10

The Module System

Objectives

After completing this lesson, you should be able to:

- Describe the purpose (in general terms) of the `module-info` class
- Create modules with defined module dependencies and module encapsulation
- Compile modules and create modular JAR files on the command line
- Describe how NetBeans IDE organizes its folders for source, compiled modules, and modular JARs

Module System

- The module system:
  - Supports programming in the “large”
  - Is built into the Java language
  - 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

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 all developers; it will be done by architects, and some developers will program only in the “small.”

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
    - Which packages it makes available to other modules
      - Only the public interfaces in those available packages are visible outside the module.

Java Modular Applications

- No missing dependencies
- No cyclic dependencies
- No split packages

Dependencies are fully checked during compilation and run time. The program will not attempt to run if all dependencies are not found.

Cyclic dependencies—module A requires module B and vice versa—are not permitted. This ensures more robust applications.

Split packages—module A and module B contain packages with the same names—are not permitted.

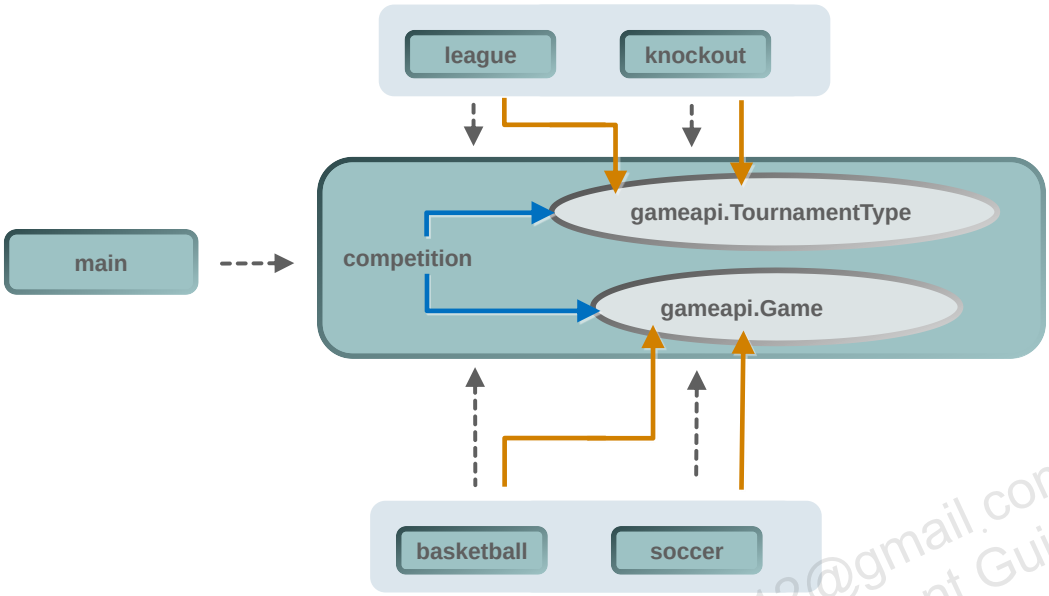
All these are explored further later in this lesson.

What Is a Module?

A module is a set of packages that make sense being grouped together.

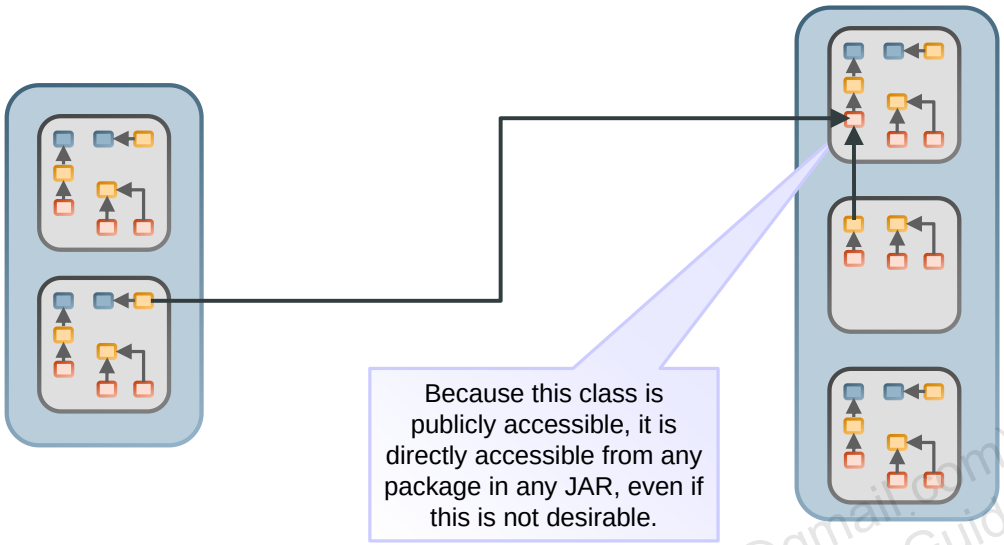
- Modularity was introduced in JDK SE 9.
- Modules add a higher level of aggregation above packages.
  - They are the basic unit of distribution and reuse.
- A module is a reusable group of related packages, as well as resources (such as images and XML files) and a module descriptor; that is, **programs 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.

