16

Understanding Modules

ORACLE

Objectives

After completing this lesson, you should be able to:

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

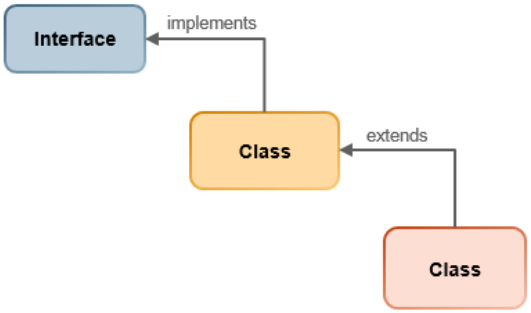
Topics

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

Reusing Code in Java: Classes

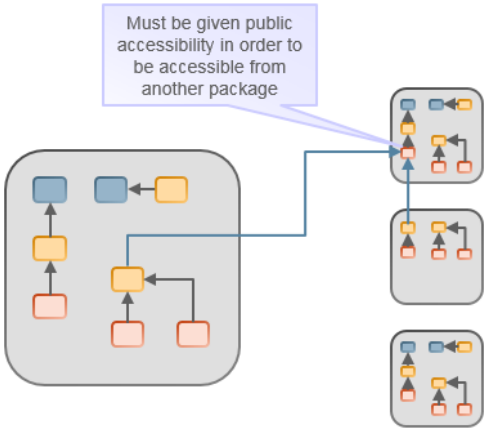
One of the core features of any programming language is its ability to reuse code.

- 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.



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?

Programming in the “large” means programming techniques and component organization at the level of the unit of distribution, rather than at class level. For example, lambda can be considered programming in the “small,” whereas designing modules is programming in the “large.”

Because the module system is concerned with programming in the “large”, designing modules is not likely to be done by developers; it will be done by architects, and developers will program in the “small.”

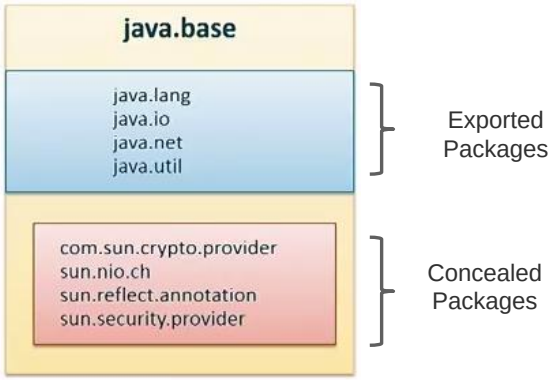
What Is a Module?

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.



java.base module is the foundation module for all other modules similar to java.lang class, which is the foundation for every Java class. You'll learn more about java.base later in this lesson.

In blue are the packages in java.base module, which are intended to be used outside the module. These are its exported packages.

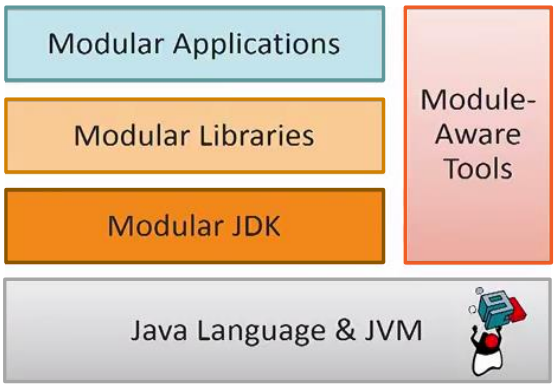
In red are the packages that are internal to the module and are meant to be used by code inside the module and not by code outside the module. These are its concealed packages.

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.



