Each module, what we called a part up to now, has clear responsibilities

and a well-defined contract it implements. It is self-contained, opaque to its clients, and can be replaced by a different module as long as that one

implements the same contract. Its few dependencies are APIs and not

implementations.

* A name, preferably one that is globally unique.
* Declarations of dependencies on other modules.
* A clearly defined API that consists of exported packages.

list of all modules contained in a JDK or JRE by running **java –list-modules**

To get details for a single module execute **java --describe-module**

**${module-name}**

**Tipos de Modulos:**

– Platform modules are packed into **JMOD** files

– modulos fuera del jdk son modular JARs , which are just plain JARs that contain a new construct, the module descriptor , which defines the module’s name, dependencies, and exports.

– modules that the module system creates on the fly from JARs that were not yet transformed into modules.

no matter how types and resources are presented to the compiler or the virtual machine, they will end up in a module

java --module-path mods --module modulo\_con\_clase\_main/clase\_main

--module-path: carpeta con los modulos

--module: Modulo con el main

dependencias entre clases predeterminan dependecias entre modulos

Each module has a module declaration , by convention this is a file module-info.java in the project’s root source folder. From this, the compiler creates a module descriptor, a module-info.class. When the compiled code gets packaged into a JAR, the descriptor must end up in its root folder for the module system to recognize and process it.

The name typically follows the package naming convention: It should be globally unique by reverting a domain name, it’s all lower case, and sections are separated by dots

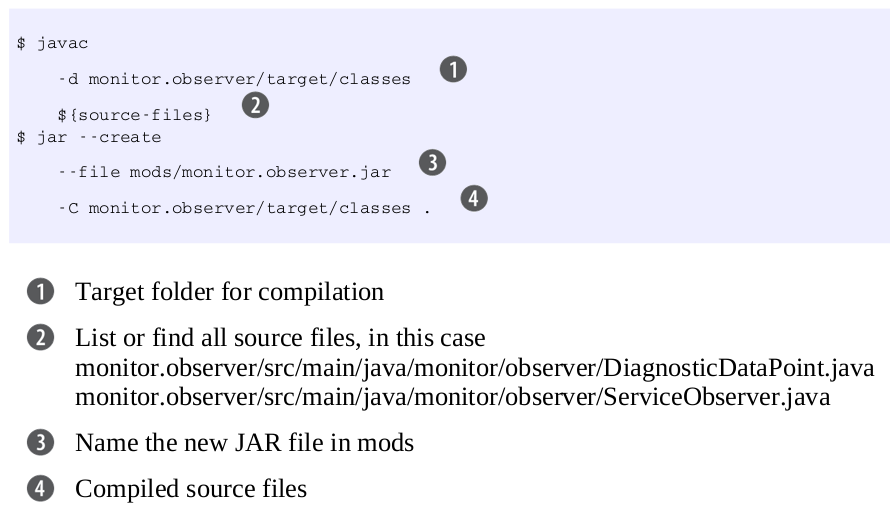
Inside the module block, requires and exports directives define dependencies and exported packages

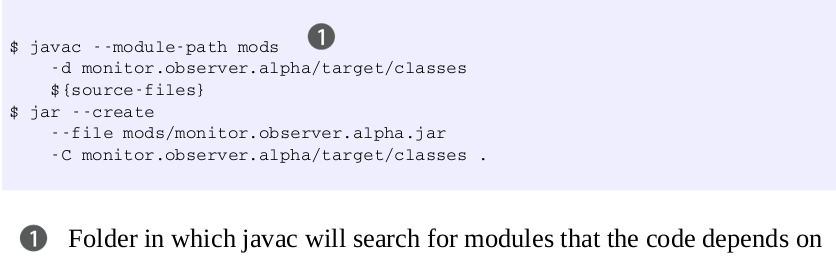
A requires directive contains a module name and tells the JVM that the declaring module depends on the one given by the directive.

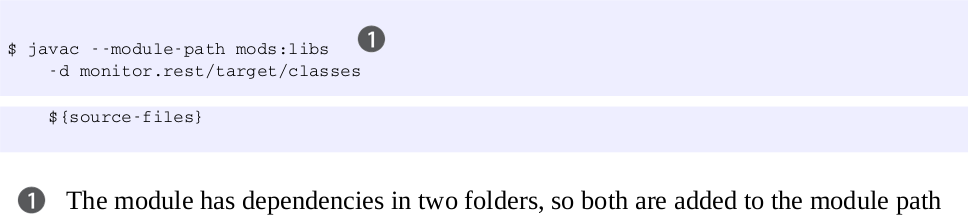
the module system forbids declaring cyclic dependencies.

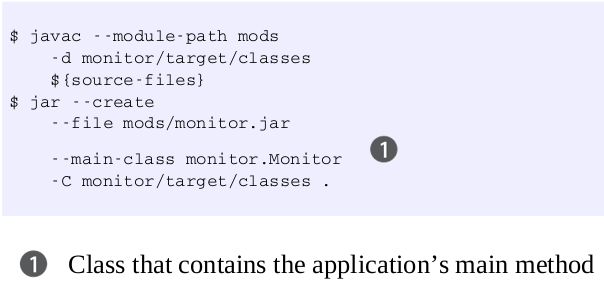
An exports directive contains a package name and informs the JVM that other modules depending on the declaring one can see public types in

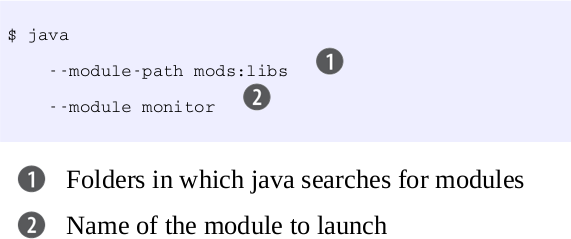
that package.











**goals of the module system:** configuracion confiable y fuerte encapsulamiento

- cuando hay dos definiciones de un modulo al ejecutar una aplicacion no se lanzara indicando el error

- si no hay un modulo requerido al lanzar la aplicacion no se ejecutará e indicara el error

- **Only types in the packages we export are visible and the rest are safe—even from reflection**

- Optional dependencies, which must be present at compile time but are not required at run time

- **open modules and open packages**. They make packages available at run time only. In exchange they allow reflection over private members as is often required by reflection-based tools. Similar to exports, there are also qualified opens, with which you can open a package to just a selected module.

- the module system offers services

--module-path or -p

**Tipos de modulos**

- modulos que desarrolladores crean para sus proyectos (librerias, frameworks, applicaciones) son **application modules** → modular jars, non JDK modulos

- el modulo que contiene el punto de ejecucion de la aplicacion (main) o donde la compilacion comienza es **initial module**

- **root modules** son donde el module system comienza resolviendo dependencias (el modulo que contiene el main es un root module, es parte de los observables)

- los modulos que construyen el jdk son **platform modules,** definidos por la Java SE Platform Specification (java.\*\*\*) y especificos del jdk (jdk.\*\*\*).

- un tipo especial de platform modules son los **incubator modules** (jdk.incubator.\*\*\*), contienen funcionalidades experimentales que se moveran a otro modulo cuando dejen de ser experimentales

- jlink puede incluir un subset de **platform modules** y **application modules**, cuando se genera la imagen son llamados en conjunto **system modules →** modulo contenido en una runtime image, sin el uso de jlink son igual al **platform modules**

- java --list-modules para para listar los modulos de la imagen

- todos los **platform modulos** en el actual runtime junto a los **application modules** especificados en la linea de comandos son llamados observables, juntos forman el universo de **observable modules**, los que el JPMS puede usar para completar las dependencias

- para el module system todos los modulos son lo mismo excepto java.base el **base module**

→ java.base contiene java.lang, java.util, java.nio, java.time…

→ java.base es el unico hard-coded en el module system

- **platform modules** y la mayoria de los **application modules** tienen module descriptors entregados por el creador del modulo y son llamados **explicit modules** (modulo con un descriptor)

- **automatic modules** son **application modules** pero creados por el runtime (modulos sin un module descriptor, plain jar en el module path, inferidos por el JPMS)

- **explicit modules** y **automatic modules** forman los **named modules**

- tambien hay **unnamed modules**: modulos sin nombre, son contenidos en el class path

el module system verifica que el conjunto de modulos observables contiene todas las requeridas dependencias, directas y transitivas, si ocurre un error lo reporta

→ no tienen que existir ambiguedad, no puede existir ambiguedad, 2 artefactos no pueden decir que tienen el mismo nombre

→ no tienen que existir dependencias ciclicas entre modulos, causara error en compilacion

→ no tienen que ser dependientes en compilacion

→ split package no seran compilados o lanzados, no importa si son o no exportados

→ lanzar una aplicacion con una dependencia transitiva perdida no funcionará

→ no pueden existir modulos duplicados en el class path

→ no hay concepto de version en modulos

→ se lanzara una exception si existe un modulo repetido AUNQUE no se use

un tipo foo en un modulo bar es **accesible** si:

→ foo es publico

→ foo pertenece a un modulo que bar exporta

→ otro modulo lee el modulo bar ← se incluye el modulo que quiere acceder al tipo, es solo accesible al modulo que quiera usarlo, tiene que indicar que quiere usarlo para ser parte de la accesibilidad, aunque tambien es correcto decir que el modulo es accesible aunque no exista un modulo que lo requiera, es accesible porque es publico y exportado

reglas de accesibilidad de miembros (campo, metodo, clase anidada) se mantiene (publicos son completamente accesibles, protected solo en clases con herencia, …)

- no es necesario exportar el packete que contiene el main

**module public api:**

- todo lo que no puede ser cambiado provocando errores en el codigo que lo usa

→ nombres de los tipos publicos

→ nombres de los paquetes exportados

→ nombres y tipos de nombres en campos publicos y protegidos

→ nombres, tipos de argumentos, tipos de retorno, nombres de metodo publicos y protegidos (firma del metodo)

→ se pueden cambiar los cambos privados y visibles en el paquete, cuerpo de metodos publicos, los nombres de todo lo que otro codigo pueda usar para la compilacion no tiene que ser cambiado para no provocar errores de compilacion (nombre de tipos, firma de metodos publicos)

si un tipo no es accesible no es posible interactuar en ninguna forma que es **especifica a su tipo**: instanciarlo, acceder a sus campos, invocar metodos o usar clases anidadas, si es posible cuando se accede atravéz de un super tipo (como una interface que el tipo implementa)

→ no public implementaciones de una public interface pueden ser usadas atravez de una interface

→ refleccion tampoco puede sobrepasar los limites de los modulos

**qualified export:**

exports … to concrete.jpa.implementation

**abrir un paquete:**

– **opens** nombre\_paquete

→ en tiempo de compilacion el paquete es fuertemente encapsulado, no hay diferencia en ser encapsulado o no

→ en runtime son completamente accesibles, incluyendo no publicos clases, metodos o campos

→ NO es exportado, hay que agregar el export en la declaracion del modulo

→ marca el paquete como *diseñado* para ser usado por reflexion

→ puede ser abierto en forma cualificada: **opens** nombre\_paquete **to** modulo1,modulo2,modulo3

- modulos tienen la habilidad de pasar sus propias dependencias a módulos que dependen de ellos: **implied readability**

- ahora NO se puede acceder a clases que no son publicas en paquetes que se exportan, anteriormente se podía acceder por reflexión a clases con acceso package NO funciona usando modulos

- **qualified export**: exportar un paquete A un modulo

- el module path consiste en varias entradas (directorios o JARs), no hay check de ambigüedad cuando hay varias entradas con el mismo nombre de modulo, cada entrada tiene que tener un modulo solo una vez, cuando hay distintas entradas con el mismo nombre de modulo solo queda la primera encontrada (se ordenan por el nombre que fueron nombradas en el module path), el primero ensombrece a los que se repiten

- distintos JARs los módulos no son mezclados, cuando el module system escoge uno como origen, busca las clases en ese jar, nunca mira otros jars.

- si no hay error al construir el **module graph** durante el **module resolution** significa que los módulos con el nombre correcto están presentes

- si hay tipos de datos que no existen existirá error en etapas posteriores

el module system tambien ve un grafo de artefactos

- el module graph consiste de módulos (nodos) conectados de acuerdo a sus dependencias directas

- el grafo es construido durante el module resolution y esta disponible en runtime usando reflexion

- modulos que fueron considerados durante la resolución de módulos no estan disponibles en compilación ni ejecución

- con la opcion **--add-modules** a,b,c,d,e (javac y java) usa la lista de modulos a,b,c,d y las define como root modules

→ root modules forman el inicial set de modulos desde donde el module graph es construido resolviendo sus dependencias

→ permite agregar modulos que posteriormente fueran requeridos pero no necesarios para compilar (quedaron fuera de la resolucion de modulos)

→ tiene las opciones ALL-DEFAULT, ALL-SYSTEM, ALL-MODULE-PATH

→ las primeras 2 solo funcionan en runtime

→ con la ultima todos los modulos del module path terminan el module graph

- no es posible decirle al module system que una dependencia no sera satisfecha y que está bien así

- **--add-reads** nombre\_modulo=objetivos agrega limites de legibilidad desde nombre\_modulo a todos los modulos objetivos (separados por coma)

→ nombre\_modulo tiene acceso a todos los public types de los paquetes exportados por objetivos AUNQUE require no sea explicito en nombre\_modulo

→ si objetivos incluyen ALL\_UNNAMED nombre\_modulo puede leer contenido desde el classpath

– con add-modules quedan los módulos disponibles en le module graph (son accesibles por reflexión los modulos que lo permiten), pero no hay una relacion directa con algun modulo

– con add-reads crea una relacion entre los modulos

– cuando el modulo system resuelve todas las dependencias y contruye el module graph y establece **legibilidad** entre modulo se mantiene activo validando las reglas de **accesibilidad**

→ si las reglas se rompen genera error en runtime o tiempo de compilacion

– modulos vienen desde los disponibles de java = **platform modules**

– solo **java.base** es explicitamente conocido por el module system

– developers crean module jars (plain jars con module descriptor module-info.class) = **application modules**, incluyendo el modulo con el main que es el **initial module**

– el module descriptor se compila desde el module declaration (module-info.java) el cual define:

→ el nombre que tiene que ser unico globalmente

→ dependecias a modulos usando **require**

→ paquetes que hace públicos con **exports**

– **reliable configuration:** dependency declarations and limites de lectura que el module system genera para cada modulo

→ se asegura que todos los modules estan presentes una vez y no hay dependencias circulares

– limites de lectura y package exports son las bases para strong encapsulation

→ el module sistem se asegura que solo tipos publicos en exported packages son accesibles y solo a modulos que los leen

– limites de accesibilidad aplican tambien a reflection

– module path (--module-path) consiste en archivos y carpetas y hace JARs disponibles al module system el cual los representa como modulos.