A Modular Java Application



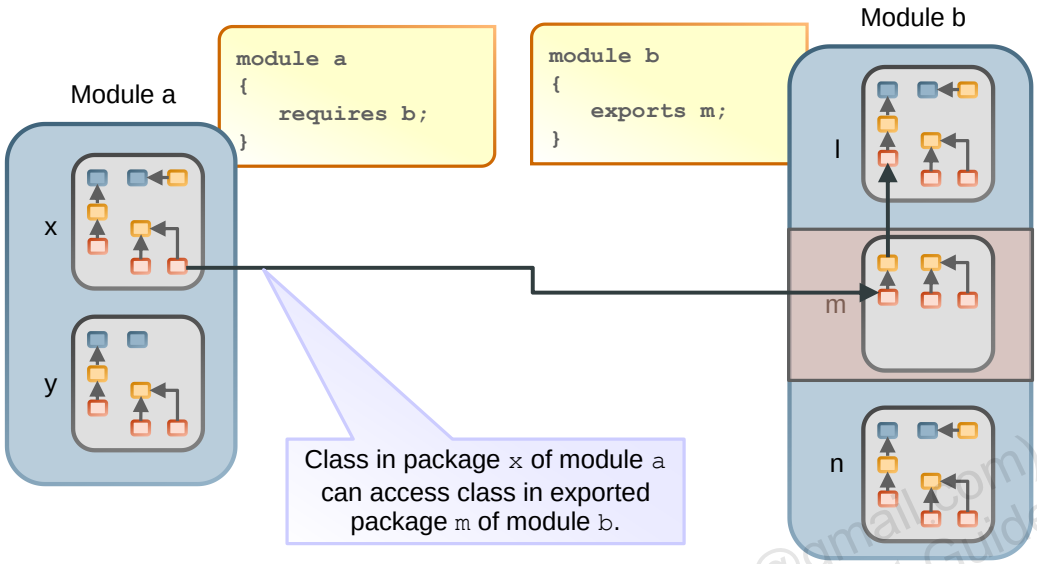
The slide shows a dependency diagram for the TeamGameManager application. The types of games supported can easily be added to it, as can the types of competition.

Issues with Access Across Nonmodular 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.

Dependencies Across Modules

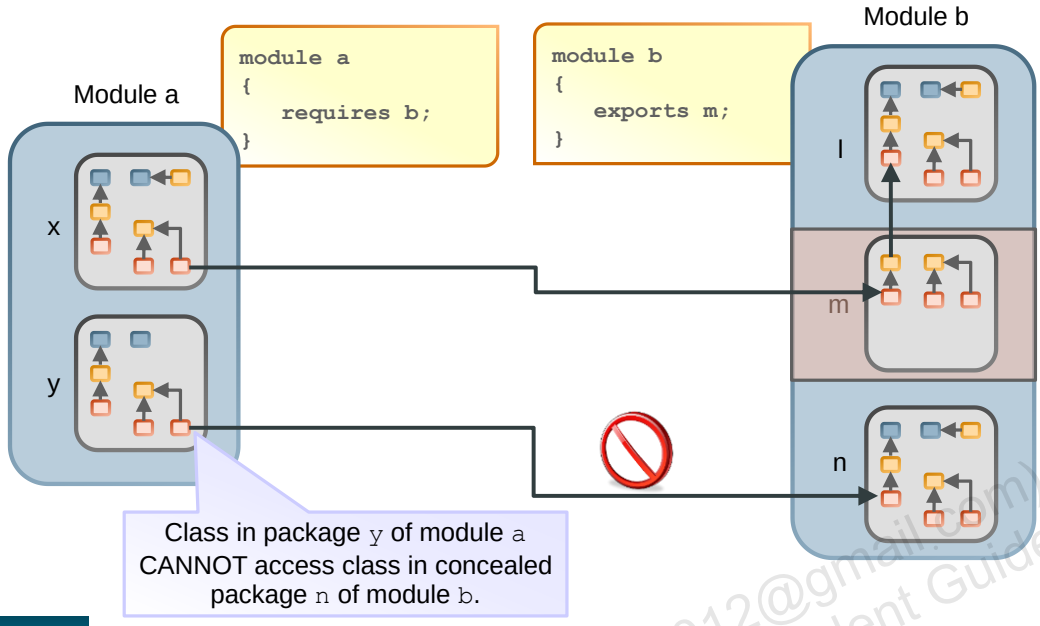


In the example shown, module `a` has a dependency on module `b`; it "requires" module `b`. But module `b` only makes its `m` package available (it "exports" package m).

The class in package `x` can access the class in package `m`, because `m` is exported, but it cannot access any class in packages `l` or `n`, even if these are public classes. Classes in modules `l` or `n` are concealed packages only accessible (if public) within module `b`.

