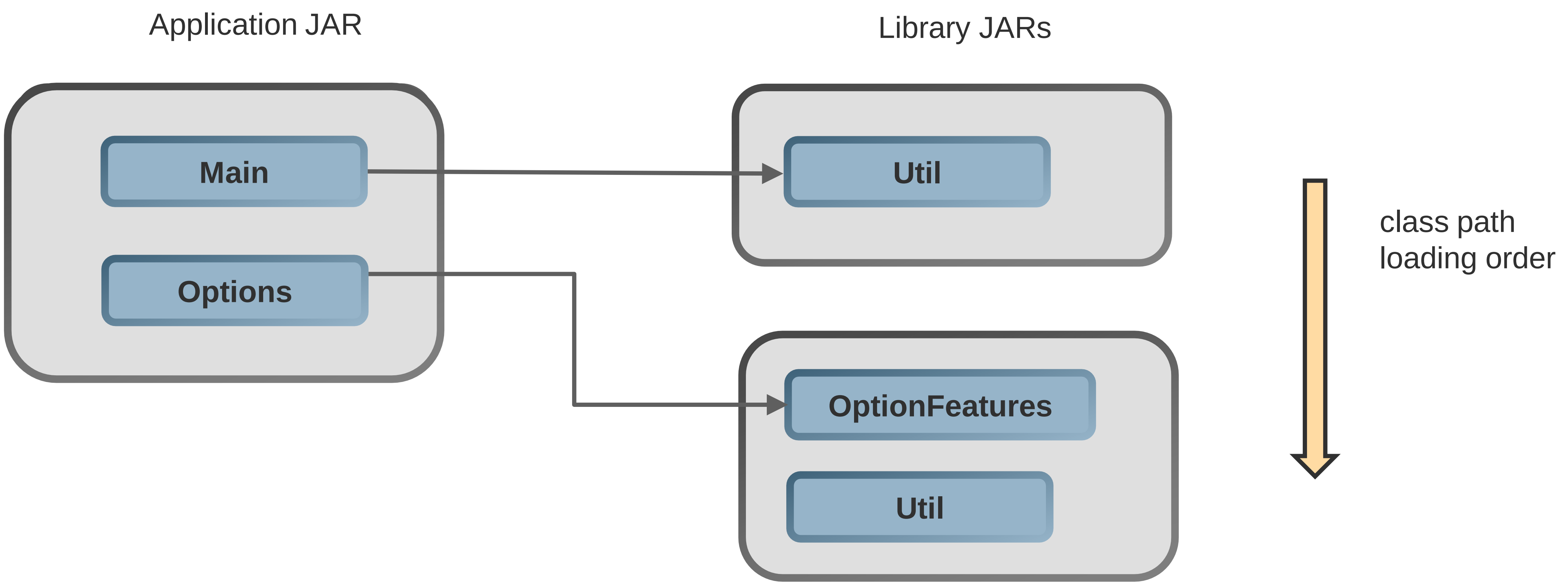
JAR Files and Distribution Issues

- JAR files are:
  - Typically used for packaging the class files for:
    - The application
    - The libraries
  - Composed of a set of packages with some additional metadata
    - For example: main class to run, class path entries, multi-release flag
  - Added to the class path in order that their contents (classes) are made available to the JDK for compilation and running
    - Some applications may have hundreds of JARs in the class path.