– los application modules del module path y los platform modules que se encuentran contenidos en runtime construyen el universo de los observables modules

→ durante la resolucion los modulos son buscados en el universo, comenzando por el root module, asi todos los modulos requeridos tienen que estar en el module path o en runtime

– module resolution verifica que la configuracion es segura/confiable (todas las dependencias estan presentes, no hay ambigüedad)

– el resultado de la resolucion es el module graph (representacion dentro del module system de como vemos dependencias de artefactos)

– solo modulos dentro del module graph estan diponibles en runtime

**Reflexion**

- refleccion permite acceder a no public miembros y clases, con los modulos no funciona de la misma forma.

- el codigo compilado tiene los limites de package

- con los modulos la refleccion tiene las mismas reglas de accesibilidad que el codigo compilado

→ puede acceder a miembros publicos de clases publicas en paquetes exportados

→ solo aplica a codigo en explicit modules, si el codigo corre desde el classpath no aplica (no esta encapsulado)

- export no soporta deep reflection sobre non-private campos y metodos

- en linea de comandos **--add-opens** nombre\_modulo/nombre\_paquete = modulo\_que\_usa\_reflexion || ALL-UNNAMED

→ permite acceder desde modulo\_que\_usa\_reflexion a todos los tipos y miembros de nombre\_paquete en nombre\_modulo (otros modulos no)

→ con ALL-UNNAMED todo el codigo del class path (unnamed modules) pueden acceder a nombre\_paquete en nombre\_modulo

→ solo es un efecto en runtime, no afecta en compilacion

*javac soporta command lines argument files*

**opens vs export**

**=**

- hacen el contenido del paquete disponible sobre los limites del modulo

- tienen la variante **to** otros,modulos dando acceso solo a esos modulos

- tienen command lines options para java y javac para superar el fuerte encapsulamiento si es necesario (--add-exports / --add-opens)

**!=**

- los paquetes exportados dan acceso a los tipos y miembros publicos a los modulos que los importen en tiempo de compilacion

- paquetes abiertos entregan acceso a todos los tipos y miembros, pero no en tiempo de compilacion, solo en runtime

**abrir un modulo**

**open** nombre\_modulo {

...

}

- abre todos sus paquetes

- es equivalente a un opens por cada paquete

- el compilador NO permite que se agregen **opens** adicionales

- el sistema de modulos no limita la visibilidad, la accesibilidad está limitada

→ **AccessibleObject::setAccesible** si el acceso no es dado por el modulo reflejado, invocando un constructor o metodo, acceder a un metodo y llamarlo generará una InaccessibleObjectException (RuntimeException), utilizar **AccessibleObject::trySetAccesible:boolean**

**Variable/Method handler**

- ofrece mas type safety y mejor performance que la reflexion

- reflexion requiere que el usuario abra los paquetes para que sean usados por la reflexion, el usuario tiene que conocer basado en su entendimiento del sistema de modulos

- el lookup usado por method y variable handlers crea una instancia que captura los derechos de acceso del que crea el lookup sin importar al modulo que pertenece (se puede crear dentro del modulo sin open y pasarlo a un modulo que lo importa y tendra acceso = la clase X del modulo Y le entrega acceso a otro modulo Z sin ser declarado open).

**Acceso a las propiedades del Modulo**

- Module m = getClass().getModule();

**Modificar propiedades de un modulo con reflexion**

- m.addExports() → exporta un package a un modulo

- m.addOpens() → abre un paquete a un modulo

- m.addReads() → permite al modulo leer otro

- m.addUses() → hace que el modulo use un servicio

- los metodos son *caller sensitive*, dependiendo del codigo que lo llama es el comportamiento que tienen.

- para que sea exitoso tiene que ser llamado desde el interior del modulo que se esta modificando o desde el unnamed module, en otro caso fallara y lanzará IllegarCallerException (se puede usar --patch-module)

**- paquetes de modulos abiertos a otros modulos pueden ser abiertos a otros modulos = modulos con acceso reflexivo a un paquete (open) pueden abrirlos a otros modulos**

**Layers**

- una **module layer** comprende un grafo completamente resuelto de nombres de modulos y de classloaders utilizados para cargar las clases.

- cada classloader tiene un unnamed module asociado con el (ClassLoader::getUnnamedModule)

- también tiene referencias a uno o mas parent layer

- módulos en una capa pueden leer de sus padres, pero no de las capas que estan a su alrededor

- es posible apilar tantos gráficos como se requiera

- parent layers se definen cuando una capa es creada y posteriormente no puede ser modificada

- no es posible crear dependencias ciclicas

- no todos los módulos van en capas, los **unnamed modulos** y **dinamic modules** no, los demas si

- cuando se lanza la JVM crea una initial layer = **boot layer** que contiene los application y platform modules que fueron resueltos basados en el command line options

→ no contien padres y tiene 3 classloaders

→ **boot class loader:** entrega a las clases que carga todos los permisos de seguridad (java.base)

→ **platform class loader:** carga clases de todos los otros platform modules. Puede ser accedido con el metodo estatico ClassLoader::getPlatformClassLoader

→ **system/application class loader**: carga las clases desde los modulos y classpath, es responsable de todos los application modules. Puede ser accedido con ClassLoader::getSystemClassLoader

– app < platform < boot ← class loaders

– no puedo obtener el padre del platform class loader

– solo el system class loader tiene acceso al class path, de esta manera solo este unnamed module (que referencia al system class loader) nunca estará vacio

– class loaders no estan aislados, tienen un padre e implementaciones, primero consultan al padre antes de hacerlo por ellos

→ system → platform → boot

→ system class loader tiene acceso a todos los application y JDK classes del boot y platform class loaders.

→ layers son representadas por instancias de java.lang.ModuleLayer

→ modules() retorna un Set<Module>

→ findModule(String nombreModulo) busca el modulo en la capa y sus ancestros, retorna un Optional<Module>

→ parents() retorna un List<ModuleLayer>

→ sus class loader puede ser determinado usando findLoader(String nombreModulo)

→ configuration(): Configuration

ModuleLayer layer = getClass().getModule().getLayer();

– getLayer() retornará null si viene desde un unnamed module o dynamic module (no están en layers)

– el metodo estatico ModuleLayer::boot retornara el boot layer

requerimientos fundamentales para carga y descarga de plugins, aplications, bundles y otros fragmentos corriendo en la JVM:

– tiene que ser posible hacer un transición/rotar en un fragmento de set de jars en runtime

– tiene que permitir interacción con los fragmentos cargados

– tiene que permitir aislamiento entre los fragmentos

- antes de modulos se generaban class loaders para los nuevos jars, se delega a otro class loader que permite acceder a otras clases en el running JVM.

→ una clase se identifica por su nombre cualificado, solo puede existir uno por cada class loader

→ puede ser cargado por varios classloaders

→ aísla los fragmentos y permite que cada uno sea ejecutado con sus dependencias sin conflicto con otros classloaders

- module system agrega capas alrededor de los class loaders, deja sin modificar su jerarquización

→ permite integracion con modulos cargados en launch time

– **configuration** permite lanzar el proceso de resolución de módulos

→ la instancia creada representa un exitoso grafo de módulos resuelto

→ resolve / resolveAndBind (agrega bind services, el primero no)

→ metodos:

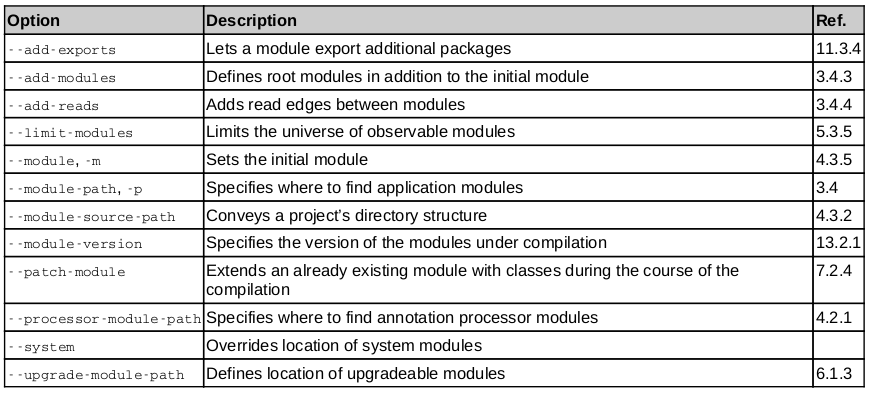
– ModuleFinder before: consultado para ubicar modulos antes de mirar en el parent configuration

– List<Configuration> parents: configuraciones de los parents modulos

– ModuleFinder after: consultado para ubicar modulos despues de mirar en el parent configuration

– Collection<String> roots: los root modules para el proceso de resolución

**javac command**



From Java 9 on the compiler prevents that common operating error with the --release option that sets all three options to the correct value (-source / -target / -bootclasspath)

javac

--module-path mods:libs

--module-source-path "./\*/src/main/java"

-d classes

--module monitor

jar --create

--file mods/monitor.jar

--main-class monitor.Main

-C monitor/target/classes .

jar --list --file mods/monitor.observer.jar

jar -t -f some.jar

jar --describe-module --file mods/monitor.observer.jar

jar -d -f mods/monitor.observer.jar

Un modular jar contiene la clase adicional module-info.class, lo demas es identico a los pre-modulos

--main-class ${class} , where ${class} is the fully qualified name (meaning the package name appended with a dot and the class name), It will be recorded in the module descriptor and used by default as the main class when the module is the initial module for launching an application.

**jar -d -f comun.jar**

modulos.comun jar:file:///home/sebastian/java/workspace/maven/modulos/comun/target/comun.jar/!module-info.class

exports com.sebastian.modulos.comun

requires java.base mandated

qualified opens com.sebastian.modulos.comun.abierto to modulos.segundo

contains com.sebastian.modulos.comun.privado

main-class com.sebastian.modulos.comun.Main

**compilar, empaquetar y ejecutar una aplicación:**

1 → find src -name \\*.java -print > file.list

2 → javac @file.list -d target/classes

3 → jar --create --main-class com.sebastian.modulos.comun.Main --file target/comun.jar -C target/classes/ .

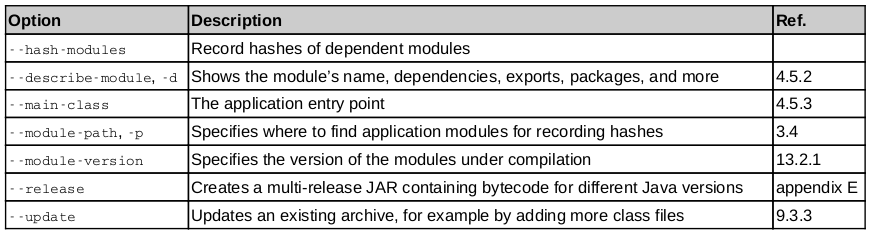
4 → java --module-path target/comun.jar --module modulos.comun/com.sebastian.modulos.comun.Main

java --module-path target/comun.jar --module modulos.comun

java -jar target/comun.jar

– al crear el jar no se valida que exista el main con --main-class (se confía)

**jar command relacionados con modulos:**



– Make sure to pick a directory structure that fulfills your project’s requirements—if in doubt stick to your build system’s default structure.

– The javac command to compile all of a module’s sources, including the declaration, is the same as before Java 9 except that it uses the module path instead of the class path.

– The module source path ( --module-source-path ) informs the compiler of how the project is structured. This lifts the compiler operation from processing types to processing modules, allowing us to compile a selected module and all its dependencies with a simple option ( --module or -m ) instead of listing source files.

– Modular JARs are just JARs with a module descriptor module-info.class . The jar tool will process it just as well as other class files, so packaging all of them into a JAR requires no new options.

– Optionally, jar allows the specification of a module’s entry point (with --main-class ), which is the class with the main method. This makes launching the module simpler.

The compiler supports so-called multi-module compilation , where it can build multiple modules at once. The command line option --module-source-path ${path} is used to enable this mode and to point out the directory structure containing the modules. All other compiler options work as usual

–

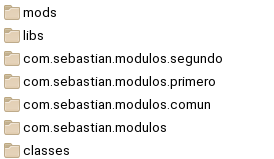
The java command has an option --module ${module} , which specifies the initial module ${module} . Module resolution will start from there and it is also the one from which a main class, meaning one with a public static void main method, will be launched. The specific class is either defined by the initial module’s descriptor or it can be specified with --module ${module}/${class} by appending the module name with a slash and and the fully qualified class name

→ utilizar --module ${module}/${class} sobreescribe el valor que pueda tener el modulo como clase main (manifest)

→

**compilar todos los proyectos, empaquetar y ejecutar:**

– estructura:

– mods contendra el destino al crear los jar modulos

– list tiene las librerias externas

– classes contendra las clases compiladas

[1] find . -name \\*.java -print > file.list

[2] javac --module-path mods:libs --module-source-path "./\*/src/main/java" -d classes @file.list --module com.sebastian.modulos.segundo

[3] jar --create --file mods/segundo.jar -C classes/com.sebastian.modulos.segundo/ .

