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

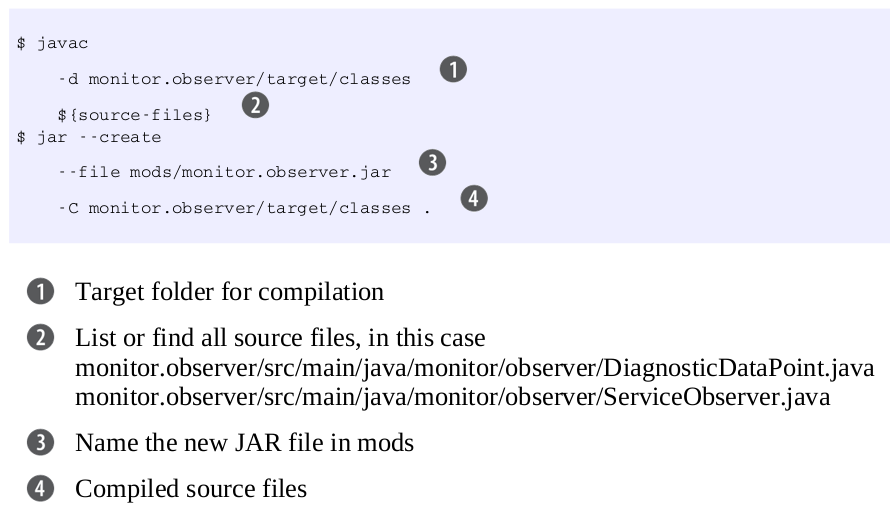
Inside the module block, requires and exports directives define dependencies and exported packages

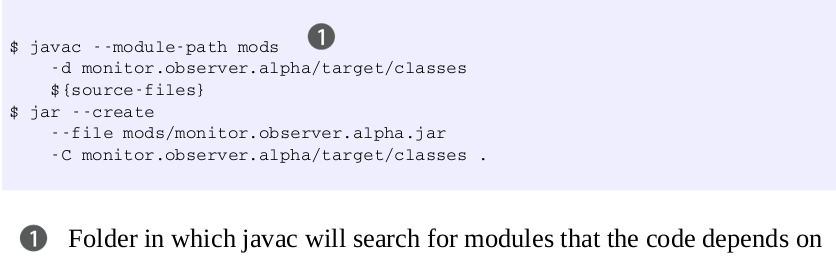
A requires directive contains a module name and tells the JVM that the declaring module depends on the one given by the directive.

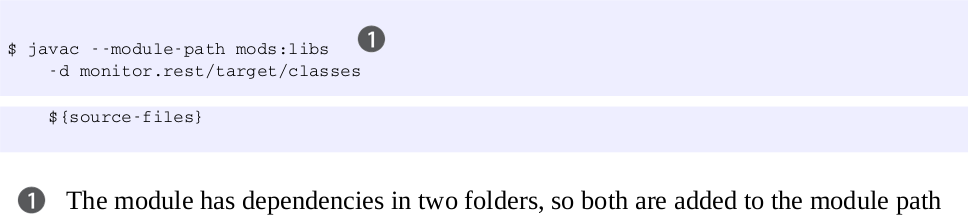
the module system forbids declaring cyclic dependencies.

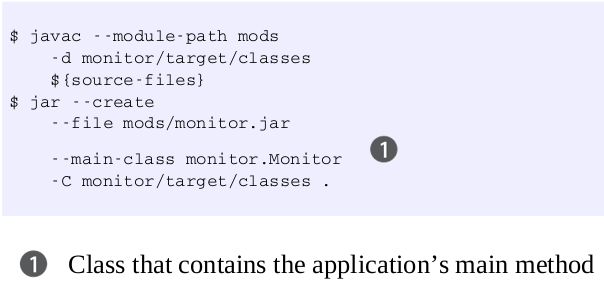
An exports directive contains a package name and informs the JVM that other modules depending on the declaring one can see public types in

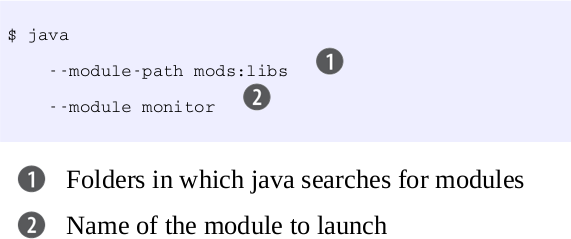
that package.