Note that the `requires` and `exports` keywords are introduced here just to illustrate that modules offer explicit dependencies and encapsulation at the module level. They, and other module-related directives, will be covered in detail later in this lesson.

Dependencies Across Modules



Note that the class in package `y` cannot access the class in package `n` (assuming the class in `L` is public), even though module a requires module `b`. Access across modules 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.
- The class being accessed must be public.

What Is a Module?

A module:

- Contains one or more packages and other resources such as images or xml files
- Is defined in its module descriptor (`module-info.class`), which is stored in the module's root folder
  - The module descriptor must contain the module name.
  - Additionally the module descriptor can contain details of:
    - Required module dependencies (other modules that this module depends on)
    - Packages that this module exports, making them available to other modules
      - Otherwise all packages in the module are implicitly unavailable to other modules.
    - Permissions to open content of this module to other modules via the use of reflection
    - Services this module offers to other modules
    - Services this module consumes


```
module <this module name> {
  requires <another module name>
  exports <packages of this module to other modules that require any
           public types they contain>
  opens <packages, including non-public types, to other modules>
  uses <services provided via a an implementation>
  provides <services to any module> with <a service implementation>
  version <value>
}
```

Module Dependencies with `requires`

A module defines that it needs another module using the `requires` directive.

- `requires` specifies a normal module dependency (this module needs access to some content provided by another module).
- `requires transitive` specifies a module dependency and makes the module dependent upon available to other modules.
- `requires static` indicates module dependency at compile time, but not at the run time.

requires—module a is dependent on module b; it needs at least one class or interface in b.

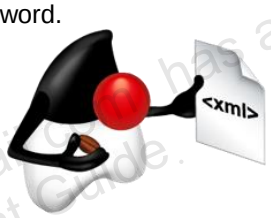
requires transitive—module a is dependent on module b; it needs at least one class or interface in b, AND it will make that class or interface in b available to other modules.

This is a very brief summary—it's covered in much more details later.

Module Package Availability with `exports`

A module defines what content it makes available for other modules using the `exports` directive.

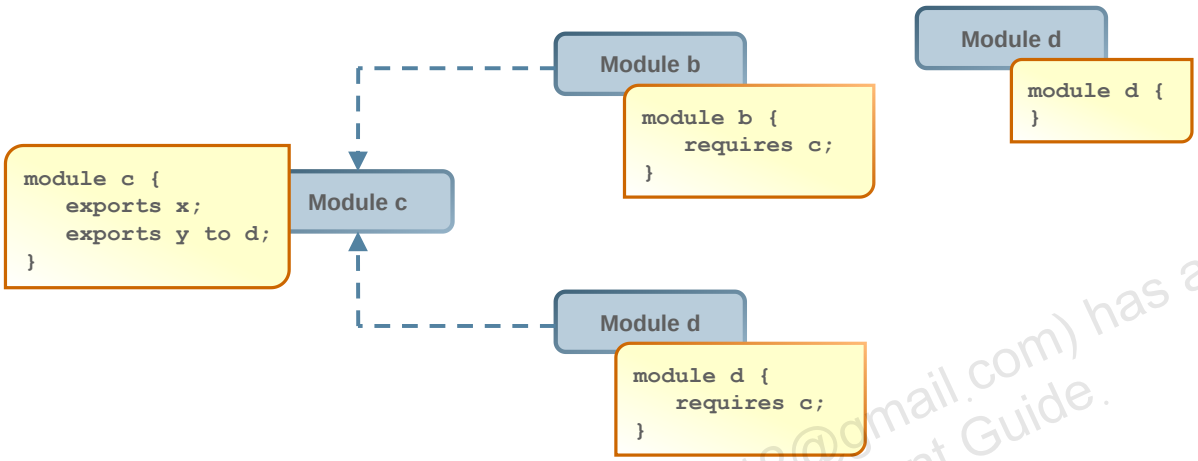
- Exporting a package makes all of its public types available to other modules.
- There are two directives to specify packages to export:
  - The `exports` directive specifies a package whose public types are accessible to all other modules.
  - The `exports ... to` directive restricts the availability of an exported package to a list of specific modules.
    - Accepts a comma-separated list of module names after the `to` keyword.



- Only classes (and their variables and methods) that have public access can be accessed from another module.
- To access types in an exported package, the module requiring the use of those types must use the `requires` directive.