[4] jar --create --file mods/comun.jar -C classes/com.sebastian.modulos.comun/ .

[5] jar --create --file mods/tercero.jar -C classes/com.sebastian.modulos/ .

[6] java --module-path mods --module com.sebastian.modulos.segundo/com.sebastian.modulos.segundo.Main

**– para poder ejecutar una clase main en otro modulo:**

java --module-path mods --add-modules=ALL-MODULE-PATH --module com.sebastian.modulos.comun/com.sebastian.modulos.comun.Main

– obtener un recurso desde una clase es relativo a la clase si no se usa el /

– desde el classloader comienza la busqueda desde la raiz aunque no se coloque el /

When it comes to **loading resources** across module boundaries things are a little different:

– resources from a **package** are by default be **encapsulated**

– resources from the **JAR’s root** or from **folders** whose **names can not be mapped to packages** (like **META-INF** because of the dash) are **never**

**encapsulated**

– **.class** files are **never encapsulated**

– If resources are encapsulated, the getResource call returns null .

– **exports** clauses **do not give access to resources**

– **give access** to a resource is the **opens** clause (usar reflexion para cargarlo desde la Class<?> obtenida)

– si el recurso esta en un paquete es fuertemente encapsulado

– if you want to give access to resources in a module’s package, you have to open it.

[**https://docs.oracle.com/javase/10/docs/api/java/util/spi/ResourceBundleProvider.html**](https://docs.oracle.com/javase/10/docs/api/java/util/spi/ResourceBundleProvider.html)

jmod describe /usr/lib64/jvm/java-11/jmods/jdk.attach.jmod

→ muestra las propiedades de un JMOD

jar --describe-module --file mods/comun.jar

→ describe un jar module

java --describe-module nombre\_modulo

→ describe el modulo asociado al nombre

**jmod describe** and **jar --describe-module** operate on **artifacts**, **java --describe** operates on **modules**

**java --module-path mods --describe-module com.sebastian.modulos.segundo**

com.sebastian.modulos.segundo file:///home/sebastian/java/workspace/maven/modulos/mods/segundo.jar

exports com.sebastian.modulos.segundo.exportado

requires com.sebastian.modulos.comun

requires java.base mandated

uses com.sebastian.modulos.segundo.exportado.Implementable

contains com.sebastian.modulos.segundo

contains com.sebastian.modulos.segundo.otro

java --module-path mods --**validate-modules**

→ escanea el module path buscando errores

→ reporta modulos duplicados y split packages

→ NO construye el module graph

→ NO descubre modulos faltantes o modulos con dependecias ciclicas

→ con la opcion **--dry-run** se constrira con la completa resolucion de modulos (el module graph y la configuracion confiable) pero se detendrá antes de ejecutar el main

→ **--dry-run** must come before --module even if that looks sequentially displeasing.

→ And a note for experts: If you’re using a custom class loader, custom security manager, or agents, then they will be initiated even with **--dry-run**

**java --list-modules**

→ muestra todos los modulos de la plataforma en el actual runtime

→ **lists the universe of observable modules** (platform modules (runtime) + application modules (en el module path))

→ no lanza la app, no indica errores, …

→ modulos seran elejidos de ese universo para construir el module graph

→ la aplicación NO tendra modulos que no estan listados acá

→ pero modulos listados pueden no terminar en el module graph

→ when calling java --list-modules we tasked the JVM with listing all observable modules and because we did not specify a module path only the runtime’s platform modules would be printed.

→ java --module-path mods:libs –list-modules

– muestra todos los del runtime + los del module path

– java **--limit-modules** java.xml --list-module

→ muestra las dependencias transitivas del modulo inicial

→ java --module-path mods:libs --list-modules --limit-modules com.sebastian.modulos.segundo

:

java.base@11.0.2

com.sebastian.modulos.comun file:///home/sebastian/java/workspace/maven/modulos/mods/comun.jar

com.sebastian.modulos.segundo [file:///home/sebastian/java/workspace/maven/modulos/mods/segundo.jar](mods/segundo.jar)

(en mods hay 3 modulos, la salida muestra solo las dependencias, sin –limit-modules muestra los 3 + todos los del runtime)

The option --limit-modules ${modules} accepts a list of

comma-separated module names. It limits the universe of observable

modules to the specified ones and their transitive dependencies. If the

options --add-modules (see section 3.4.3) or --module (see section 5.1)

are used together with --limit-modules , the modules specified for those

two options become observable but their dependencies do not !

1. Starting from the modules specified to --limit-modules , the JPMS determines all their

transitive dependencies. This is subject to the requirements for reliable configuration as

described in section 3.2.1.

**2. If --add-modules or --module was used, it adds the specified modules (but not their**

**dependencies).**

3. It uses the resulting set as the universe of observable modules for any further steps (like

listing modules or launching the application).

java **--module-path** mods:libs --limit-modules com.sebastian.modulos.segundo --module com.sebastian.modulos.segundo/com.sebastian.modulos.segundo.Main

→ si los modulos que fueron listados con --list-modules (java --module-path mods:libs --limit-modules com.sebastian.modulos.segundo --list-modules) tienen alguna dependencia NO será agregado al module graph (NO se podra ejecutar)

– LOGS del JPMS

→ diagnostic messages from the resolver

→ unified JVM logging

→ With the option **--show-module-resolution** the module system prints messages during module resolution.

→ it identifies the root modules (one on this case), modules that were loaded as a dependency, and which dependency that was

→ **java --module-path mods:libs --limit-modules com.sebastian.modulos.segundo --show-module-resolution --dry-run --module com.sebastian.modulos.segundo/com.sebastian.modulos.segundo.OtroMain**

root com.sebastian.modulos.segundo file:///home/sebastian/java/workspace/maven/modulos/mods/segundo.jar

com.sebastian.modulos.segundo requires com.sebastian.modulos.comun [file:///home/sebastian/java/workspace/maven/modulos/mods/comun.jar](mods/comun.jar)

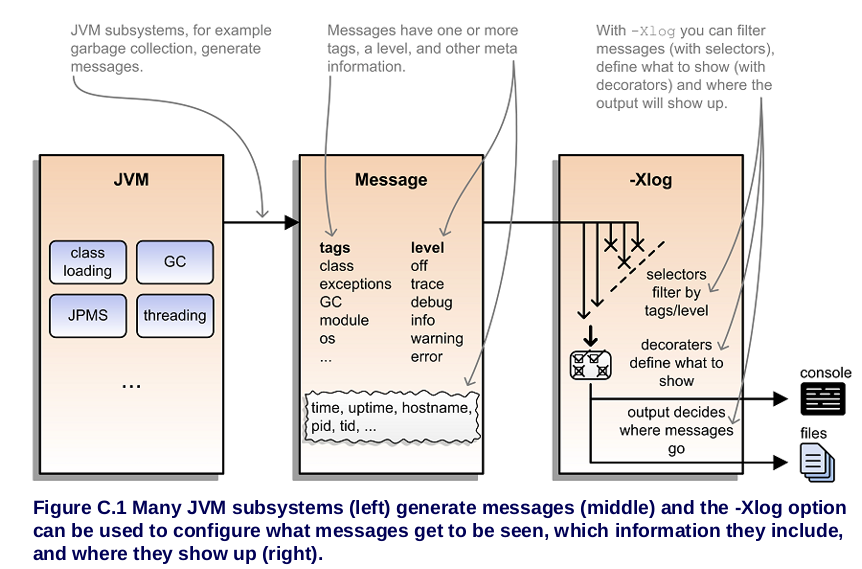
– **-Xlog** = **logs de la JVM** (NO de la JDK → no de swing, ni de la aplicación que se esta lanzando)

→ JVM-internal unified logging

→ which messages to log (by tag and/or by log level)

→ which information to include (for example time stamps and process IDs)

→ which output to use (for example into a file)



→ usar log level y tags para definir **cuales logs mostrar**

→ se definen pares **<tag-set>=<level>** llamados **selectors**

→ usando **all** se seleccionan **todos los tags**, el **nivel** es **opcional** y por **defecto** es **info**

→ **java -Xlog:all=warning** -version → no muestra logs =)

→ **java -Xlog:logging=debug** -version → el tag logging lo muestra desde el nivel debug

→ **java -Xlog:gc** -version → SOLO los que tengan EL tag gc (no otros)

→ **java -Xlog:gc,gc+heap** -version → SOLO los que tengan LOS tags gc o gc y heap (no otros)

→ **java -Xlog:gc\*** -version → los que tengan el tag gc y opcionalmente cualquier otro tag asociado

→ **java -Xlog:help** → comandos de ayuda

**–** java -Xlog:gc\***:file=/tmp/jvm.log** -version → hay opciones para determinar el destino de la salida

→ **:file=qwerty** → envia la salida al archivo qwerty

→ **:stdout** → salida por defecto

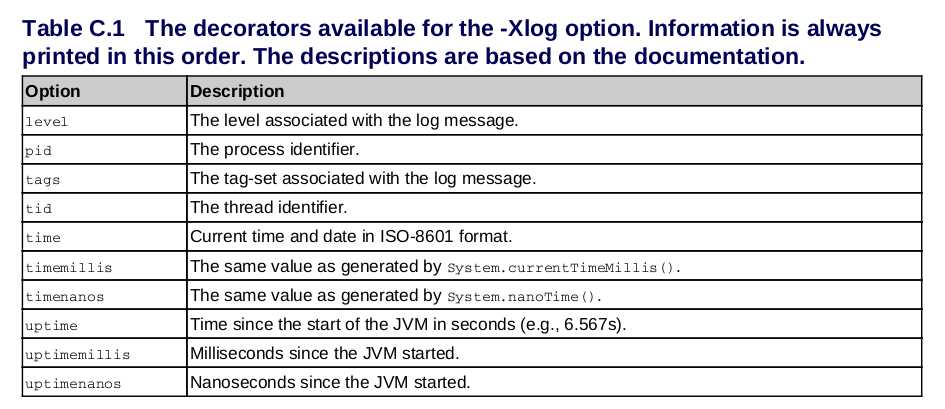
→ **:stderr** → salida de error

→ NO se pueden mezclar

**–** despues del output location se puede definir que deberian decir los mensajes

→ cada mensaje consiste en texto y meta informacion

→ **decorators** definen que imprimirá la JVM (separados por comas, QUE informacion el mensaje incluye)

→ java -Xlog:gc\*:file=/tmp/jvm.log**:time** -version

→ [2019-03-06T00:38:41.242-0300] Heap region size: 1M

→ -Xlog:<selectors>:<output>:<decorators>:<output-options>

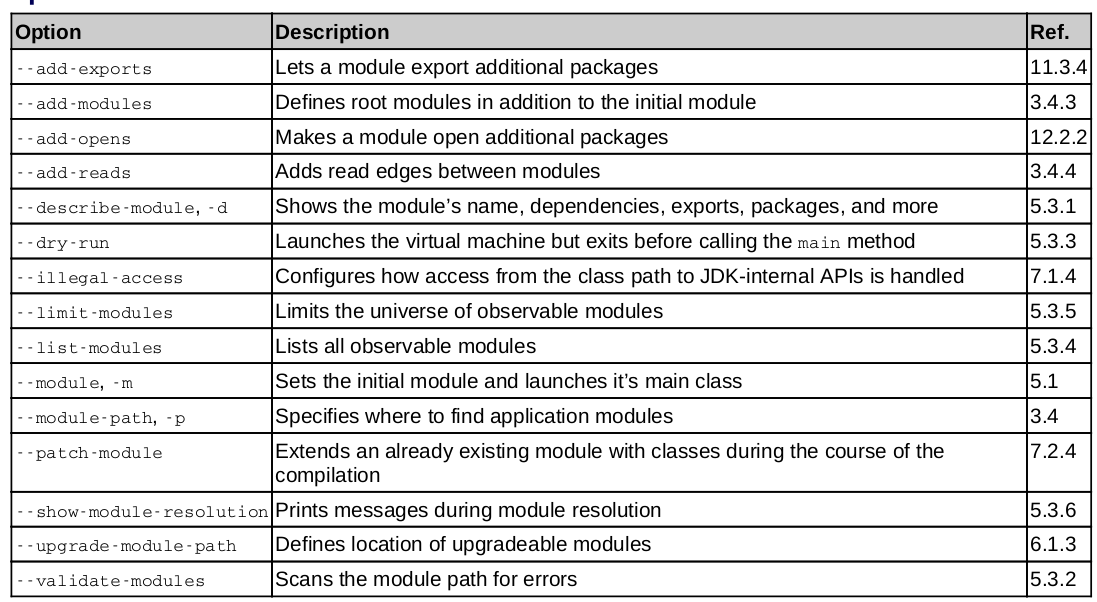
→ los parametros son opcionales PERO si se usa uno HAY que usar los anteriores

[**https://docs.oracle.com/javase/9/tools/java.htm#JSWOR-GUID-BE93ABDC-999C-4CB5-A88B-1994AAAC74D5**](https://docs.oracle.com/javase/9/tools/java.htm#JSWOR-GUID-BE93ABDC-999C-4CB5-A88B-1994AAAC74D5)

→ trace > debug > info

→ toda la carga sucede en un solo hilo

<https://docs.oracle.com/javase/9/tools/java.htm>



– Make sure to list all JVM options before --module or they will be treated as application options and won’t affect the module system.