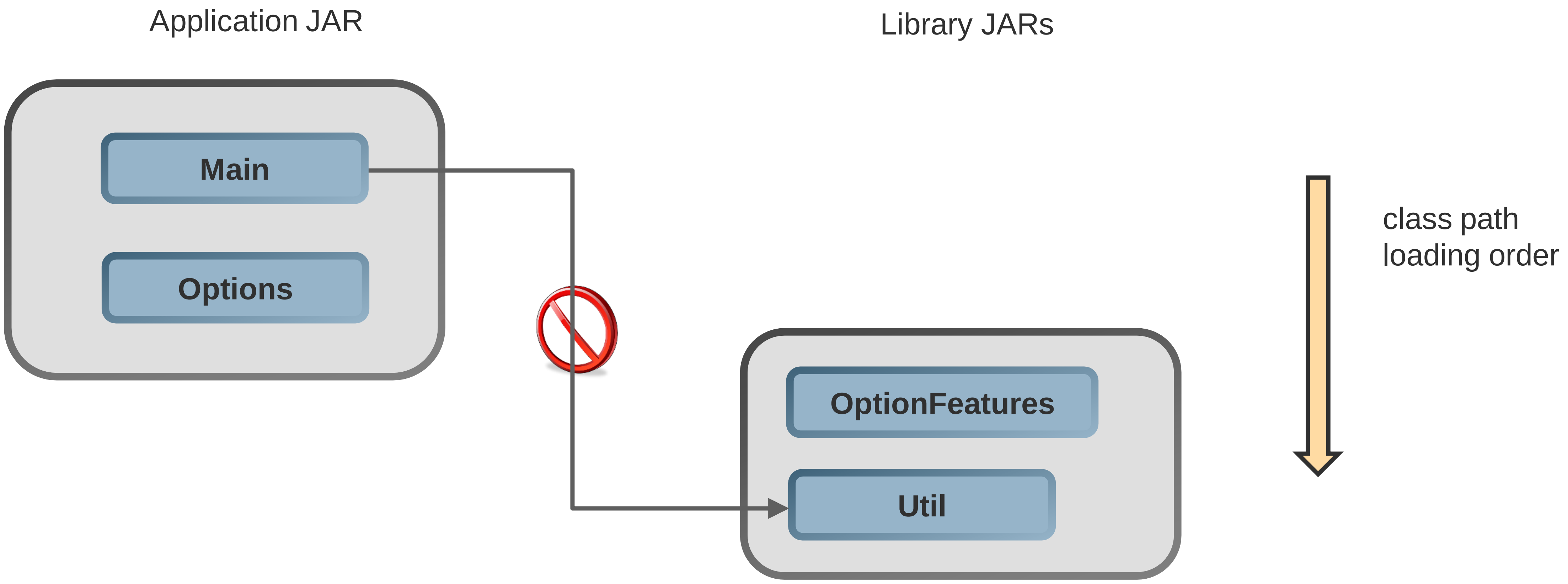
Class Path Problems

- JARs in the class path can have duplicate classes and/or packages.
- Java runtime tries to load each class as it finds it.
  - It uses the first class it finds in class path, even if another similarly named class exists.
  - The first class could be the wrong class if several versions of a library exist in the class path.
  - Problems may occur only under specific operation conditions that require a particular class.

Example JAR Duplicate Class Problem 1



Example JAR Duplicate Class Problem 2



Module System: Advantages

- Addresses the following issues at the unit of distribution/reuse level:
  - Dependencies
  - Encapsulation
  - Interfaces
- The unit of reuse is the module.
  - It is a full-fledged Java component.
  - It explicitly declares:
    - Dependencies on other modules
    - What packages it makes available to other modules
      - Only the public interfaces in those available packages are visible outside the module.

Accessibility Between Classes

Accessibility (JDK 1 – JDK 8)