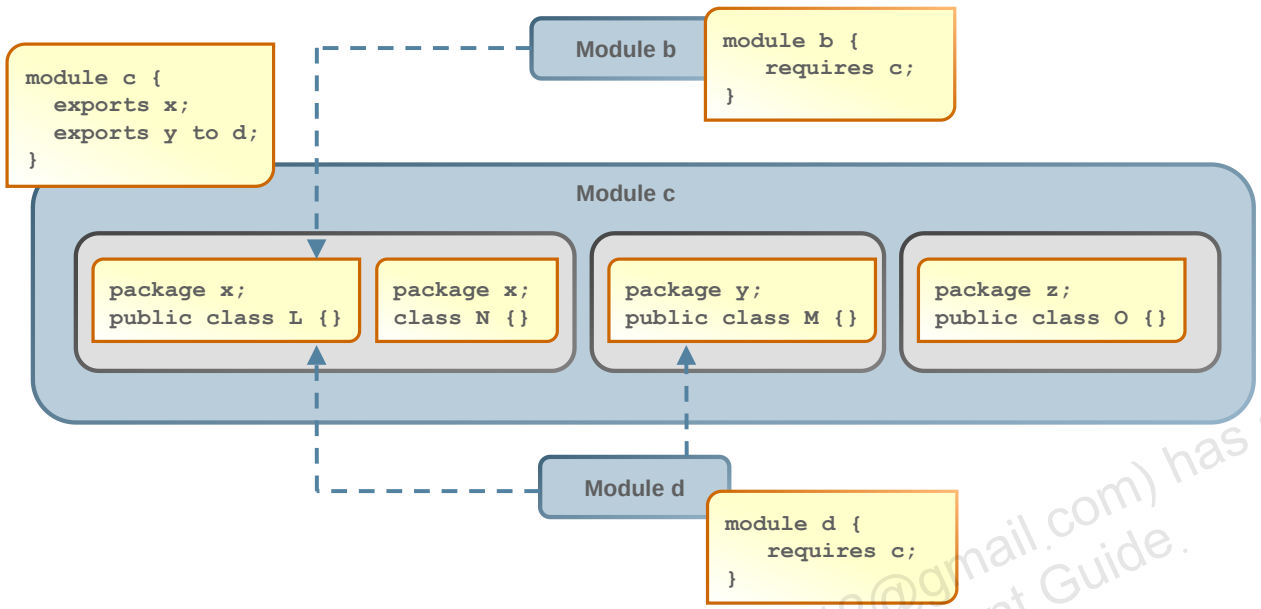
Module Graph 1

- The module system resolves all dependencies expressed in the `requires` directives.
  - This can be illustrated as a module graph.



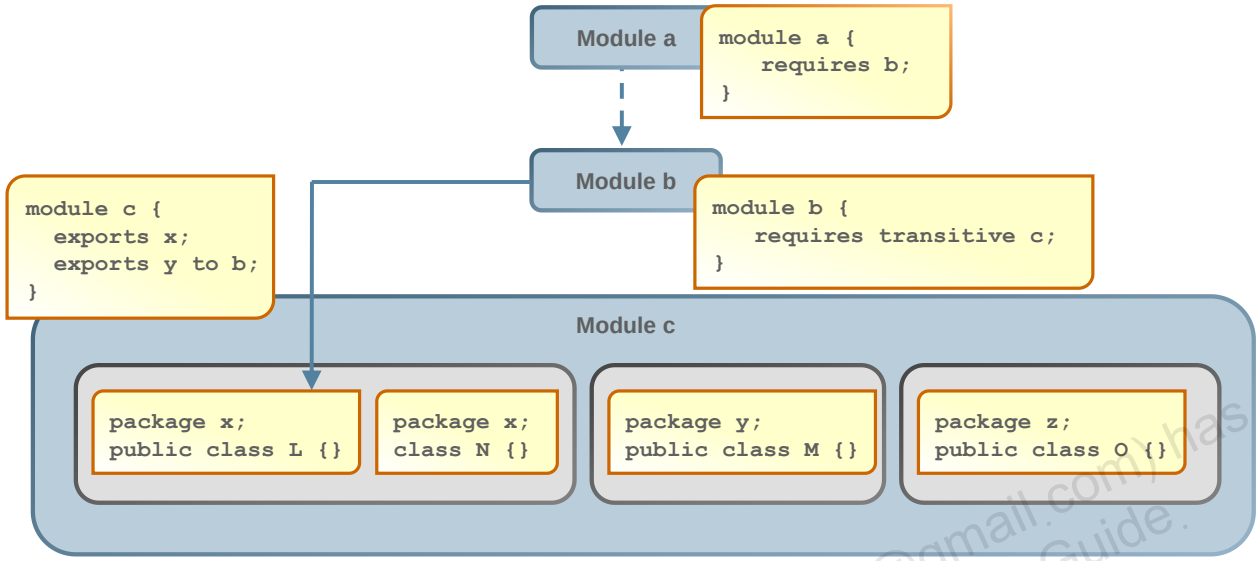
The module system *resolves* the dependencies expressed in its **requires** clauses by locating additional observable modules to fulfill those dependencies and then resolves the dependencies of those modules, and so forth, until every dependency of every module is fulfilled. The result of this transitive-closure computation is a *module graph* that, for each module with a dependency that is fulfilled by some other module, contains a directed edge from the first module to the second.

Module Graph 2



- In the example above, module `c` exports package `x` to all other interested modules and package `y` just to module `c`.
- Thus:
  - Class `L` is visible to modules `b and d`.
  - Class `N` is not visible to modules `b` and `d` because it is not public.
  - Class `M` is not visible to module b because its package has only been exposed to module `d`.
  - Class `O` is not visible to anyone outside of the module `c` because package `z` is not exported.