– Observable modules can be listed with **--list-modules**. This comes in handy if you need to debug problems and want to see which modules were **available for resolution**.

– If --limit-modules is used, the universe of observable modules only consists of the specified modules and their transitive dependencies, thus reducing the modules that are available during resolution. Together with --list-modules , it is a great way to determine a module’s transitive dependencies.

– The option --add-modules can be used to define additional root modules beyond the initial module. If a module is not required, for example because it is only accessed via reflection, --add-modules must be used to make sure it becomes part of the module graph.

– The option --dry-run launches the JVM and lets the module system process the configuration (module path, initial module, and so on) and build a module graph but exits just before the main method is called. This allows verifying a configuration without launching the application.

– The module system logs a variety of messages, which can be printed with either the simple --show-module-resolution or the more complex -Xlog:module\* . They allow analyzing how the module system puts together the module graph, which can help trouble-shooting.

– Loading resources from modules works much like loading them from JARs. The only exception are resources that are not .class files and in a different module’s package (as opposed to, for example, the JAR’s root or META-INF folder)—these are by default encapsulated and thereby not accessible.

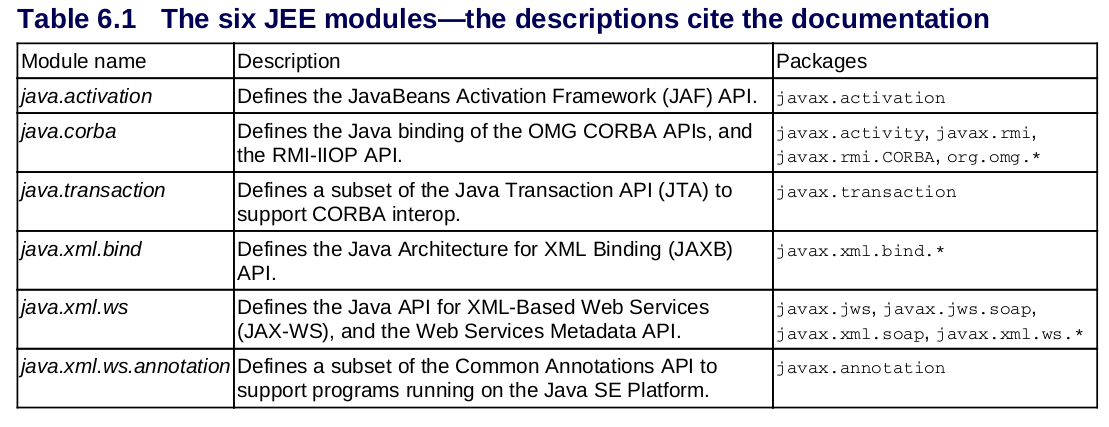
– A module can use opens directives to give reflective access to packages, which exposes resources located therein and thus allows other modules to load them. Unfortunately, this solution invites other code to depend on the module’s internal structure.

– When loading resources, default to the methods getResource and getResourceAsStream on Class or their counterparts on the new type java.lang.Module . Those on ClassLoader have a generally less useful behavior.

**– The class path is still fully functional and during a migration to Java 9 you will continue to use it instead of the module path.**

**– Even then, the module system is still in play, for example regarding module resolution**

**– Code on the class path will automatically read most modules (but not all, check section 6.1), so they are available at compile or run time without additional configuration.**



– Java SE contains a few packages that are comprised of **endorsed (aprobados) standards and standalone technologies**

→ These technologies are developed outside the Java Community Process, often because they rely on standards governed by other bodies. Examples are the Document Object Model (DOM), developed by the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG), and Simple API for XML (SAX).

→ [**https://docs.oracle.com/javase/8/docs/technotes/guides/standards/**](https://docs.oracle.com/javase/8/docs/technotes/guides/standards/)

→ many of them fall into the JEE modules

→ Historically, the Java Runtime Environment (JRE) shipped with implementations of these technologies but was ready to let users upgrade them independently of the JRE. This could be done with the endorsed standards override mechanism

→ Similarly, application servers often extend or upgrade the Corba, JAXB, or JAX-WS APIs as well as the JavaBeans Activation Framework (in java.activation ) or the Java transaction API (in java.transaction ) by providing their own implementations

→ java.xml.ws.annotation contains the javax.annotation package. It is often extended by the various JSR 305 implementations

→ In all these cases of extending or replacing APIs that ship with the JRE, the trick is to use the exact same package and class names, so the classes get loaded from an external JAR instead of the JRE

→ In the parlance (lenguaje) of the module system this is called a **split package: The same package is split across different modules or a module and the class path**

→ Splitting packages no longer works in Java 9 and later

→ classes on the class path that are in a package that is present in the JRE are effectively invisible

→ If the JRE contains a class with the same fully qualified name, that one will get loaded.

→ If the JRE-version of the package does not contain the required class, the result is the compile error or NoClassDefFoundError (And that happens regardless of whether the class is present on the class path or not)

→ la version de modulo hace invisible al de classpath (si esta repartido entre modulos y classpath)

→ Java 9 and 10 will by default not resolve the JEE modules, meaning they do not make it into the module graph and are hence (xlo tanto) not available

→ To get rid of this complexity and to properly separate Java SE from JEE, these modules are deprecated in Java 9 and removed in Java 11. With their removal, command line tools like wsgen and xjc are also no longer shipped with the JDK

→ **JDeps analysis (see appendix D)**

→

javac

--class-path ${jars}

--add-modules java.xml.bind

-d ${output-dir}

${source-files}

→

java

--class-path ${jars}

--add-modules java.xml.bind

${main-class}

→ java.se.ee → It makes all six EE modules available

→ The effort of manually adding JEE modules is only required for unmodularized code

[**https://stackoverflow.com/questions/48204141/replacements-for-deprecated-jpms-modules-with-java-ee-apis**](https://stackoverflow.com/questions/48204141/replacements-for-deprecated-jpms-modules-with-java-ee-apis)

[**http://openjdk.java.net/jeps/320**](http://openjdk.java.net/jeps/320)

– **--upgrade-module-path**:

→ acepta una lista de directorios formateados como el module path. Cuando el module system crea el module graph busca en esos directorios los artefactos que y los usa para reemplazar los **modulos actualizables**, siempre los 6 modulos JEE son actualizables:

java.activation

java.corba

java.transaction

java.xml.bind

java.xml.ws

java.xml.ws.annotation

→ jdk vendors pueden hacer más modulos actualizables

→ oracle jdk tiene java.jnlp actualizables

→ aplication modules que fueron linkeados en una imagen con **jlink** son siempre actualizables

→ jars en el **upgrade module** path no necesitan ser modulos, terminarán en el **automatic modules** si no tienen module descriptory pueden reemplazar java modules.

– In all Java versions, the application class loader (also often called the system class loader ) is one of three class loaders that the JVM uses to run an application.

→ It loads JDK classes that do not need any special privileges as well as all application classes (unless the app uses its own class loaders, in which case none of the following applies).

→ You can access the **application class** loader by calling **ClassLoader.getSystemClassLoader()** or by taking an instance of one of your classes and calling **getClass().getClassLoader()** on it. Both methods promise to give you an instance of type ClassLoader

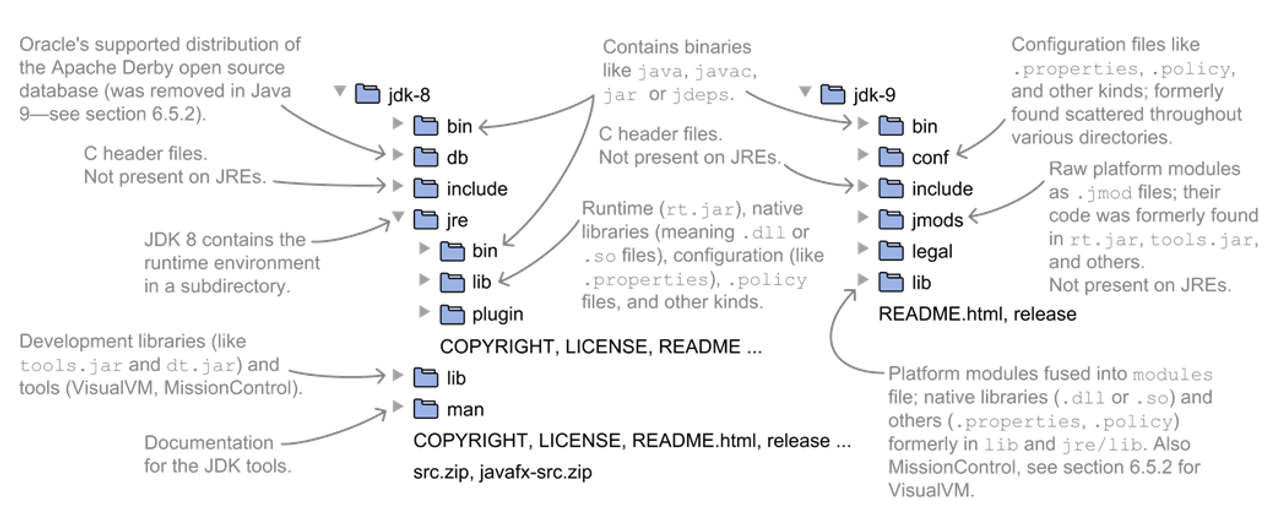
→ On **Java 8 and before**, the **application class loader** is a **URLClassLoader**, a **subtype** of **ClassLoader**

→ sin modulos URLClassLoader no tiene idea en cual artefacto encontrar una clase, por lo tanto busca en todos los artefactos del class path hasta que lo encuentran

→ desde java 9, los jars tienen una representacion apropiada en run time, cuando la clase necesita ser cargada, se identifica el paquete al que pertenece y se usa para determinar su modulo.

→ solo se escanea ese jar encontrado

→ se asume que solo un modulo puede contener la clase, no pueden 2 jars modulares contener tipos en el mismo paquete, si lo hacen es un split package y el module system lanza el error



→ the URL you get for system resources, for example from ClasLoader::getSystemResource , changed. It used to be of the following form, where ${path} is something like java/lang/String.class:

jar:[file:$](../../../../../../$){java-home}/lib/rt.jar!${path} (java pre 9)

jrt:/${module}/${path} (java 9)

→ Furthermore, the Class::getResource\* and ClassLoader::getResource\* methods no longer read JDK-internal resources. Instead, to access module-internal resources, use Module::getResourceAsStream or create a JRT file system as follows:

FileSystem fs = FileSystems.getFileSystem(URI.create("jrt:/"));

fs.getPath("java.base", "java/lang/String.class"));

→ With the module system in play, much more flexible **runtime images can be created with jlink** (see section 14.1) and compact profiles are no longer needed. The Java 9 compiler will hence only accept -profile if compiling for Java 8. To compile against a specific selection of modules you can use the --limit-modules option

– These are the modules you need to get the same APIs as the three compact profiles:

for **compact1** profile: java.base , java.logging , and java.scripting

for **compact2** profile: those for compact1 plus java.rmi , java.sql , and java.xml

for **compact3** profile: those for compact2 plus java.compiler , java.instrument, java.management , java.naming , java.prefs , java.security.jgss , java.security.sasl, java.sql.rowset , and java.xml.crypto

→ NO se soporta agregar clases al JDK sin agregarlas al classpath (agregadas en java.ext.dirs, lib/ext o alguna forma que seaplatform-specific system-wide directory),

– alternativas:

→ el compilador y el runtime terminaran con error si existe el directorio JRE o el system property es definido

→ se puede usar con java y javac --patch-module para inyectar modulos o --upgrade-module-path para reemplazar un modulo actualizable

→ el artefacto extendido puede ser colocado en el class path (183)

– Before Java 9, the endorsed standards override mechanism allowed replacing certain APIs with custom implementations. It loaded them from the directories named by the system property java.endorsed.dirs or the lib/endorsed directory in the JRE. Java 9 removes this feature and compiler and runtime will exit with an error if the JRE directory exists or the system property is set (las alternativas son las mismas).

– The **-Xbootclasspath and -Xbootclasspath/p options were removed**. Use the following options instead:

the javac option --system specifies an alternate source of system modules the javac option --release specifies an alternate platform version the java and javac option --patch-module injects content into modules in the initial module graph

– NO hay compilacion para Java 5

→ puede **procesar** versiones anteriores con **-source**

→ puede **producir** bytecode usando **-target**

→ java 9 soporta 9, 8, 7 y 6

→ java 8 soporta 8, 7, 6 y 5

→ usando la ultima soportada (-source o -target) genera deprecated warnings

→ con java 10 y 11 es posible compilar hasta java 6

→ el compilador puede reconocer y procesar bytecode de todas las versiones pero NO produce bytecode de versiones antes de la 6

– Before Java 9, you could use the **-version:N** option on java (or the corresponding manifest entry) to launch the application with a JRE of version N. In Java 9 the feature was **removed** and the Java launcher quits with an error for the command line option and prints a warning for the manifest entry while otherwise ignoring it

– new format for version strings

→ ya no es 1.x

→ the system property java.version and its siblings java.runtime.version, java.vm.version, java.specification.version, and java.vm.specification.version no longer start with 1.x but with x. Similarly java -version returns x , so on Java 9 you hence get 9.something .

→ 9.${MINOR}.${SECURITY}.${PATCH} , where ${SECURITY} has the peculiarity that it doesn’t reset to zero when a new minor version is released

→ ${FEATURE}.${INTERIM}.${UPDATE}.${PATCH}, where ${FEATURE} starts with 10 and increases every six months with each **feature release**. ${**INTERIM**} acts as you would expect from ${MINOR} , but because **no minor releases are planned** in the new

schedule, it is assumed to always stay 0 .