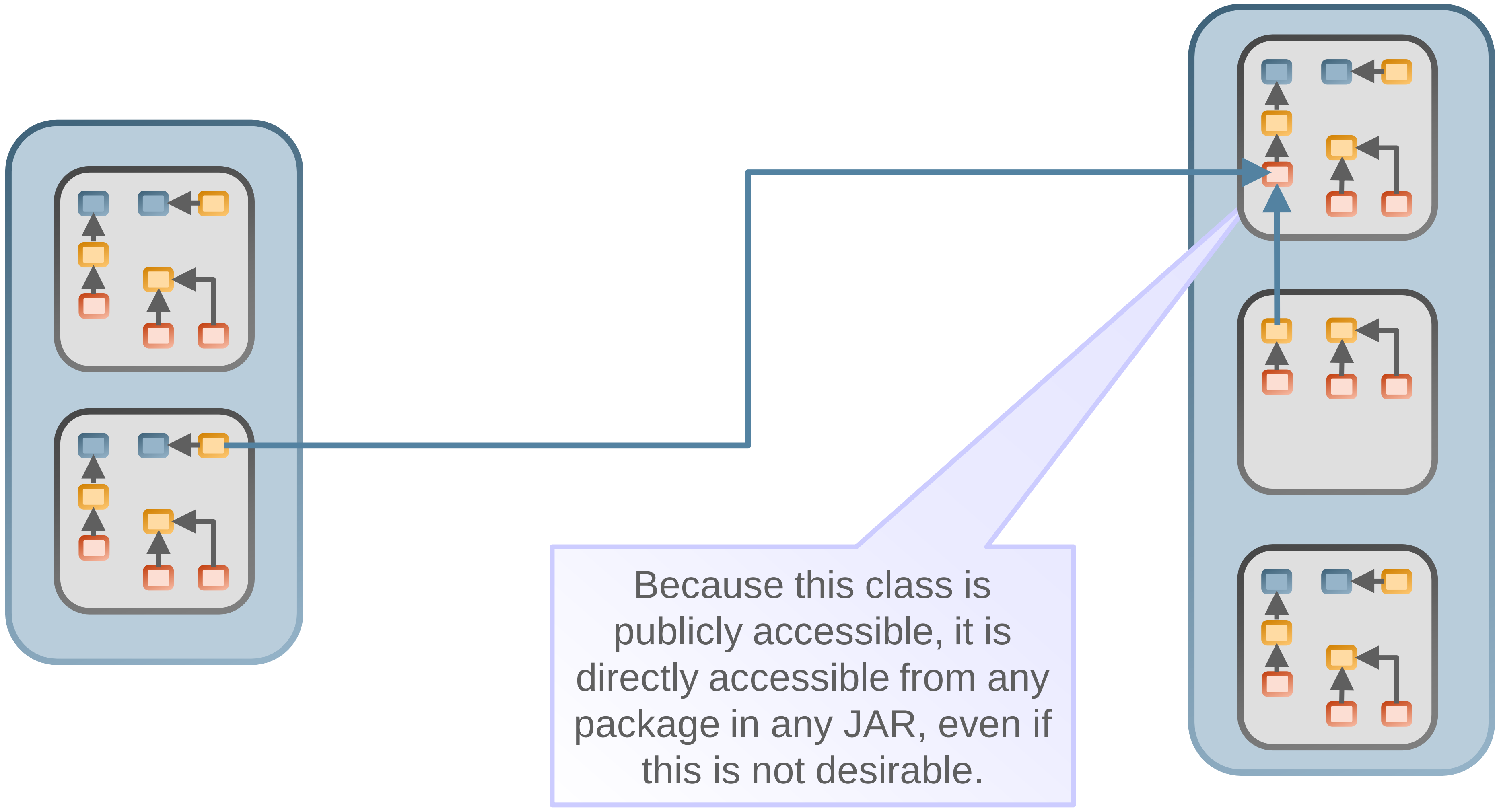
- `public`
- `protected`
- <package>
- `private`

Accessibility (JDK 9 and later)