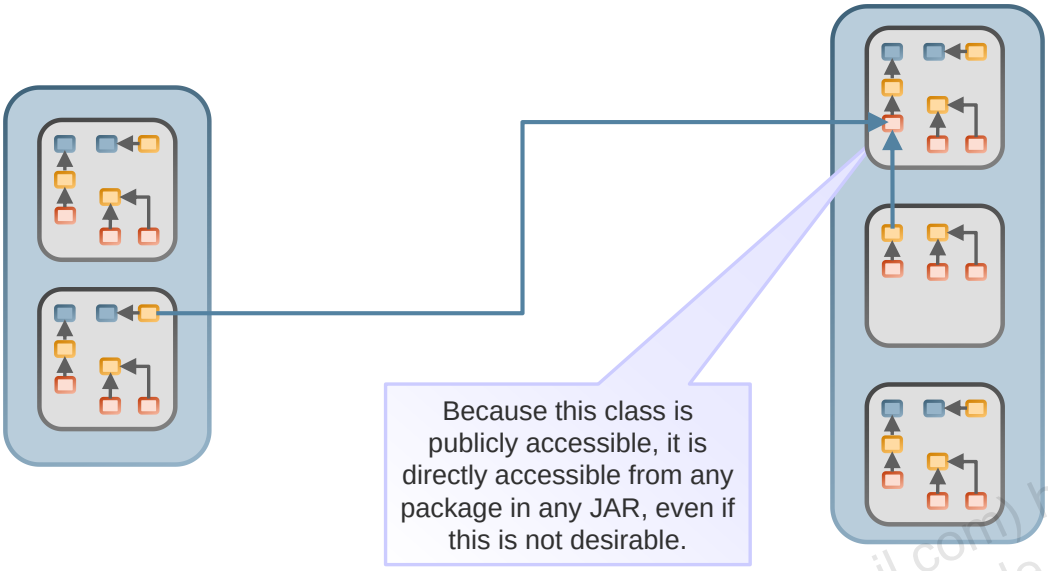
The module system enables modular development all the way down to the JDK. It is built into the Java language and the Java Virtual Machine so that the applications you write and the libraries you consume and even the JDK itself can all be developed and tested and packaged and deployed as modules managed by the module system. Making all levels of an application play by the same rules for modularity has great benefits for reliability, maintainability, and security. It's not essential to use the module system. The previous method of structuring applications by using JARs is still supported. But managing dependencies is an issue as evidenced by current approaches, such as maven. The module system addresses this dependency issue as well as encapsulation at the "large" level.

In JDK 9, the Java language and the Java virtual machine understand modules very well. So the application you run, the libraries you consume, and the JDK itself enable you to develop, package, test, and deploy modules. The modular development also makes your code more reliable and secure.

Topics

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

JARs



JARs are the main unit of distribution. But JARs do not add any access or dependency information to the application.

JAR Files and Distribution Issues

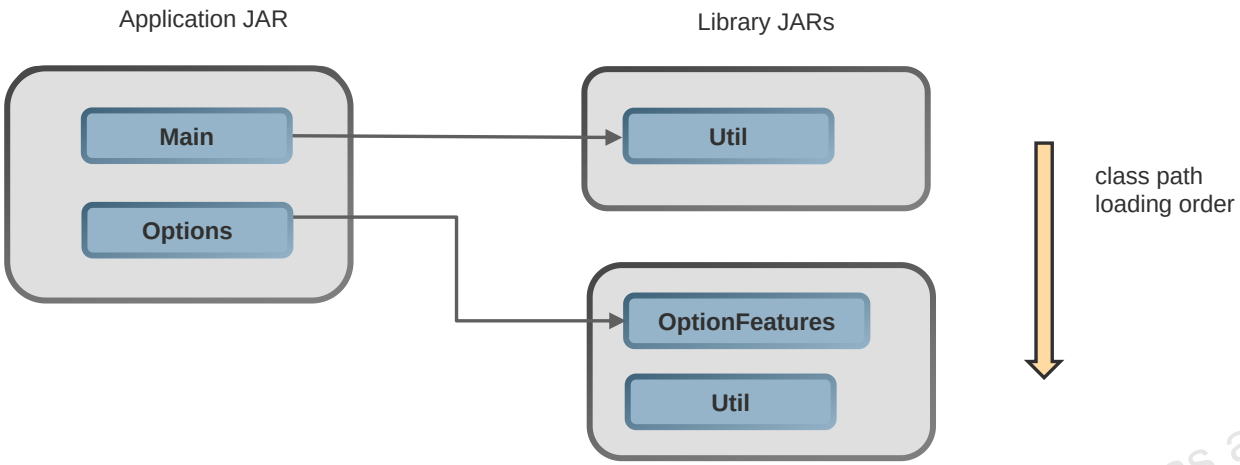
- JAR files are:
  - Typically used for packaging the class files for:
    - The application
    - The libraries
  - Composed of a set of packages with some additional metadata
    - For example: main class to run, class path entries, multi-release flag
  - Added to the class path in order that their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