→

Version version = Runtime.version();

System.out.println("feature: " + version.feature());

System.out.println("interim: " + version.interim());

System.out.println("update: " + version.update());

System.out.println("patch: " + version.patch());

– removal of a number of JDK and JRE tools

→ JavaDB no es incluida

→ VisualVM no es incluida

→ hprof no es incluida (jcmd y jmap lo pueden reemplazar y el Java Flight Recorder)

→ jhat fue removido

→ java-rmi.exe y java-rmi.cgi fue removido (usar servlet como proxy RMI sobre HTTP para tener una alternativa)

→ native2ascii fue removido (convertia UTF8 base properties resource bundle a ISO8859-1), Java 9 soporta UTF8 based bundles

→ comandos relacionados con JEE fueron removidos (wsgen, xjc, wsimport)

– single underscore is not a valid identifier

– JNLP syntax update

<https://jcp.org/aboutJava/communityprocess/maintenance/jsr056/9.html>

– removed JVM options

→ combinations are no longer supported ( DefNew + CMS , ParNew + SerialOld , Incremental CMS )

→ configurations were removed ( -Xincgc , -XX: +CMSIncrementalMode ,

-XX:+UseCMSCompactAtFullCollection , -XX:+CMSFullGCsBeforeCompaction ,

-XX:+UseCMSCollectionPassing )

→ deprecated ( -XX:+UseParNewGC ). Java 10, in

turn, removes -Xoss , -Xsqnopause , -Xoptimize , -Xboundthreads , and

-Xusealtsigs .

– deprecated in Java 9 with the intention to remove them in future releases:

→ the Applet API in the java.applet package together with the appletviewer tool and the Java browser plugin

→ Java Web Start, JNLP, and the javaws tool

→ the Concurrent Mark Sweep (CMS) garbage collector

→ the HotSpot FlatProfiler, activated with -Xprof

→ the policytool security tool

– Java 10 and 11 already followed through on some of the deprecations:

→ Java 10 removes the FlatProfiler and policytool

→ Java 11 removes the Applet API and Web Start

→ <http://www.oracle.com/technetwork/java/javase/9-deprecated-features-3745636.html>

→ <http://www.oracle.com/technetwork/java/javase/10-relnote-issues-4108729.html#Removed>

→ <https://docs.oracle.com/javase/9/docs/api/deprecated-list.html#forRemoval>

→ <https://docs.oracle.com/javase/10/docs/api/deprecated-list.html#forRemoval>

— 7 —

– usar JDK internal APIs genera errores de compilacion (sun.\*, tambien codigo de internals a librerias o frameworks)

– splitting package repartidos entre artefactos causa errores de compilacion y runtime (entre mi codigo y los de la JDK, entre otros 2 artefactos, mi codigo y otras dependencias,…)

– Java Dependency Analysis Tool (JDeps)

– APIs internal: clase que no es publica o no esta en un paquete exportado.

– tipos de JDK apis:

→ standarizada

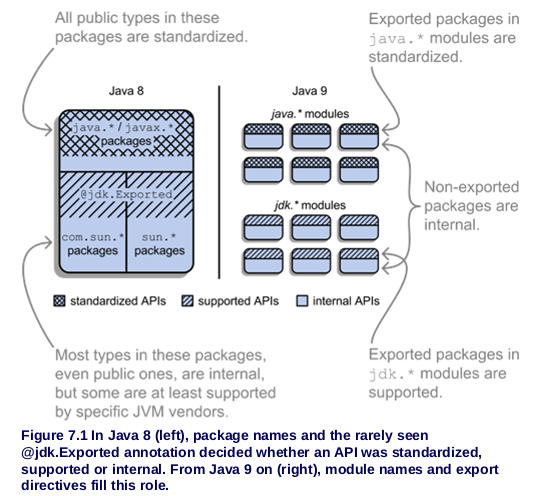
→ soportada

→ internal

→ clases publicas de java.\* (java specification) y javax.\* son estandarizadas y completamente soportadas a travez de todos los JRE, usandolas se obtiene codigo portable = **standardized**

→ algunas com.sun.\* y jdk.\* (paquetes o clases) son marcadas con la anotacion jdk.Exported = son soportadas por Oracle y no necesariamente estan presentes en jres no oracle (el codigo queda atado al especifico JRE) = **supported**

→ la mayoria de com.sun.\* y todos los sun.\* paquetes y no publicas clases son internas y pueden cambiar entre distintas versiones y jre (mas inestables y el codigo puede dejar de estar disponible en cualquier minor update) + otras clases = **internal APIs**

– jdk.\* modules are not standardized, son una convencion y el sistema de modulos no tiene preocupacion de ellos, no es deseable depender de ellos pero el JPMS no los encapsulará y ninguno de los comandos discutidos sera necesario

– Here, when we talk about internal APIs we mean those that the module system makes inaccessible because classes are not public or packages not exported

– **jdk.unsupported**: APIs que no tienen reemplazo, existen antes de java 9, su contenido puede cambiar en una siguiente release:

→ from sun.misc : Signal , SignalHandler and Unsafe

→ from sun.reflect : Reflection and ReflectionFactory

→ from com.sun.nio.file : ExtendedCopyOption , ExtendedOpenOption ,

→ ExtendedWatchEventModifier , SensitivityWatchEventModifier

– removidos:

→ everything in sun.misc and sun.reflect that is not part of jdk.unsupported, for example sun.misc.BASE64Encoder , sun.misc.BASE64Decoder , sun.misc.Cleaner ,and sun.misc.Service

→ com.sun.image.codec.jpeg and sun.awt.image.codec

→ com.apple.concurrent

→ com.sun.security.auth.callback.DialogCallbackHandler

→ methods addPropertyChangeListener and removePropertyChangeListener on java.util.logging.LogManager , java.util.jar.Pack200.Packer and java.util.jar.Pack200.Unpacker (were deprecated in Java 8)

→ methods with parameter or return types from java.awt.peer and java.awt.dnd.peer (these packages were never standardized and are internal in Java 9 and later)

– **JDeps**

→ Java Dependency Analysis Tool

→ desde Java 8

→ analiza el bytecode (.class y jars)

→ registra todas las dependencias declaradas estáticamente entre clases, que luego se pueden filtrar o agregar

→ **--jdk-internals**:

— lista todas las internas apis que se referencian incluyendo las del jdk.unsupported

— muestra el jar y los modulos que contienen las APIs problematicas

— las especificas clases involucradas

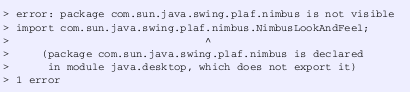
— la razon porque la dependencia es problematicas

— **jdeps --jdk-internals -R --class-path 'lib/\*:plugins/\*' scaffold-hunter-2.6.3.jar**

→ usando **--add-exports** ${modulo}/${package}=${reading-module} (en java y javac) exporta ${package} de ${modulo} a ${reading-module}. El codigo de ${reading-module} puede acceder a todos los tipos publicos en ${package} PERO no otros modulos.

→ usando ${reading-module} con ALL-UNNAMED todo el codigo desde el classpath puede acceder al paquete

→



javac

**--add-exports** java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED

--class-path ${dependencies}

-d ${target-folder}

${source-files}

— permite **COMPILAR** el codigo contra clases encapsuladas, pero no en runtime (no queda informacion en el bytecode)

→at least in Java 9, 10, and 11, JDK-internal dependencies are still available at run time

→ Running code from the module path expresses that it has been prepared for the module system, in which case there is no need to make an exception. It is hence **limited to class path code**.

→ From a compatibility perspective, there is no reason to grant access to application modules as these did not exist before Java 9, so the exception is **limited to platform modules**.

→ at least Java 11, if the exception were permanent, the incentive to update troublesome code would be much lower.

– cuando se le permite al classpath acceder a JDK internals se distinge el codigo que accede en forma estatica del que accede por reflexion:

→ acceder por reflexion genera warnings, no es pobible conocer todas las llamadas que se hacen, en ejecucion es el unico tiempo que se pueden reportar

→ acceso estatico no genera warnings

– la opcion **--illegal-acces=${value}** controla como el acceso a las APIs del JDK es controlado

→ **permit**: se permite el acceso a todo el JDK internals desde el classpath, al usar reflexion un unico warning es reportado en el primer acceso a cada paquete (default)

→ **warn**: como permit pero el warning es por cada acceso con reflexion

→ **debug**: como warn pero se incluye un stacktrace en cada warning

→ **deny**: se prohibe todo el illegal access

usando java 9.0:

javac --add-exports=java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED -d target/classes src/main/java/com/sebastian/nimbus/AccesoNimbus.java

java --illegal-access=deny --add-exports=java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED -cp target/classes com.sebastian.nimbus.AccesoNimbus

– **--illegal-access** NO aplica a las nuevas JDK internals, solo a las que existen pre java 9

→ administra el acceso ilegal en forma globally

→ es una opcion de transicion y eventualmente desaparecera

→ avisa con warnings

– **--add-exports** permite usar **publicos miembros de publicos tipos** en los paquetes cubriendo todo el acceso estatico

→ NO permite el acceso a otros typos no publicos y campos no publicos, se mantienen encapsulados

→ para reflexion sobre esos tipos de datos hay que usar –**add-opens**

**– --add-opens** usa la misma sintaxis que --add-exports

→ permite abrir los paquetes para deep reflection-based: todos los tipos y sus miembros son accessibles sin importar su modificador de visibilidad

→

> Caused by: java.lang.reflect.InaccessibleObjectException:

> Unable to make ClassLoader.defineClass accessible:

> module java.base does not "opens java.lang" to unnamed module

→

java

--class-path ${jars}

--illegal-access=deny

**--add-opens java.base/java.lang=ALL-UNNAMED**

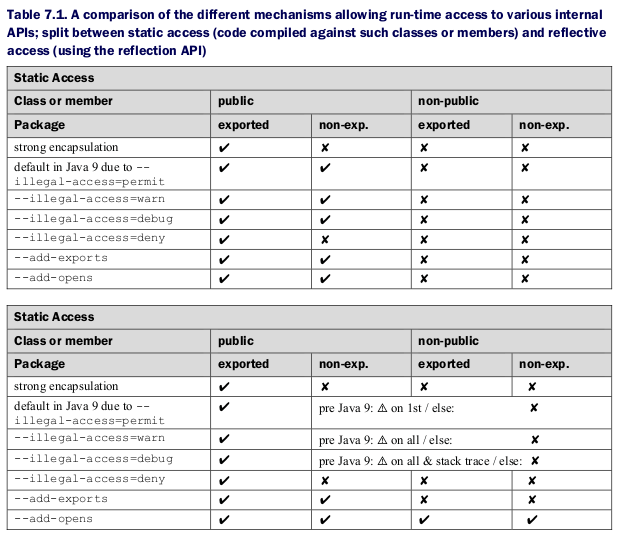
${main-class}

→ <https://stackoverflow.com/a/46551505/2525313>

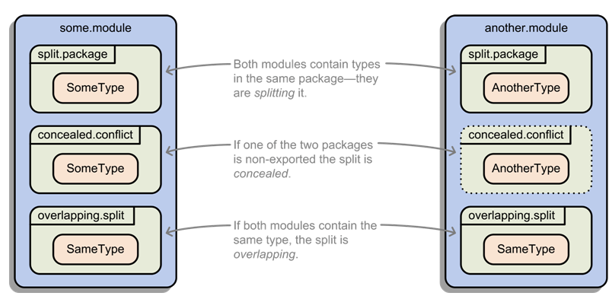
→ JDeps encuentra las dependencias estaticas, no hay forma de conocer las referencias por acceso con reflexion

→ buscar java.lang.reflect.AccessibleObject::setAccessible o setAccessible para econtrar los posibles accesos

→ usar --illegal-access=debug o deny en los test o en la aplicación para conocer cuando hay uso de reflexion



— split packages —



→ el mecanismo de carga de clases fue implementando asumiendo que cualquier full qualified name es unico dentro del mismo classloader

→por defecto hay un classloader

→ el sistema de modulos mejora el rendimiento de la carga de clases

→ conoce por cada paquete a que modulo pertenece

→ cada paquete pertenece a un unico modulos

→ split paquetes chocan con un importante objetivo del sistema de modulos: encapsulamiento atraves de los limites de los modulos

→

A **package** is supposed to contain a coherent set of **classes** with a **single purpose** and a **module** is supposed to contain a coherent set of **packages** with a **single**, although somewhat larger, **purpose**. In that sense, **two modules containing the same package have overlapping purposes**. Maybe **they should be one module**, then?