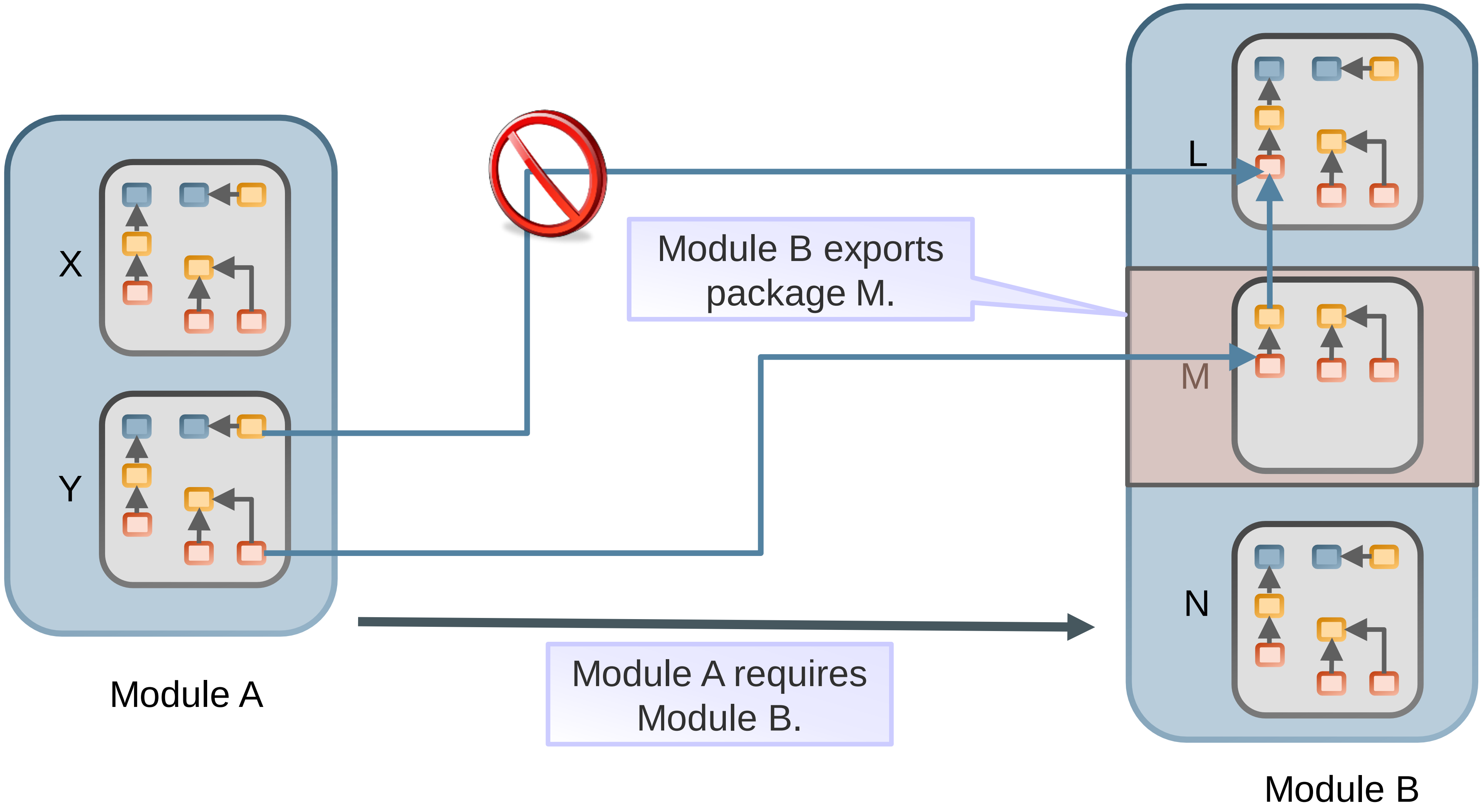
- `public` to everyone
- `public`, but only to specific modules
- `public` only within a module
- `protected`
- <package>
- `private`

- `public` no longer means "accessible to everyone."
- You must edit the `module-info` classes to specify how modules read from each other.

Access Across Non-Modular JARs


Because this class is publicly accessible, it is directly accessible from any package in any JAR, even if this is not desirable.

Access Across Modules



Topics

- Module system: Overview
- JARs
- Module dependencies
- Modular JDK

`module-info.java`

- A module must be declared in a **`module-info.java`** file.
  - Metadata that specifies the module's dependencies, the packages the module makes available to other modules, and more.
- Each module declaration begins with the keyword `module`, followed by a unique module name and a module body enclosed in braces, as in:

```
module modulename
{
}
```

- The module declaration's body can be empty or may contain various module directives, such as `requires, exports`.
- Compiling the module declaration creates the **module descriptor**, which is stored in a file named **`module-info.class`** in the module's root folder.