Transitive Dependencies



Imagine that module `b` provides an API and for this reason is required by module `a`. In addition, imagine that the API returns a type `L` that exists in package `x` in module `c`. In order for module a to have access to this type, `L`, either:

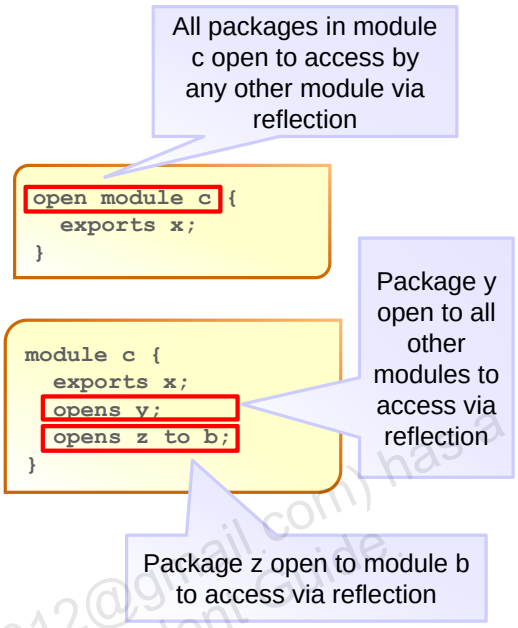
- Module `b` must transitively require c (as shown here) OR
- Module `a` must state its own requirement for module `c`

In the example shown, module `b` has a transitive dependency on module `c`. Note that dependencies between modules are shown with a dotted line and transitive dependencies with a solid line.

Access to Types via Reflection

A module may be set up to allow runtime-only access to a package by using the opens directive.

- The `opens` directive makes a package available to all other modules at run-time but not at compile time.
- The `opens ... to` directive makes a package. available to a list of specific modules at run-time but not compile time.
- Using `opens` for a package is similar to using `exports`, but it also makes all of its nonpublic types available via reflection.
  - Modules that contain injectable code should use the opens directive, because injections work via reflection.
- All packages in a module can be made available to access via reflection by using the `open` directive before the module directive.



Before Java 9, reflection could be used to learn about all types in a package and all members of a type—even its private members—whether you wanted to allow this capability or not. Thus, nothing was truly encapsulated.

A key motivation of the module system is strong encapsulation. By default, a type in a module is not accessible to other modules unless it's a public type and you export its package. You expose only the packages you want to expose. With Java 9 and later, this also applies to reflection.

The `opens` directive allows you to specify a package so that all its types (and all of its types' members) are accessible via reflection (this includes types with nonpublic access).

The `opens… to` directive is the same as the `opens` directive except it allows you to limit the access by reflection to a set of specified modules.

Note that if the types are public, `exports <package name>` makes all the types in a package available via reflection, but `opens` is required to make nonpublic types available.

Opening packages to access via reflection is covered in more details in the lesson titled "Migration.".

Example Hello World Modular Application Code

Here is a simple Hello World application with two modules. It will be used for examples later in this lesson.

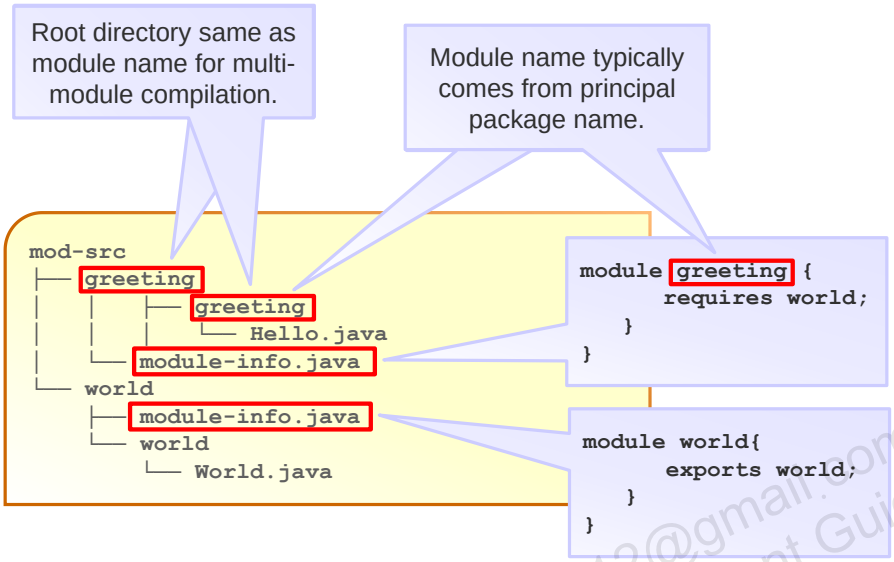
Module `greeting`

```
package greeting;
import java.util.logging.Logger;
import world.World;
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello " +
         World.say());
   }
}
```

Module `world`

```
package world;
public class World {
   public static String say() {
      return "World!";
   }
}
```

Example Hello World Modular File Structure



Compiling a Modular Application

Single module compilation:

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

Multi-module compilation:

```
javac -d <output folder>
   --module-source-path <root directory of the module source> \
   <list of source code file paths>
```

Get description of the compiled module:

```
java --module-path <path to compiled module>
     --describe-module <module name>
```

When compiling a single module, all source files must be listed, including the module-info.java file. The output folder needs to be set to the directory you wish the modules to be placed in. A multi-module application can be compiled by using a sequence of single module compilations, but this is not ideal as it requires a number of commands, and it cannot guarantee the dependencies.