7.2.2 **— efectos del split packages**

→ un modulo NO puede leer el mismo paquete desde dos modulos distintos

→ 2 modulos en la misma capa NO estan permitidos contener el mismo paquete EXPORTADO o NO

→ estamos siempre implicitamente en el caso de una capa

→ cuando **2 modulos** (xej platform y application) forman un **split package** el sistema de modulos lo detecta y lanza un **error en compilacion y runtime** (modulo con un paquete que es tambien exportado desde un modulo requerido)

> monitor/src/main/java/monitor/statistics/SimpleStatistician.java:1:

error: package exists in another module: monitor.statistics

>package monitor.statistics;

> 1 error

→ cuando es split package pero no es exportado es concealed por que uno no lo exporta. Se puede compilar pero NO se puede ejecutar, indica que el paquete esta presente en ambos modulos

> Error occurred during initialization of boot layer

> java.lang.reflect.LayerInstantiationException:

> Package monitor in both module monitor.statistics and module monitor

→ lo mismo pasa en runtime cuando los modulos no son requeridos pero forman un split package

**— partido entre un modulo y el classpath**

→ todo el codigo en el classpath termina en el unnamed module

→ no hay checks sobre él

→ el module system no descubre split packages y dejará compilar y lanzar la aplicación

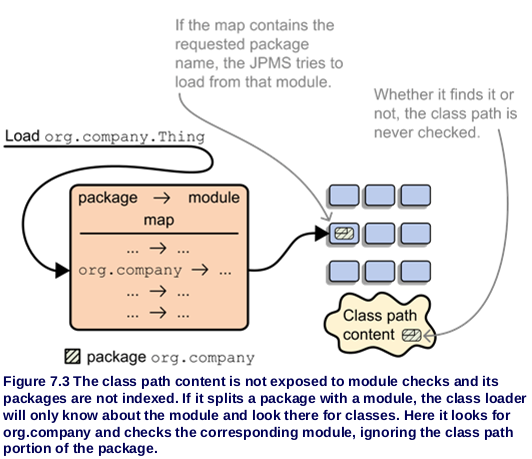
→ el systema de modulos conoce por cada named module cuales paquetes contiene y que cada paquete solo pertenece a un modulos

→ cuando se intenta cargar una clase el sistema de modulos busca el modulo que lo contiene y si no encuentra la clase genera un NoClassDefFoundError

→ al cargar una clase solo se buscará en el modulo cuando este cargando clases del paquete

→ las clases que pertenecen al mismo paquete pero esten en el class path son INVISIBLES

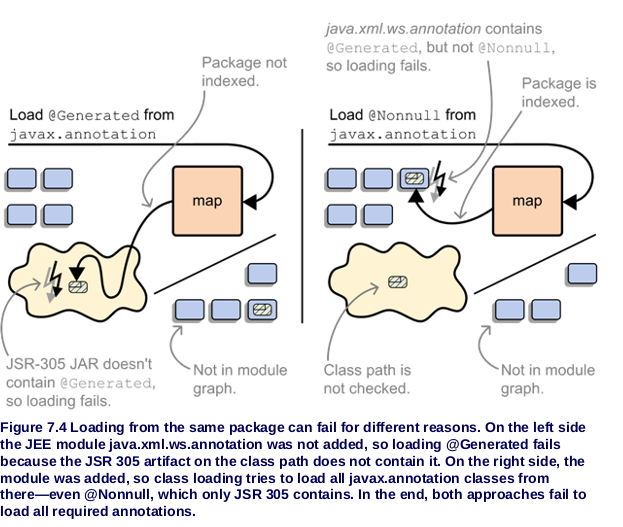
→ tambien aplica para modulos del module path y classpath



→ si creo el paquete javax.annotation con mis clases X en mi jar NO compilará

→ el modulo que contiene al paquete es de JEE: java.xml.ws.annotation que no es resuelto por defecto

→ --add-modules java.xml.ws.annotation agrega el modulo faltantes



1-rename one of the packages

2-move all parts of the split package into the same artifact

3-merge the artifacts

4-leave both artifacts on the class path

5-upgrade the JDK module with the artifact

6-patch a module with the artifact’s content

—

Earlier we looked at the example of a project that uses the annotations @Generated (from the java.xml.ws.annotation module) and @Nonnull (from a JSR 305 implementation).

We discovered three things:

both annotations are in the javax.annotation package, thus creating a split we need to add the module manually because it’s a JEE module doing so makes the JSR 305 portion of the split package invisible

Now we know that we can use --patch-module to mend the split:

javac

--add-modules java.xml.ws.annotation

--patch-module java.xml.ws.annotation=jsr305-3.0.2.jar

--class-path 'libs/\*'

-d classes/monitor.rest

${source-files}

This way all classes in jsr305-3.0.2.jar become part of the module java.xml.ws.annotation and can hence be loaded for a successful compilation (or, on java , execution). Yay!

→ con patch el modulo NO es agregado automaticamente al module graph, si no es requerido explicitamente se agrega con --add-modules

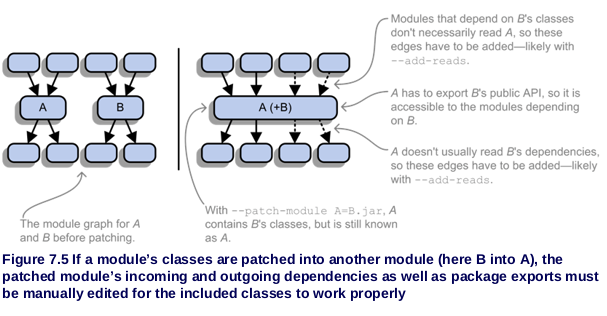
→ agregas con patch estan sujetas a las normales reglas de validacion de accesibilidad

→ codigo que depedende de las clases tiene que leer del patched module, el cual tiene que exportar los requeridos paquetes

→ las dependencias tienen estar en paquetes exportados en modulos leidos del patched one

→ puede ser necesario usar **--add-reads** y **--add-exports**

→ **--upgrade-module-path** y --patch-module son utiles para los split packages o upgradeables modulos



\* javac -cp libs/\* src/main/java/com/sebastian/jsr/SplitModulos.java -d classes

\* jar --create --file jsr-305.jar -C classes/ .

\* java -cp jsr-305.jar com.sebastian.jsr.SplitModulos

\*

\* -- agregado paquete javax.annotation para que no compile:

\* javac --add-modules java.xml.ws.annotation -cp libs/\* src/main/java/com/sebastian/jsr/SplitModulos.java src/main/java/javax/annotation/Saludador.java -d classes

\* -- si compila:

\* javac -cp libs/jsr305-3.0.2.jar src/main/java/com/sebastian/jsr/SplitModulos.java src/main/java/javax/annotation/Saludador.java -d classes

\* -- si es necesario:

\* javac --patch-module java.xml.ws.annotation=libs/libs/jsr305-3.0.2.jar --add-modules java.xml.ws.annotation -cp libs/jsr305-3.0.2.jar src/main/java/com/sebastian/jsr/SplitModulos.java src/main/java/javax/annotation/Saludador.java -d classes

\*

\* jar --create --file jsr-305.jar -C classes/ .

\*

\* genera un error por split packages:

\* java --add-modules java.xml.ws.annotation -cp jsr-305.jar com.sebastian.jsr.SplitModulos

\*

\* se soluciona con:

\* java --patch-module java.xml.ws.annotation=jsr-305.jar --add-modules java.xml.ws.annotation -cp jsr-305.jar com.sebastian.jsr.SplitModulos

\*

\*

\* tambien funciona

\* java --patch-module java.xml.ws.annotation=libs/jsr305-3.0.2.jar:libs/jsr305-3.0.3.jar:jsr-305.jar --add-modules java.xml.ws.annotation -cp libs/jsr305-3.0.2.jar:libs/jsr305-3.0.3.jar:libs/jsr-305.jar com.sebastian.jsr.SplitModulos

[1] → clase no exportada

**javac src/main/java/com/sebastian/nimbus/AccesoNimbus.java -d target/classes**

src/main/java/com/sebastian/nimbus/AccesoNimbus.java:3: error: package com.sun.java.swing.plaf.nimbus is not visible

import com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel;

^

(package com.sun.java.swing.plaf.nimbus is declared in module java.desktop, which does not export it)

1 error

[2] → se exporta la clase y compila con **warnings** por el acceso (ahora se podra acceder luego al ejecutarlo, el acceso es permitido por defecto en runtime estatica y por reflexion)

**javac --add-exports java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED src/main/java/com/sebastian/nimbus/AccesoNimbus.java -d target/classes**

src/main/java/com/sebastian/nimbus/AccesoNimbus.java:3: warning: NimbusLookAndFeel is internal proprietary API and may be removed in a future release

import com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel;

^

src/main/java/com/sebastian/nimbus/AccesoNimbus.java:30: warning: NimbusLookAndFeel is internal proprietary API and may be removed in a future release

NimbusLookAndFeel nimbus = new NimbusLookAndFeel();

^

src/main/java/com/sebastian/nimbus/AccesoNimbus.java:30: warning: NimbusLookAndFeel is internal proprietary API and may be removed in a future release

NimbusLookAndFeel nimbus = new NimbusLookAndFeel();

^

Note: src/main/java/com/sebastian/nimbus/AccesoNimbus.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

3 warnings

[3] → ejecutar la clase compilada genera warnings

**java -cp target/classes/ com.sebastian.nimbus.AccesoNimbusSoloReflexion**

WARNING: An illegal reflective access operation has occurred

WARNING: Illegal reflective access by com.sebastian.nimbus.AccesoNimbusSoloReflexion (file:/home/sebastian/java/workspace/maven/modulos/nimbus/target/classes/) to constructor com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel()

WARNING: Please consider reporting this to the maintainers of com.sebastian.nimbus.AccesoNimbusSoloReflexion

WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations

WARNING: All illegal access operations will be denied in a future release

por reflexion: [Nimbus Look and Feel - com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel]

[4] → no permitir el acceso a las classes internas (por defecto es posible)

java **--illegal-access=deny** -cp target/classes/ com.sebastian.nimbus.AccesoNimbus

Exception in thread "main" java.lang.IllegalAccessException: class com.sebastian.nimbus.AccesoNimbusSoloReflexion cannot access class com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel (in module java.desktop) because module java.desktop does not export com.sun.java.swing.plaf.nimbus to unnamed module @34ce8af7

[5] → exportar la clase

**java --illegal-access=deny --add-exports java.desktop/com.sun.java.swing.plaf.nimbus=ALL-UNNAMED -cp target/classes/ com.sebastian.nimbus.AccesoNimbus**

[Nimbus Look and Feel - com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel]

[Nimbus Look and Feel - com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel]

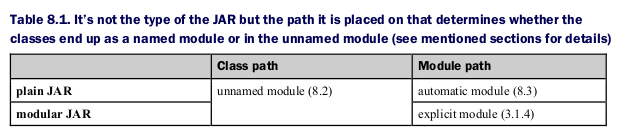
**– unnamed module** , which the module system implicitly creates with all the content loaded from the **class path**. In it, "the chaos of the class path" lives on

– The module system creates an **automatic module** for each **plain JAR** that it finds on the **module path**

– el classpath no hace distincion entre plain modular jar

→ si esta en el **classpath** termina en el **unnamed module**

→ si esta en el **module path** termina en su **propio nombre de modulo** (automatic module si es plain JAR, module JAR crea uno acorde a su descripcion)



→ cuando se decide donde colocar el JAR es necesario considerar **donde necesito que esté** (unnamed o named)

→ unnamed module es para compatibilidad, permite usar el module system aunque las dependencias no esten modularizadas

**The unnamed module—a.k.a the class path**

– el unnamed module contiene todos los no modular classes, las cuales son:

→ en tiempo de compilacion, las clases que están siendo compiladas si no incluyen un module descriptor

→ en tiempo de compilacion y runtime todas las clases cargadas del classpath

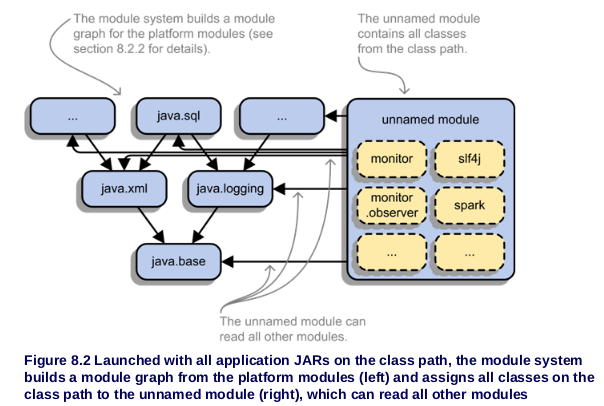
– **propiedades** del unnamed module

→ **no tiene nombre:** ningun otro modulo puede mencionarlo en sus declaraciones (xej require)

→ **dependencias**: lee todos los otros modulos que estan en el graph

→ **exports**: exporta todos los paquetes y abre todos para reflexion

– java --class-path 'jars/\*':'libs/\*' monitor.Main



→ el objetivo del unnamed module es capturar el contenido del classpath y hacerlo funcionar con el module system

→ no hay concepto de limites entre los jars, todas las clases son accesibles como antes y los split packages no existen

$ java --class-path 'mods/\*':'libs/\*' -jar monitor

$ java --module-path mods:libs --module monitor

String moduleName = Main.class.getModule().getName();

System.out.println("Module name: " + moduleName);

→ desde el classpath = null

→ desde el module path = monitor.jar

→ dentro del unnamed module NO hay restriccion de acceso a recursos en paquetes, todos los jars en el classpath pueden acceder a los recursos de los otros

→ se abren todos los paquetes para reflexion

→ encapsulacion aplica al acceder desde el unnamed a un named module's

– cuando el inicial modulo es el unnamed la resolucion de modulos comienza con el system modules que no contienen JEE apis:

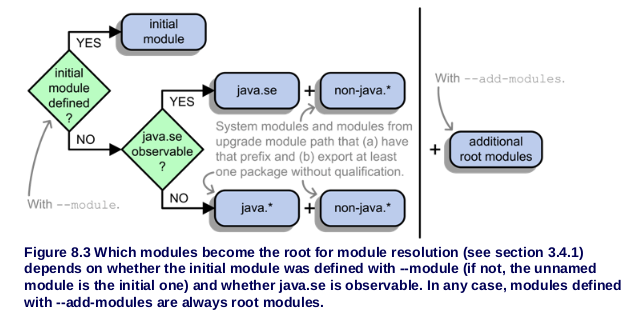
→ el preciso set de modulos java.\* que hacen el root depende de la precensia del modulo java.se (esta presente en full jre images, puede estar ausente en custom images creadas con jlink)