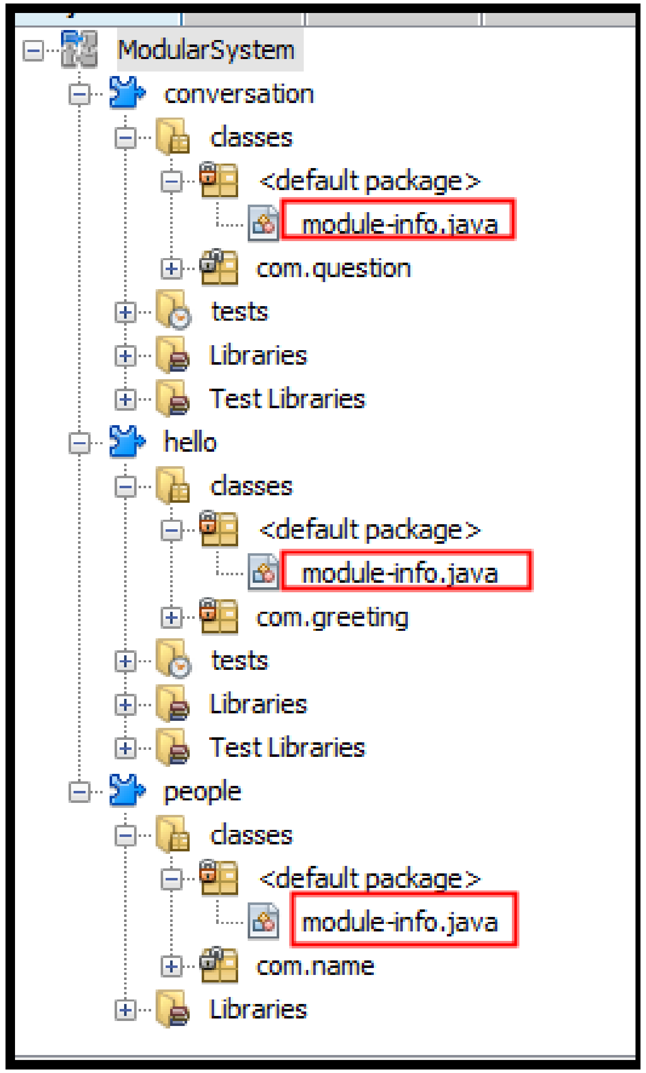
Example: `module-info.java`

```
module soccer {

    requires competition;
    requires gameapi;
    requires java.logging;
    exports soccer to competition;
}
```

Creating a Modular Project

- Name of the project
- Place `module-info.java` in the root directory of the packages that you want to group as a module.
- NetBeans marks this as the default package
- One modular JAR is produced for every module.
  - Modular JARs become the unit of release and reuse.
  - They're intended to contain a very specific set of functionality.



```
ModularSystem
  conversation
    classes
      <default package>
        module-info.java
      com.question
    tests
    Libraries
    Test Libraries
  hello
    classes
      <default package>
        module-info.java
      com.greeting
    tests
    Libraries
    Test Libraries
  people
    classes
      <default package>
        module-info.java
      com.name
    Libraries
```

`exports` Module Directive

- An `exports` module directive specifies one of the module's packages whose public types (and their nested `public` and `protected` types) should be accessible to code in all other modules.
- For example:
  - The `conversation` module's `module-info` class explicitly states which packages it's willing to let other modules read, using the `exports` keyword.

```
module conversation {

    exports com.question;
}
```

`exports...to` Module Directive

- An `exports…to` directive enables you to specify in a comma-separated list precisely which module’s or modules’ code can access the exported package; this is known as a qualified export.
- Consider this, the conversation module's `module-info` class explicitly states:
  - Which packages it's willing to allow to be read
  - Which modules are allowed to read a particular package
- This is done with the `exports` and `to` keywords.

```
module conversation {

    exports com.question to people;
}
```

`requires` Module Directive

A `requires` module directive specifies that this module depends on another module. This relationship is called a module dependency.