Multi-module compilation is achieved by using the `--module-source-path` option to point to the root directory of the source file structure.

You can get a description of a compiled module by using the java command with `--module-path` and `--describe-module` options, with `--module-path` pointing to either the module folder or to the containing folder for the module folder. If the module being described has not explicitly defined that it requires `java.base` and relied upon implicit inclusion, then `--describe-module` displays:

```
java.base mandated for this module.
```

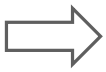
Single Module Compilation Example

You can compile the application module by module.

Output folder points to module folder.

```
javac -d mods/world src/world/module-info.java \
src/world/world/World.java
javac -d mods/greeting --module-path mods src/greeting/module-info.java \
src/greeting/greeting/Hello.java
```

```
mod-src
├── greeting
│   │   ├── greeting
│   │   │   └── Hello.java
│   └── module-info.java
└── world
    ├── module-info.java
    └── world
        └── World.java
```



```
mods
├── greeting
│   │   ├── greeting
│   │   │   └── Hello.class
│   └── module-info.class
└── world
    ├── module-info.class
    └── world
        └── World.class
```


In the example above:

- For each module to be compiled, the `module-info.java` file must be included in the list of source files.
- `world` is compiled first as it has no dependencies on any other module.
- The second module to be compiled must use `--module-path` to point to the compiled `world` module.
- `-d` (the output folder specification) has the name of the module to be compiled as the name of the final directory in its path. This is not mandatory for single module compilation, but it is recommended.
- The module name is based on the package name. It is recommended as good practice to name the module the same as the principal package in the module, but this is not mandatory.

Multi Module Compilation Example

- Passing just the filename for the source of the main class.

```
javac -d mods --module-source-path src src/greeting/greeting/Hello.java
```

Output folder points to containing folder for modules.

- Passing all source filenames.

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

- Checking module contents.

```
java --module-path mods --describe-module greeting
greeting file:///home/<username>/demo/mods/greeting
exports greeting
requires java.base mandated
requires world
```


The `javac` examples above use `--module-source-path` so `javac` can find the source code.

In the first example, even though `javac` is only passed `Hello.java` as a file to compile, it can determine that modular compilation is required. This means that:

- The `greeting` module will be compiled (`Hello.java` and the `greeting` module's `module-info.java` file).
- The `world` module will be compiled (`World.java` and the `world` module's `module-info.java` file).
- For the first compilation (assume `mods` directory does not exist), all necessary files will be compiled.
- For compilation where the compiled files already exist in the destination directory, only `Hello.java` will be compiled.

In the second example, the unix `find` command is used to ensure that all source files are compiled.

Creating a Modular JAR

- Use the `jar` command to create a modular JAR:

```
jar --create -f <path and name of JAR file>
    --main-class <package name>.<main class name>
    -C <path to compiled module code> .
```

- Hello World application example:

```
jar --create -f jars/world.jar -C mods/world .
jar --create -f jars/hello.jar --main-class greeting.Hello -C mods/greeting/ .
```

The `jar` command options shown are:

- `--create` instructs the jar utility to create new jar file.
- `-f` sets path and name of the JAR file.
- `-C` sets path to compiled code of the module.
- `--main-class` sets the main class of the JAR, so it doesn't need to be passed to the `java` command on the command line.

Running a Modular Application

- Running an unpackaged module application:

```
java --module-path <path to compiled module or modules> \
     --module <module name>/<package name>.<main class name>
```

- Running an application packaged into modular JARs (assuming main class specified when creating JARs):

```
java --module-path <path to JARs> --module <module name>
```

- Running the Hello World application example:

```
java -p jars -m greeting
```

Some options have short versions:

- `--module-path` has a short version, `-p`.
- `--module` has a short version, `-m`.

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

Java SE Modules

`java.se`:

- This module doesn't contain any code but has only dependencies declared in the module descriptor:

```
module java.se {
    requires transitive java.desktop;
    requires transitive java.sql;
    requires transitive java.xml;
    requires transitive java.prefs;
    // .. many more
}
```

  - In the module descriptor, a `requires transitive` clause is listed for each module that is part of the Java SE specification.
  - When you say requires `java.se` in a module, all these modules will be available.