Most non-trivial applications comprise the application itself and multiple libraries. Typically the application will be in a JAR file, and then each of the libraries will be in its own JAR file.

Class Path Problems

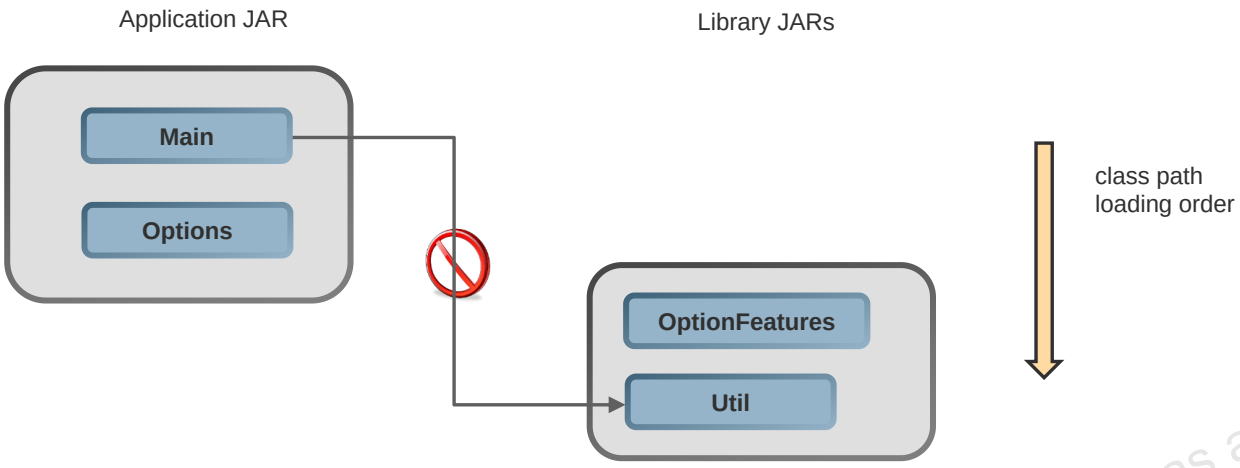
- JARs in the class path can have duplicate classes and/or packages.
- Java runtime tries to load each class as it finds it.
  - It uses the first class it finds in class path, even if another similarly named class exists.
  - The first class could be the wrong class if several versions of a library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

Example JAR Duplicate Class Problem 1



JARs in the class path can be difficult to manage and can cause problems. The diagram illustrates an application JAR that has a class, `Main`, that uses a class, `Util`, in a library JAR. The application is then enhanced by the addition of an `Options` class that uses a class, `OptionFeatures`, in a different JAR. This second library JAR also contains another `Util` class. There are two different classes named `Util`, but this is permitted in JARs in the class path. As long as the `Main` class finds the `Util` class in the first library, the application will work correctly, but the existence of two different `Util` classes makes it fragile.

Example JAR Duplicate Class Problem 2



The diagram is a continuation of the one on the previous slide and shows what might happen if the first library JAR is omitted from the class path (or placed later in the class path than the second JAR). In this case, the Main class searches the class path for the `Util` class it needs, but the first `Util` class it finds is not the one it was designed for. This is a very simple example, but it is always a potential problem as applications are enhanced and as further libraries that support the enhancements are added to the class path.