- Each module must explicitly state its dependencies.
  - When module A requires module B, module A is said to read module B and module B is read by module A.
- To specify a dependency on another module, use `requires`, as in:

```
requires modulename;
```

- Example: The `main` module's `module-info` class explicitly lists which modules it depends on.

```
module hello {
    requires people;
}
```

`requires transitive` Module Directive

- To specify a dependency on another module and to ensure that other modules reading your module also read that dependency known as implied readability, use requires transitive, as in:

```
requires transitive modulename;
```

Implementing a Requires Transitively Relationship

```
module hello {
    requires people;
}
```

```
module people {
    exports com.name;
    requires transitive conversation;
}
```

Summary of Keywords

Keywords and Syntax	Description
export <package>	Declares which package is eligible to be read
export <package> to <module>	Declares which package is eligible to be read by a specific module
requires <module>	Specifies another module to read from
requires transitive <module>	Specifies another module to read from. The relationship is transitive in that indirect access is given to modules requiring the current module.

- These are restricted keywords.
- Their creation won't break existing code.
- They're only available in the context of the `module-info` class.

Compiling Modules

- When compiling a module, specify all of your java sources from various packages that you want this module to contain.
- Make sure to include packages that are exported by this module to other modules and a module-info.

```
javac -d <complied output folder>  <list of source code file paths
including module-info>
```

- For example:

```
javac -d mods --module-source-path src $(find src -name "*.java")
```

Running a Modular Application

- Running a modular application:

```
java --module-path <path to complied module>
     --module <module name>/<package name>.<main class name>
```

- For example:

```
java -p mods -m hello/com.greeting.Main
```

Note: To execute a modular application, don't use CLASSPATH !