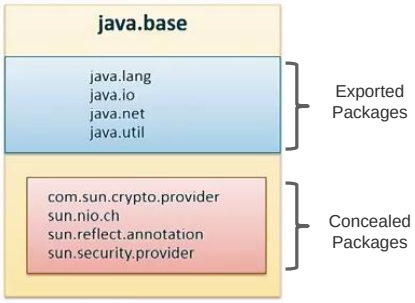
Java SE Modules

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.
  - 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;
}
```

Finding the Right Platform Module

You can get a list of the packages a platform module contains with the `--describe-module` switch:

```
/home/oracle$ java --describe-module java.base
```

Partial Output is shown:

```
exports java.io
exports java.lang
exports java.lang.annotation
exports java.lang.invoke
exports java.lang.module
exports java.lang.ref
exports java.lang.reflect
exports java.math
exports java.net
……..
```

1

1 The `java.base` module exports the `java.math` package.

Illegal Access to JDK Internals in JDK 9

- You can disable the warning message on a library-by-library basis by using the `--add-opens` command-line flag.
- For example, you can start Jython in the following way:

```
$java --add-opens  java.base/sun.nio.ch=ALL-UNNAMED --add-opens
java.base/java.io=ALL-UNNAMED -jar jython-standalone-2.7.0.jar
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
```

  - This time the warning is not issued because the Java invocation explicitly acknowledges the reflective access.
- As you can see, you may need to specify multiple `--add-opens` flags to cover all the reflective access operations that are attempted by libraries on the class path.

To better understand the behavior of tools and libraries, you can use the --illegal-access=warn command line flag. This flag causes a warning message to be issued for every illegal reflective-access operation. In addition, you can obtain detailed information about illegal reflective-access operations, including stack traces.

What Is a Custom Runtime Image?

- You can create a special distribution of the Java run time containing only the necessary modules.
  - Application modules and only those platform modules used by your application
- You can do this in Java SE 9 with *custom runtime image*.
- A custom runtime image is self-contained:
  - It bundles the application modules and platform modules with the JVM and everything else it needs to execute your application.

Link Time

- In Java SE 9, an optional **link time** is introduced between the compilation and runtime phase.
- Link time:
  - Requires a linking tool that will assemble and optimize a set of modules and their transitive dependencies to create a runtime image.

Using `jlink` to Create a Runtime Image

- A basic invocation of `jlink`:

```
jlink [options] --module-path modulepath --add-modules mods
--output path
```

- You will have to specify the following three parameters:
  - `modulepath`: The module path where the platform and application modules to be added to the image are located. Modules can be modular JAR files, JMOD files, or exploded directories.
  - `mods`: The list of the modules to be added to the runtime image. The `jlink` tool adds these modules and their transitive dependencies.
  - `path`: The output directory where the generated runtime image will be stored.

JMOD format:
JDK 9 introduced a new format, called JMOD, to package modules. JMOD files are designed to handle more content types than JAR files can. The JDK 9 modules are packaged in JMOD format for you to use at compile time and link time. JMOD format is not supported at run time. You can package your own modules in JMOD format. Files in the JMOD format have a .jmod extension. JDK 9 ships with a new tool called `jmod`. It is located in the `JDK_HOME\bin` directory. It can be used to create a JMOD file, list the contents of a JMOD file, and print the description of a module.

Example: Using `jlink` to Create a Runtime Image

The following command creates a new runtime image:

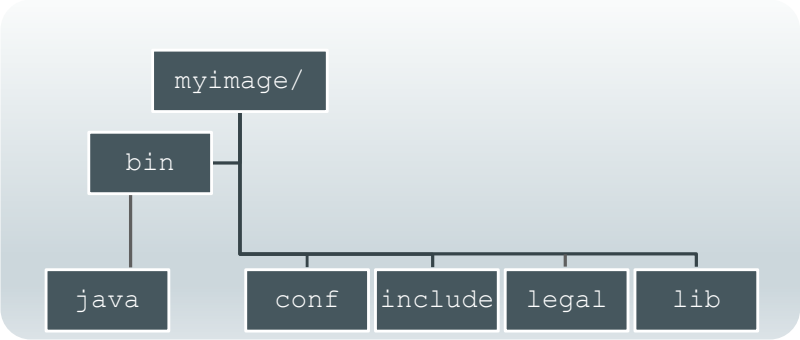
```
/Hello$ jlink
 --module-path dist/Hello.jar:/usr/java/jdk-9/jmods
 --add-modules com.greeting
 --output myimage
```

- `--module-path`: This constructs a module path where `HelloWorldApp` is present, and the `$JAVA_HOME/jmods` directory contains the platform modules.
- `--add-modules`: This indicates that `com.greeting` is the module that needs to be added in the runtime image.
- `--output`: This directory is where the runtime image generated, `myimage`, is stored.

Note: In Windows, the path separator is ; instead of :

Examining the Generated Image

The generated image, `myimage`, has the following directory layout:



The custom run time generated is:

- Fully self-contained. It bundles the application modules with the JVM and everything else it needs to execute your application.
- Platform-specific and is not portable to other platforms.

bin: Contains executable files. On Windows, it also contains dynamically linked native libraries (.dll files).

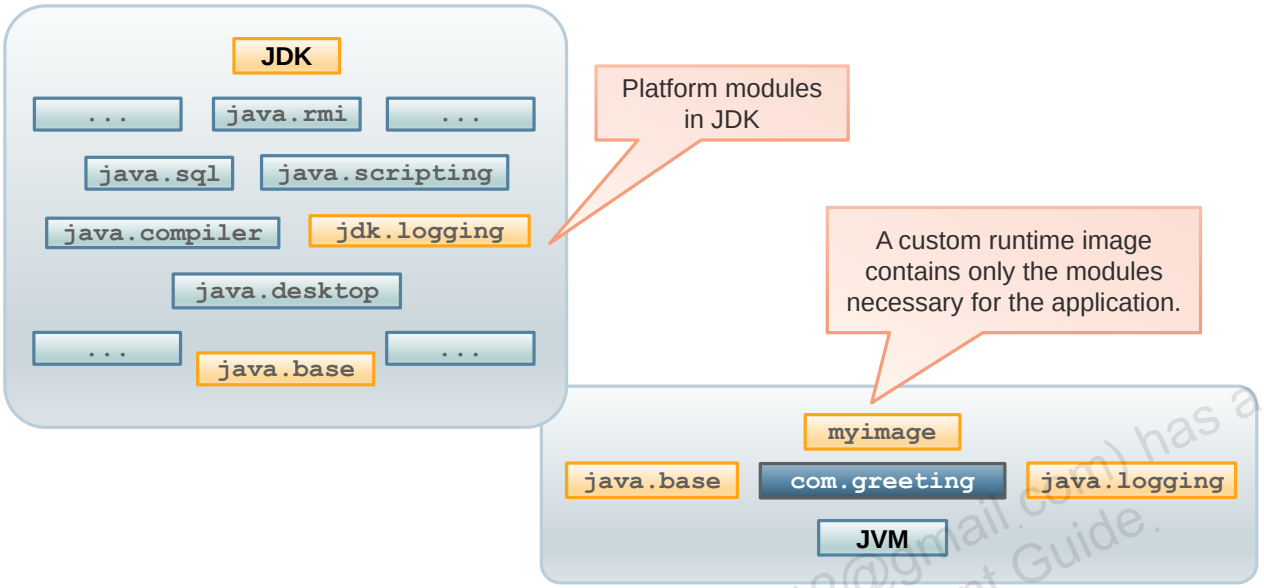
conf: Contains the editable configuration files such as .properties and .policy

include: Contains C/C++ header files

legal: Contains legal notices

lib: Contains, among other files, the modules added to the runtime image

Modules Resolved in a Custom Runtime Image



Advantages of a Custom Runtime Image

Creating a custom runtime image is beneficial for several reasons:

- Ease of use:
  - Can be shipped to your application users who don't have to download and install JRE separately to run the application.
- Reduced footprint:
  - Consists of only those modules that your application uses and therefore is much smaller than a full JDK
  - Can be used on resource-constrained devices or to run an application in the cloud
- Performance:
  - Runs faster because of link time optimizations that are otherwise too costly.

JIMAGE Format

- The runtime image is stored in a special format called JIMAGE, which is:
  - Optimized for space and speed
  - A much faster way to search and load classes than from JAR and JMOD file
- JDK 9 ships with the `jimage` tool to let you explore the contents of a JIMAGE file.

Running the Application

- You can use the `java` command, which is in `myimage`, to launch your application.

```
$ myimage/bin/ java --module com.greeting
```

```
$ myimage/bin/ java -m  com.greeting
```

Name of the module

- You don't have to set the module path. The custom runtime image is in its own module path.

Running the Application

The `jlink` command has a `–launcher` option that creates a platform-specific executable in the `bin` directory.

```
/Hello$ jlink
 --module-path dist/Hello.jar:/usr/java/jdk-9/jmods
 --add-modules com.greeting
 --launcher Hello=com.greeting
 --output myimage
```

Name of the module

You can use this executable to run your application:

```
$ myimage/bin Hello
```

The `--launcher` option makes `jlink` create a platform-specific executable such as a `Hello.bat` file on Windows in the bin directory. You can use this executable to run your application. The file contents are simply a wrapper for running the main class in this module. You can use this file to run the application.

Summary

In this lesson, you should have learned how to:

- Describe the purpose (in general terms) of the module-info class
- Create modules with defined module dependencies and module encapsulation
- Compile modules and create modular JAR files on the command line
- Describe how NetBeans IDE organizes its folders for source, compiled modules, and modular JARs

Practice 10: Overview

This practice covers the following topics:

- 10-1: Creating a modular application from the Command Line
- 10-2: Compiling modules from the Command Line
- 10-3: Creating a modular application from NetBeans
- 10-4: Requiring a module transitively
- 10-5: Beginning to modularize an older Java application
- 10-6: Creating and Optimizing a Custom Runtime Image by Using `jlink`
- 10-7: Using NetBeans to Create and Optimize a Runtime Image