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)

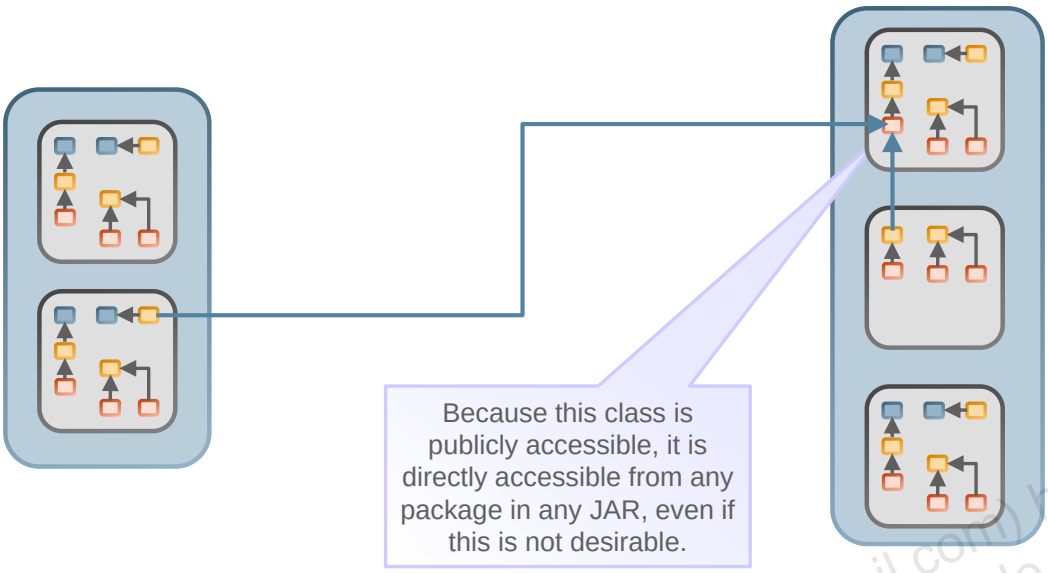
- `public` to everyone
- `public`, but only to specific modules
- `public` only within a module
- `protected`
- <package>
- `private`

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

By default, modules cannot access each other. They can't instantiate each other's classes. They can't call each other's methods. This is true even if classes, fields, and methods use the access modifier. In a sense, "public" no longer has the same definition after JDK 9.

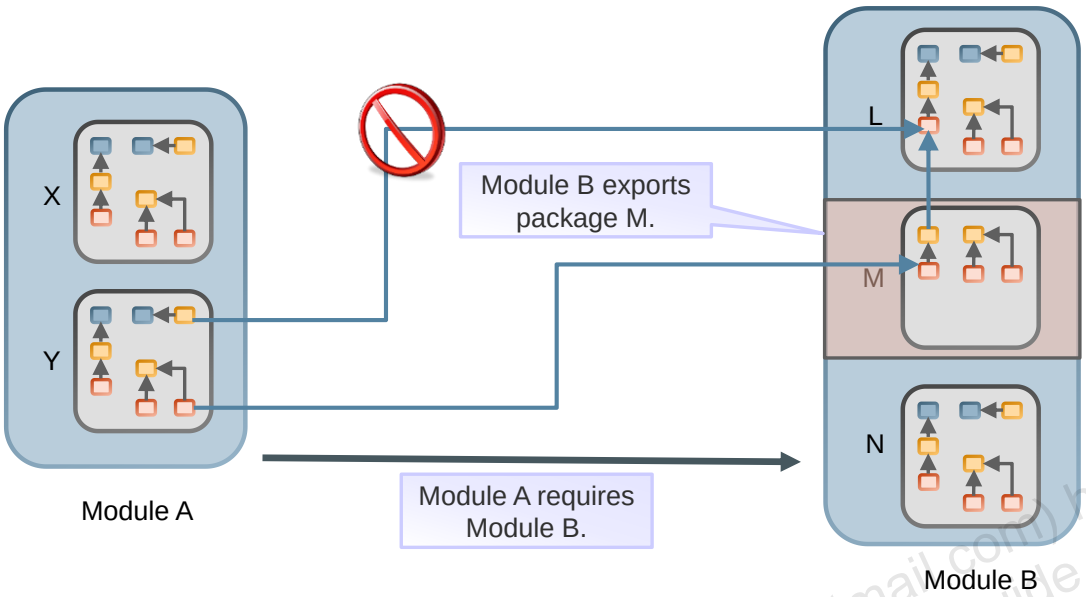
Readability is a new term used to describe the relationship between modules. Making one module readable by another is a multi-step process. You must edit each module's `module-info` class to specify how readability should occur. A module declares which of its packages can be readable by other modules. A module declares what other modules it is required to read from.