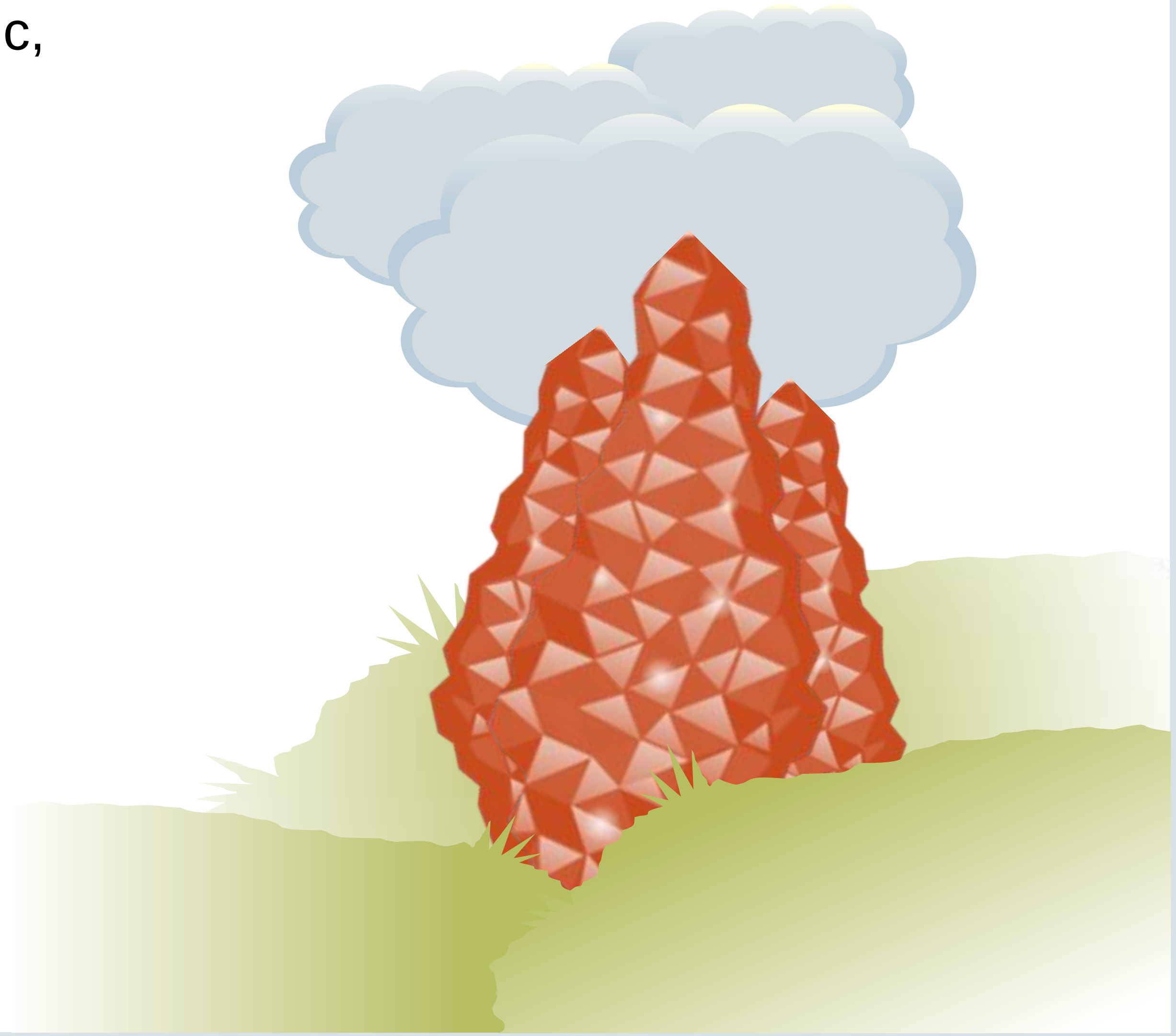
Topics

- Module system: Overview
- JARs
- Module declarations
- Modular JDK

The JDK

Before JDK 9, the JDK was huge and monolithic, thus increasing the:

- Download time
- Startup time
- Memory footprint



The Modular JDK

- In JDK 9, the monolithic JDK is broken into several modules. It now consists of about 90 modules.
- Every module is a well-defined piece of functionality of the JDK:
  - All the various frameworks that were part of the prior releases of JDK are now broken down into their modules.
  - For example: Logging, Swing, and Instrumentation
- The modular JDK:
  - Makes it more scalable to small devices
  - Improves security and maintainability
  - Improves application performance