→ si java.se es observable es el root

→ si no lo es, todo modulo java.\* del sistema y java.\* del path de actualizables modulos que exportan paquetes sin qualificacion (no esta definido a quien explicitamente con exports … to …) es root

→ despues todo otro modulo del sistema y del upgrade-module-path que NO es un incubating module y exporta al menos un paquete sin qualificacion es root (xej modulos jdk.\*, javafx.\*)

→ luego modulos definidos con --add-modules son siempre root modules



→ todos los system modules excepto los JEE y los incubating son resueltos

→ java --show-module-resolution:

****

**/java --show-module-resolution -cp target/nimbus-1.0.0.jar com/sebastian/nimbus/AccesoNimbus**

root jdk.javadoc jrt:/jdk.javadoc

root jdk.jdi jrt:/jdk.jdi

root jdk.jshell jrt:/jdk.jshell

root jdk.sctp jrt:/jdk.sctp

root jdk.xml.dom jrt:/jdk.xml.dom

root jdk.jsobject jrt:/jdk.jsobject

root jdk.unsupported jrt:/jdk.unsupported

root jdk.scripting.nashorn jrt:/jdk.scripting.nashorn

root jdk.httpserver jrt:/jdk.httpserver

root jdk.management jrt:/jdk.management

root jdk.net jrt:/jdk.net

root jdk.security.auth jrt:/jdk.security.auth

root jdk.dynalink jrt:/jdk.dynalink

root java.se jrt:/java.se

root jdk.compiler jrt:/jdk.compiler

root jdk.accessibility jrt:/jdk.accessibility

root jdk.jartool jrt:/jdk.jartool

root jdk.jconsole jrt:/jdk.jconsole

root jdk.attach jrt:/jdk.attach

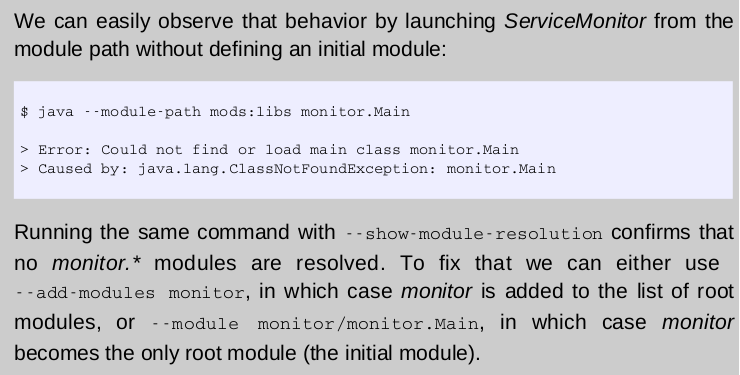
root jdk.security.jgss jrt:/jdk.security.jgss

jdk.security.jgss requires java.security.jgss jrt:/java.security.jgss

…

→ con el unnamed module como el inicial el set de root modules es un subset del system module contenidos en la runtime image

→ modulos presentes en el module path NO son resueltos a menos que se especifique --add-modules (si es necesario agregarlos todos con **--add-modules ALL-MODULE-PATH**)



**— automatic modules —**

→ el mecanismo que lo rodea aplica generalmente en tiempo de compilacion y runtime

→ por cada JAR que este en el module path y NO tenga el module descriptor el module system crea un automatic module:

: nombre = el nombre puede ser definido en el manifest del JAR con el header **Automatic-Module-Name**, si no existe genera un nombre desde el file name.

: dependencias = **lee todos los otros modulos que quedan en el graph incluyendo el unnamed module**

: exports = **exporta todos sus paquetes y tambien los abre para reflexion**

→ ejecutables JARs terminan en ejecutables modulos

→ servicios entregados en el META-INF/services quedan disponibles en el ServiceLoader

$ java

--module-path jars-mp

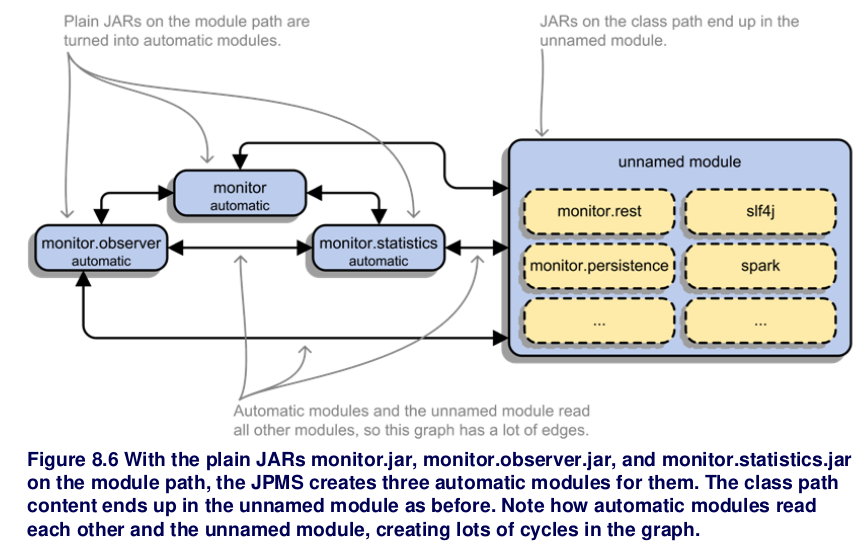
--class-path 'jars-cp/\*'

--module monitor/monitor.Main

→ jars-mp contiene los plain JARs

→ todas las otras dependencias en jars-cp

→ resulta el siguiente module graph:

- pueden usados por su nombre desde module JARs como requeridos

- **no pueden acceder a los platform moduls internals** (unnamed modules si pueden)

- se valida en split package

- el nombre puede ser definido en el MANIFEST.mf del META-INF

→ Automatic-Module-Name es la clave que tiene que tener el valor del nombre que se quiere usar

→ sin no está presente la clave del nombre el JPMS lo genera (version string es ignorado, cada carácter que no es letra o digito se reemplaza por .)

→ para conocer el nombre que tendrá como automatic module: **jar --describe-module –file guava-20.0.jar**

→ al no encontrar el module descriptor ni el nombre en el MANIFEST genera uno automatico: guava

– el proyecto no requiere ser compilado con java 9, puede ser compilado con versiones anteriores e incluir el nombre en el MANIFEST

- para construir el module graph en los explicit modules se sigue el requires

→ pero los plain JARs del module path NO pueden expresar dependencias

– automatic modules fueron creados para modularizacion que NO tiene modulares dependencias

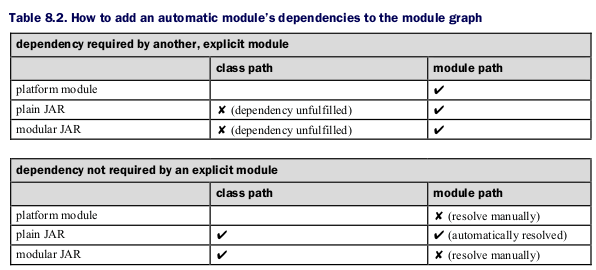
– JPMS pone TODOS los automatic modules una vez que encuentra el primero

→ si un proyecto tiene 2 dependencias de automatic modules, puede definir requires 1 de ellos, y el JPMS igual agregara TODOS los automatic modules (pero tiene que estar definida al menos 1)

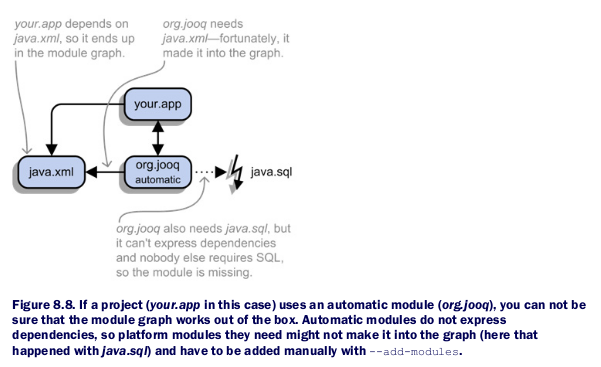
– automatic modules leen TODOS los otros modulos

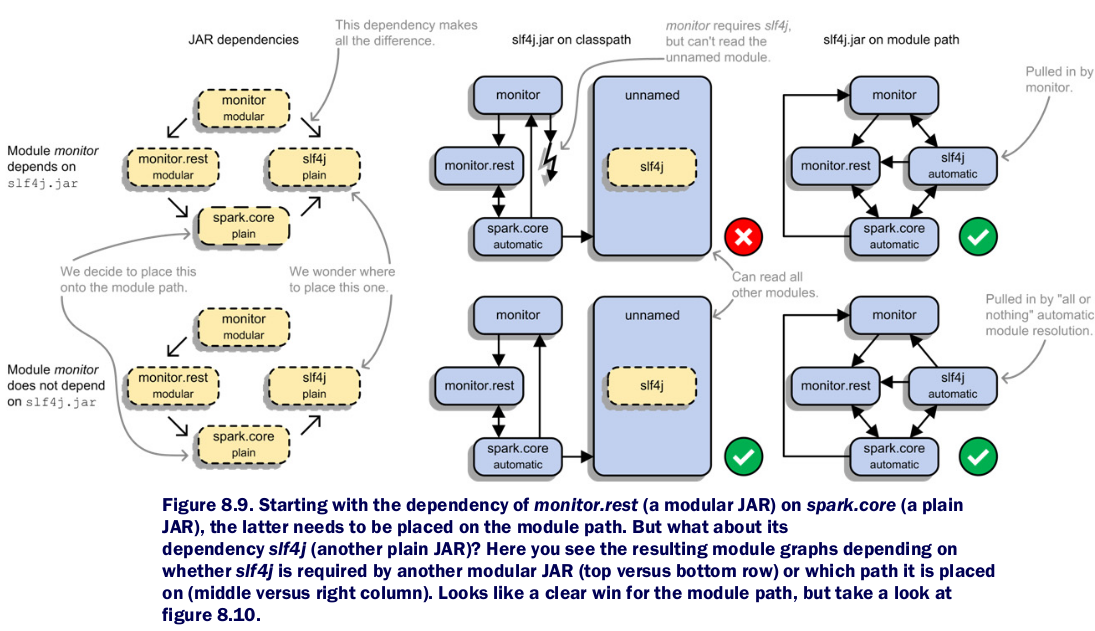
– como los automatic modules leen todos los modulos y hay modulos definidos que los requieren crean **dependencias ciclicas**, PERO son NO estaticamente declarados, fueron introducidos por el module system, NO generar error

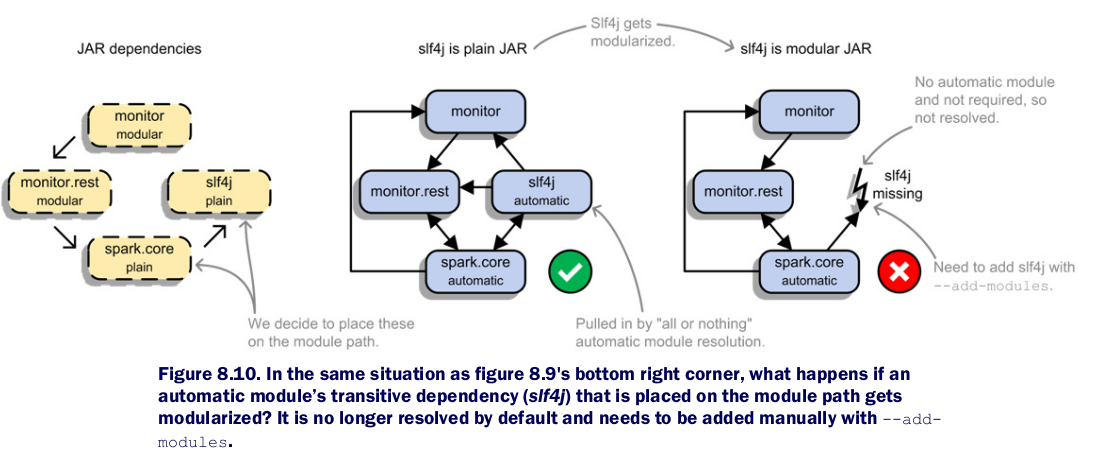
– las dependencias de los automatic modules pueden estar en el unnamed modules o en el module path



– automatic modules NO expresan dependencias, platform modules que necesitan pueden no terminar en el module graph, tienen que ser agregadas con --add-modules:







– como los automatic modules NO pueden definir dependencias no pueden decir cuales son los platform modules que requieren y pueden quedar fuera del module graph

– si las dependencias de un automatic module son requeridas por un explicit module tienen que ser colocadas en el module path y seran resueltas

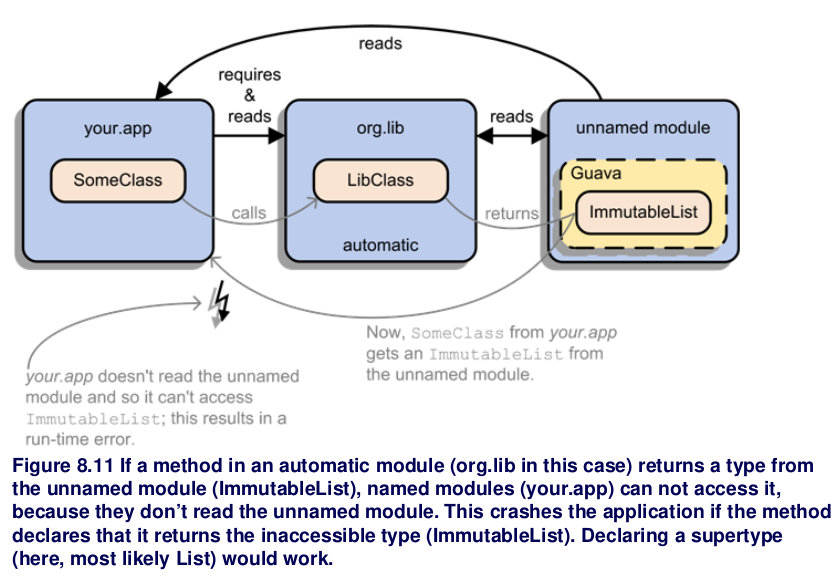
→ si no hay un modulo explicito que los requiera tienen que ser colocadas en el classpath para que sean cargadas en el unnamed module y sean accesibles