Access Across Non-Modular JARs



If a class needs to be made accessible to a different package in the same JAR, it must be made public. This makes it also accessible to any class in any package.

Access Across Modules



In the example shown, the class in package Y cannot access the class in package L, even though module A requires module B (assuming the class in L is public). Access is based on the following:

- Modules must explicitly require other modules. This gives the module system reliable dependencies. These are checked during compilation and before running the code.
- Modules must explicitly export the packages they want to make visible. This delivers encapsulation at the module level.

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.

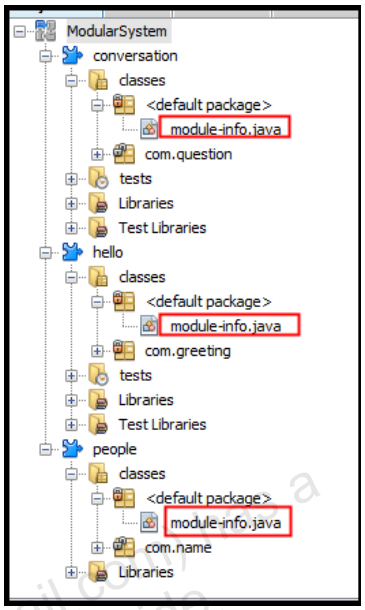
Example: `module-info.java`

```
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.



The image in the slide shows several modules. Each module contains its own module-info class. Real-world applications may contain many modules. This allows modules to be reused and swapped between programs.

This example modular project, ModularSystem can be accessed in the lab environment in /home/oracle/labs/16_ModularSystem/practices

`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;
}
```

The Effects of Exporting:

Everything in the com.question package is eligible to be read by another module.

Nothing with,, or package access is eligible.

Use the statement judiciously.

- You'll never be certain how someone else wants to use your module.
- Export only packages that you feel are safe.
- Modularity is about future-proofing.

`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;
}
```

The Effects of Requiring

Everything in the `game` package is now readable by the `main` module.

The `competition` module provides an API to the `main` module.

Classes in the `main` module can now:

- Instantiate object types defined in the `game` package
- Call methods of classes in the `game` package

Nothing with,, or package access is readable.

Nothing in the `util` package is readable.

`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;
}
```

Although the `hello` module doesn't explicitly require `conversation`, classes within this module can still read. Transitivity allows for readability up the requirement chain. `hello` requires people. people requires conversation transitively.

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.

For example, complications won't arise if your code has a variable called `export` because these keywords are restricted to 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 !

Before the introduction of modules java runtime, search for classes not by analysing packaged module dependencies, but rather by scanning classpath. Classpath could be set on a system level as well as for a specific application, and sometimes this could have resulted in applications braking at runtime. For example, if you have installed a Java application that used a particular API and then you install another application that used a newer version of the same API. Executing these applications with common classpath definition, it would potentially result in NoClassDefFound or NoSuchMethod errors depending on the order in which different versions of the same API occurred in your classpath. When designing modules, you may include explicit version identity as part of the module name and avoid such errors.