Listing the Modules in JDK 9

```
$java --list-modules
```

```
java.activation@9.0.1
java.base@9.0.1
java.compiler@9.0.1
java.corba@9.0.1
java.datatransfer@9.0.1
java.desktop@9.0.1
java.instrument@9.0.1
java.jnlp@9.0.1
java.logging@9.0.1
java.management@9.0.1
java.management.rmi@9.0.1
java.naming@9.0.1
java.prefs@9.0.1
java.rmi@9.0.1
java.scripting@9.0.1
java.se@9.0.1
java.se.ee@9.0.1
java.security.jgss@9.0.1
java.security.sasl@9.0.1
java.smartcardio@9.0.1
java.sql@9.0.1
java.sql.rowset@9.0.1
java.transaction@9.0.1
```

```
java.xml@9.0.1
java.xml.bind@9.0.1
java.xml.crypto@9.0.1
java.xml.ws@9.0.1
java.xml.ws.annotation@9.0.1
javafx.base@9.0.1
javafx.controls@9.0.1
javafx.deploy@9.0.1
javafx.fxml@9.0.1
javafx.graphics@9.0.1
javafx.media@9.0.1
javafx.swing@9.0.1
javafx.web@9.0.1
jdk.accessibility@9.0.1
jdk.attach@9.0.1
jdk.charsets@9.0.1
jdk.compiler@9.0.1
jdk.crypto.cryptoki@9.0.1
jdk.crypto.ec@9.0.1
jdk.crypto.mscapi@9.0.1
jdk.deploy@9.0.1
jdk.deploy.controlpanel@9.0.1
jdk.dynalink@9.0.1
```

```
jdk.hotspot.agent@9.0.1
jdk.httpserver@9.0.1
jdk.incubator.httpclient@9.0.1
jdk.internal.ed@9.0.1
jdk.internal.jvmstat@9.0.1
jdk.internal.le@9.0.1
jdk.internal.opt@9.0.1
jdk.internal.vm.ci@9.0.1
jdk.jartool@9.0.1
jdk.javadoc@9.0.1
jdk.javaws@9.0.1
jdk.jcmd@9.0.1
jdk.jconsole@9.0.1
jdk.jdeps@9.0.1
jdk.jdi@9.0.1
jdk.jdwp.agent@9.0.1
jdk.jfr@9.0.1
jdk.jlink@9.0.1
jdk.jshell@9.0.1
jdk.jsobject@9.0.1
jdk.jstatd@9.0.1
jdk.localedata@9.0.1
jdk.management@9.0.1
jdk.management.agent@9.0.
jdk.management.cmm@9.0.1
```

```
jdk.management.jfr@9.0.1
jdk.management.resource@9
jdk.naming.dns@9.0.1
jdk.naming.rmi@9.0.1
jdk.net@9.0.1
jdk.pack@9.0.1
jdk.packager@9.0.1
jdk.packager.services@9.0
jdk.plugin@9.0.1
jdk.plugin.dom@9.0.1
jdk.plugin.server@9.0.1
jdk.policytool@9.0.1
jdk.rmic@9.0.1
jdk.scripting.nashorn@9.0.1
jdk.scripting.nashorn.shell@9.0.1
jdk.sctp@9.0.1
jdk.security.auth@9.0.1
jdk.security.jgss@9.0.1
jdk.snmp@9.0.1
jdk.unsupported@9.0.1
jdk.xml.bind@9.0.1
jdk.xml.dom@9.0.1
jdk.xml.ws@9.0.1
jdk.zipfs@9.0.1
oracle.desktop@9.0.1
oracle.net@9.0.1
```

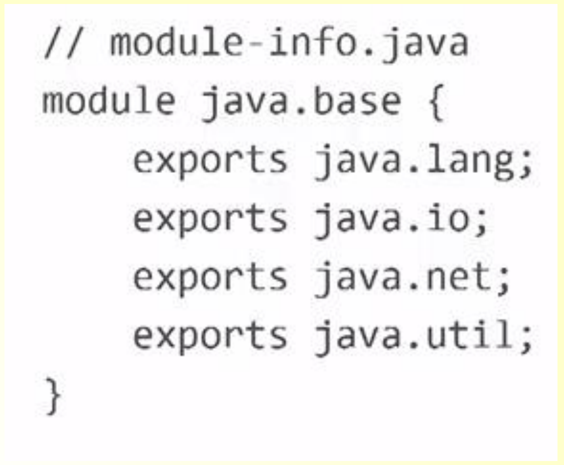
Java SE Modules

These modules are classified into two categories:

1. Standard modules (`java.*` prefix for module names):
   - Part of the Java SE specification.
   - For example: `java.sql` for database connectivity, `java.xml` for XML processing, and `java.logging` for logging
2. Modules not defined in the Java SE 9 platform (`jdk.*` prefix for module names):
   - Are specific to the JDK.
   - For example: `jdk.jshell, jdk.policytool, jdk.httpserver`

The Base Module

- The base module is `java.base.`
  - Every module depends on `java.base`, but this module doesn't depend on any other modules.
  - `java.base` module reference is implicitly included in all other modules.
  - The base module exports all of the platform's core packages.



```
// module-info.java
module java.base {
    exports java.lang;
    exports java.io;
    exports java.net;
    exports java.util;
}
```

```
module hello{
  requires java.base; //implied
  requires java.logging;
}
```

Summary

After completing this lesson, you should be able to:

- Understand Java modular design principles
- Define module dependencies
- Expose module content to other modules

Practice Overview

- 16-1: Creating a Modular Application from the Command Line
- 16-2: Compiling Modules from the Command Line
- 16-3: Creating a Modular Application from NetBeans