→ si se colocan en el module path hay que suponer que se cargara algun otro automatic module (se cargarian todos)

→ platform y explicit modules NO son cargados por defecto, si son requeridos por otro modulo usar --add-modules

**— named modules no leen el unnamed module**

→ leen del automatic module



→ si un automatic module expone tipos de otro jar, el otro jar tiene que estar en el module path

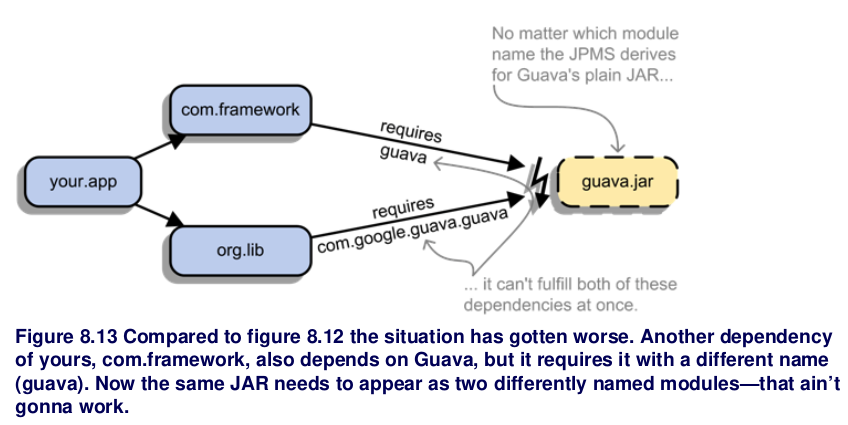
→ si no lo hace los tipos seran inaccesibles para los explicit modules = IllegalAccessError no tienen acceso de lectura

→ se puede usar **--add-reads ALL-UNNAMED** para acoplar el modulo al unnamed module

→ **--add-reads app=ALL-UNNAMED** permitiria poder usar ImmutableList desde el unnamed module

– el mismo JAR puede tener distintos nombres en distintos proyectos

– un unico JAR NO puede cumplir con los requerimientos de distintos nombres y multiples JAR con distinto contenido = split package



– cualquier JAR en el classpath, plano o modular termina en el unnamed module

– cualquier JAR en el module path termina como un named module (plain jar o explicit module)

– el unnamed module es una caracteristica de compatibilidad que hace al module sistem funcionar con el classpath

→ captura el contenido del classpath

→ no tiene nombre

→ lee todos los otros modulos

→ exporta todos sus paquetes

→ abre todos sus paquetes

→ modulos explicitos NO pueden referir al unnamed module en su declaracion (no tiene nombre), no pueden leerlo y no pueden usar sus tipos

→ sin mayor configuración lee todos los Java SE APIs (no jee)

– automatic modules son para migraciones, permiten a modulos depender de plain jars

→ un automatic module es creado por cada JAR en el module path

→ su nombre puede ser definido en el Automatic-Module-Name en el MANIFEST

→ si no esta definido el module system crea uno desde su file name

→ lee todos los otros modulos, incluyendo el unnamed

→ exporta todos sus paquetes

→ abre todos sus paquetes

→ puede ser referenciado por module declarations

→ sus dependencias pueden ser colocadas en el classpath o el module path (dejar las modulares en el module path y las otras en el classpath xdefecto)

→ cuando se resuelve el primero todos los otros se agregan

→ si un modulo lee un automatic puede leer todos los demas automatic modules

### *Increasing readability*

When testing and debugging it is sometimes necessary to arrange for one module to read some other module, even though the first module does not depend upon the second via a requires clause in its module declaration. This may be needed to, e.g., enable a module under test to access the test harness itself, or to access libraries related to the harness. The --add-reads option can be used, at both compile time and run time, to do this. Its syntax is:

--add-reads <source-module>=<target-module>

where <source-module> and <target-module> are module names.

The --add-reads option can be used more than once. The effect of each instance is to add a [readability edge](https://openjdk.java.net/projects/jigsaw/spec/sotms/#readability) from the source module to the target module. This is, essentially, a command-line form of a requires clause in a module declaration, or an invocation of an unrestricted form of the Module::addReads [method](http://download.java.net/java/jdk9/docs/api/java/lang/reflect/Module.html#addReads-java.lang.reflect.Module-). As a consequence, code in the source module will be able to access types in a package of the target module at both compile time and run time if that package is exported via an exports clause in the source module's declaration, an invocation of the Module::addExports [method](http://download.java.net/java/jdk9/docs/api/java/lang/reflect/Module.html#addExports-java.lang.String-java.lang.reflect.Module-), or an instance of the --add-exports option (defined below). Such code will, additionally, be able to access types in a package of the target module at run time if that module is declared to be open or if that package is opened via an opens clause in the source module's declaration, an invocation of the Module::addOpens [method](http://download.java.net/java/jdk9/docs/api/java/lang/reflect/Module.html#addOpens-java.lang.String-java.lang.reflect.Module-), or an instance of the --add-opensoption (also defined below).

If, for example, a test harness injects a white-box test class into thejava.management module, and that class extends an exported utility class in the (hypothetical) testng module, then the access it requires can be granted via the option

--add-reads java.management=testng

As a special case, if the <target-module> is ALL-UNNAMED then readability edges will be added from the source module to all present and future unnamed modules, including that corresponding to the class path. This allows code in modules to be tested by test frameworks that have not, themselves, yet been converted to modular form.

### *Breaking encapsulation*

It is sometimes necessary to violate the access-control boundaries defined by the module system, and enforced by the compiler and virtual machine, in order to allow one module to access some of the unexported types of another module. This may be desirable in order to, e.g., enable white-box testing of internal types, or to expose unsupported internal APIs to code that has come to depend upon them. The --add-exports option can be used, at both compile time and run time, to do this. Its syntax is:

--add-exports <source-module>/<package>=<target-module>(,<target-module>)\*

where <source-module> and <target-module> are module names and <package>is the name of a package.

The --add-exports option can be used more than once, but at most once for any particular combination of source module and package name. The effect of each instance is to add a [qualified export](https://openjdk.java.net/projects/jigsaw/spec/sotms/#qualified-exports) of the named package from the source module to the target module. This is, essentially, a command-line form of an exportsclause in a module declaration, or an invocation of an unrestricted form of theModule::addExports [method](http://download.java.net/java/jdk9/docs/api/java/lang/reflect/Module.html#addExports-java.lang.String-java.lang.reflect.Module-). As a consequence, code in the target module will be able to access public types in the named package of the source module if the target module reads the source module, either via a requires clause in its module declaration, an invocation of the Module::addReads method, or an instance of the--add-reads option.

If, for example, the module jmx.wbtest contains a white-box test for the unexported com.sun.jmx.remote.internal package of the java.managementmodule, then the access it requires can be granted via the option

--add-exports java.management/com.sun.jmx.remote.internal=jmx.wbtest

As a special case, if the <target-module> is ALL-UNNAMED then the source package will be exported to all unnamed modules, whether they exist initially or are created later on. Thus access to the sun.management package of the java.managementmodule can be granted to all code on the class path via the option

--add-exports java.management/sun.management=ALL-UNNAMED

The --add-exports option enables access to the public types of a specified package. It is sometimes necessary to go further and enable access to all non-public elements via the setAccessible [method](http://download.java.net/java/jdk9/docs/api/java/lang/reflect/AccessibleObject.html#setAccessible-boolean-) of the [core reflection API](http://download.java.net/java/jdk9/docs/api/java/lang/reflect/package-summary.html). The --add-opens option can be used, at run time, to do this. It has the same syntax as the --add-exports option:

--add-opens <source-module>/<package>=<target-module>(,<target-module>)\*

where <source-module> and <target-module> are module names and <package>is the name of a package.

The --add-opens option can be used more than once, but at most once for any particular combination of source module and package name. The effect of each instance is to add a qualified open of the named package from the source module to the target module. This is, essentially, a command-line form of an opens clause in a module declaration, or an invocation of an unrestricted form of the Module::addOpens [method](http://download.java.net/java/jdk9/docs/api/java/lang/reflect/Module.html#addOpens-java.lang.String-java.lang.reflect.Module-). As a consequence, code in the target module will be able to use the core reflection API to access all types, public and otherwise, in the named package of the source module so long as the target module reads the source module.

Open packages are indistinguishable from non-exported packages at compile time, so the --add-opens option may not be used in that phase.

The --add-exports and --add-opens options must be used with great care. You can use them to gain access to an internal API of a library module, or even of the JDK itself, but you do so at your own risk: If that internal API is changed or removed then your library or application will fail.

### *Patching module content*

When testing and debugging it is sometimes useful to replace selected class files or resources of specific modules with alternate or experimental versions, or to provide entirely new class files, resources, and even packages. This can be done via the --patch-module option, at both compile time and run time. Its syntax is:

--patch-module <module>=<file>(<pathsep><file>)\*

where <module> is a module name, <file> is the filesystem path name of a module definition, and <pathsep> is the host platform's path-separator character.

The --patch-module option can be used more than once, but at most once for any particular module name. The effect of each instance is to change how the module system searches for a type in the specified module. Before it checks the actual module, whether part of the system or defined on a module path, it first checks, in order, each module definition specified to the option. A patch path names a sequence of module definitions but it is not a module path, since it has leaky, class-path-like semantics. This allows a test harness, e.g., to inject multiple tests into the same package without having to copy all of the tests into a single directory.

The --patch-module option cannot be used to replace module-info.class files. If a module-info.class file is found in a module definition on a patch path then a warning will be issued and the file will be ignored.

If a package found in a module definition on a patch path is not already exported or opened by that module then it will, still, not be exported or opened. It can be exported or opened explicitly via either the reflection API or the --add-exports or --add-opens options.

The --patch-module option replaces the -Xbootclasspath:/p option, which has been removed (see below).

The --patch-module option is intended only for testing and debugging. Its use in production settings is strongly discouraged.

### *Compile time*

The javac compiler implements the options described above, as applicable to compile time: --module-source-path, --upgrade-module-path, --system, --module-path, --add-modules, --limit-modules, --add-reads, --add-exports, and --patch-module.

The compiler operates in one of three modes, each of which implements additional options.

* Legacy mode is enabled when the compilation environment, as defined by the -source, -target, and --release options, is less than or equal to 8. None of the modular options described above may be used.

In legacy mode the compiler behaves in essentially the same way as it does in JDK 8.

* Single-module mode is enabled when the compilation environment is 9 or later and the --module-source-path option is not used. The other modular options described above may be used; the existing options -bootclasspath, -Xbootclasspath, -extdirs, -endorseddirs, and -XXuserPathsFirst may not be used.

Single-module mode is used to compile code organized in a traditional package-hierarchical directory tree. It is the natural replacement for simple uses of legacy mode of the form

$ javac -d classes -classpath classes -sourcepath src Foo.java

If a module descriptor in the form of a module-info.java or module-info.classfile is specified on the command line, or is found on the source path or the class path, then source files will be compiled as members of the module named by that descriptor and that module will be the sole root module. Otherwise if the --module <module> option is present then source files will be compiled as members of <module>, which will be the root module. Otherwise source files will be compiled as members of the unnamed module, and the root modules will be computed [as described above](https://openjdk.java.net/jeps/261#root-modules).

It is possible to put arbitrary classes and JAR files on the class path in this mode, but that is not recommended since it amounts to treating those classes and JAR files as part of the module being compiled.

* Multi-module mode is enabled when the compilation environment is 9 or later and the --module-source-path option is used. The existing -d option to name the output directory must also be used; the other modular options described above may be used; the existing options -bootclasspath, -Xbootclasspath, -extdirs, -endorseddirs, and -XXuserPathsFirst may not be used.

Multi-module mode is used to compile one or more modules, whose source code is laid out in exploded-module directories on the module source path. In this mode the module membership of a type is determined by the position of its source file in the module source path, so each source file specified on the command line must exist within an element of that path. The set of root modules is the set of modules for which at least one source file is specified.

In contrast to the other modes, in this mode an output directory must be specified via the -d option. The output directory will be structured as an element of a module path, i.e., it will contain exploded-module directories which themselves contain class and resource files. If the compiler finds a module on the module source path but cannot find the source file for some type in that module then it will search the output directory for the corresponding class file.

In large systems the source code for a particular module may be spread across several different directories. [In the JDK itself](https://openjdk.java.net/jeps/201), e.g., the source files for a module may be found in any one of the directories src/<module>/share/classes, src/<module>/<os>/classes, or build/gensrc/<module>, where <os> is the name of the target operating system. To express this in a module source path while preserving module identities we allow each element of such a path to use braces ({ and }) to enclose commas-separated lists of alternatives and a single asterisk (\*) to stand for the module name. The module source path for the JDK can then be written as

{src/\*/{share,<os>}/classes,build/gensrc/\*}

In both of the modular modes the compiler will, by default, generate various warnings related to the module system; these may be disabled via the option -Xlint:-module. More precise control of these warnings is available via the exports, opens, requires-automatic, and requires-transitive-automatickeys for the -Xlint option.

The new option --module-version <version> may be used to specify the version strings of the modules being compiled.