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.xml@9.0.1	jdk.hotspot.agent@9.0.1	jdk.management.jfr@9.0.1
java.base@9.0.1	java.xml.bind@9.0.1	jdk.httpserver@9.0.1	jdk.management.resource@9
java.compiler@9.0.1	java.xml.crypto@9.0.1	jdk.incubator.httpclient@9.0.1	jdk.naming.dns@9.0.1
java.corba@9.0.1	java.xml.ws@9.0.1	jdk.internal.ed@9.0.1	jdk.naming.rmi@9.0.1
java.datatransfer@9.0.1	java.xml.ws.annotation@9.0.1	jdk.internal.jvmstat@9.0.1	jdk.net@9.0.1
java.desktop@9.0.1	javafx.base@9.0.1	jdk.internal.le@9.0.1	jdk.pack@9.0.1
java.instrument@9.0.1	javafx.controls@9.0.1	jdk.internal.opt@9.0.1	jdk.packager@9.0.1
java.jnlp@9.0.1	javafx.deploy@9.0.1	jdk.internal.vm.ci@9.0.1	jdk.packager.services@9.0
java.logging@9.0.1	javafx.fxml@9.0.1	jdk.jartool@9.0.1	jdk.plugin@9.0.1
java.management@9.0.1	javafx.graphics@9.0.1	jdk.javadoc@9.0.1	jdk.plugin.dom@9.0.1
java.management.rmi@9.0.1	javafx.media@9.0.1	jdk.javaws@9.0.1	jdk.plugin.server@9.0.1
java.naming@9.0.1	javafx.swing@9.0.1	jdk.jcmd@9.0.1	jdk.policytool@9.0.1
java.prefs@9.0.1	javafx.web@9.0.1	jdk.jconsole@9.0.1	jdk.rmic@9.0.1
java.rmi@9.0.1	jdk.accessibility@9.0.1	jdk.jdeps@9.0.1	jdk.scripting.nashorn@9.0.1
java.scripting@9.0.1	jdk.attach@9.0.1	jdk.jdi@9.0.1	jdk.scripting.nashorn.shell@9.0.1
java.se@9.0.1	jdk.charsets@9.0.1	jdk.jdwp.agent@9.0.1	jdk.sctp@9.0.1
java.se.ee@9.0.1	jdk.compiler@9.0.1	jdk.jfr@9.0.1	jdk.security.auth@9.0.1
java.security.jgss@9.0.1	jdk.crypto.cryptoki@9.0.1	jdk.jlink@9.0.1	jdk.security.jgss@9.0.1
java.security.sasl@9.0.1	jdk.crypto.ec@9.0.1	jdk.jshell@9.0.1	jdk.snmp@9.0.1
java.smartcardio@9.0.1	jdk.crypto.mscapi@9.0.1	jdk.jsobject@9.0.1	jdk.unsupported@9.0.1
java.sql@9.0.1	jdk.deploy@9.0.1	jdk.jstatd@9.0.1	jdk.xml.bind@9.0.1
java.sql.rowset@9.0.1	jdk.deploy.controlpanel@9.0.1	jdk.localedata@9.0.1	jdk.xml.dom@9.0.1
java.transaction@9.0.1	jdk.dynalink@9.0.1	jdk.management@9.0.1	jdk.xml.ws@9.0.1
		jdk.management.agent@9.0.	jdk.zipfs@9.0.1
		jdk.management.cmm@9.0.1	oracle.desktop@9.0.1
			oracle.net@9.0.1

Using the java command from the JDK's bin folder with the `--list-modules` option, as in:

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

Lists the JDK's set of modules, which includes the standard modules that implement the Java Language SE Specification (names starting with java), JavaFX modules (names starting with javafx), JDK-specific modules (names starting with jdk), and Oracle-specific modules (names starting with oracle).

Each module name is followed by a version string—@9 indicates that the module belongs to Java 9.

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