**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:

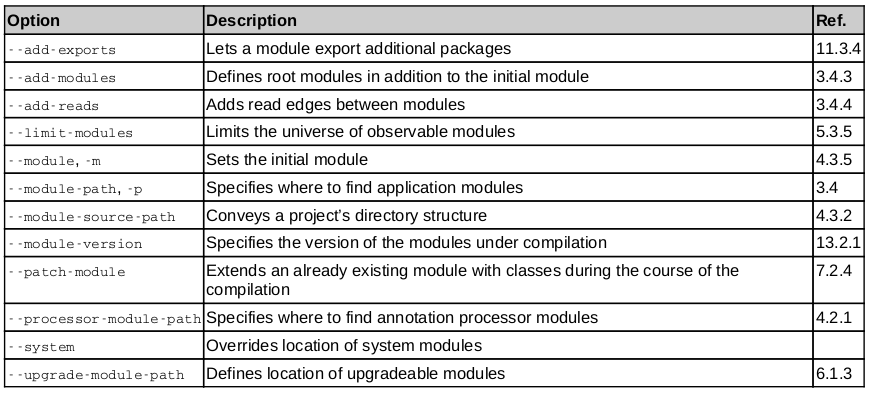
– ModuleFinder before: consultado para ubicar modulos antes de mirar en el parent configuration

– 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/ .

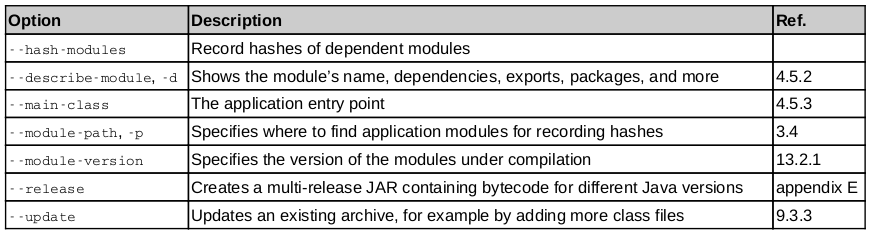
4 → java --module-path target/comun.jar --module modulos.comun/com.sebastian.modulos.comun.Main

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

–

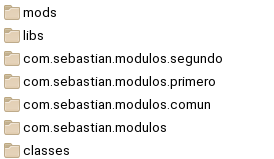
The java command has an option --module ${module} , which specifies the initial module ${module} . Module resolution will start from there and it is also the one from which a main class, meaning one with a public static void main method, will be launched. The specific class is either defined by the initial 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)

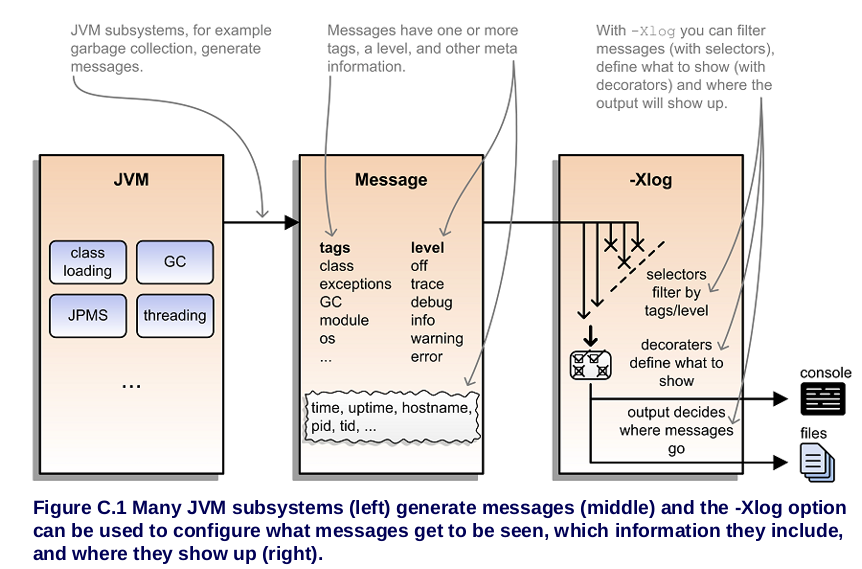
– **-Xlog** = **logs de la JVM** (NO de la JDK → no de swing, ni de la aplicación que se esta lanzando)

→ 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

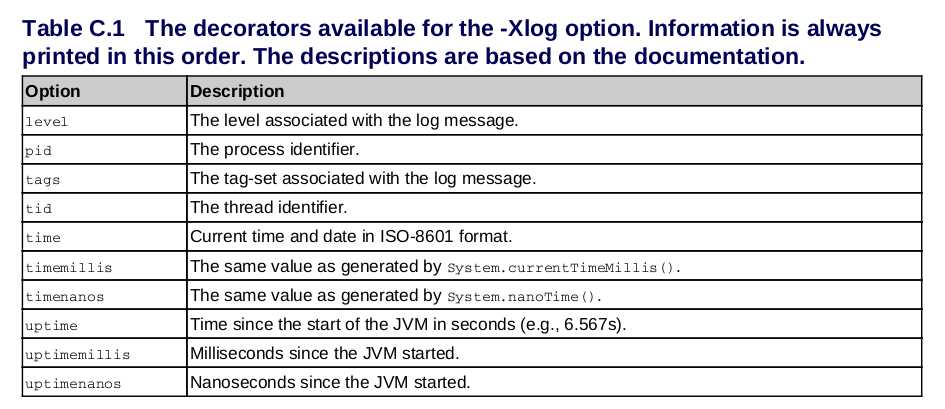
→ **:stderr** → salida de error

→ 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>

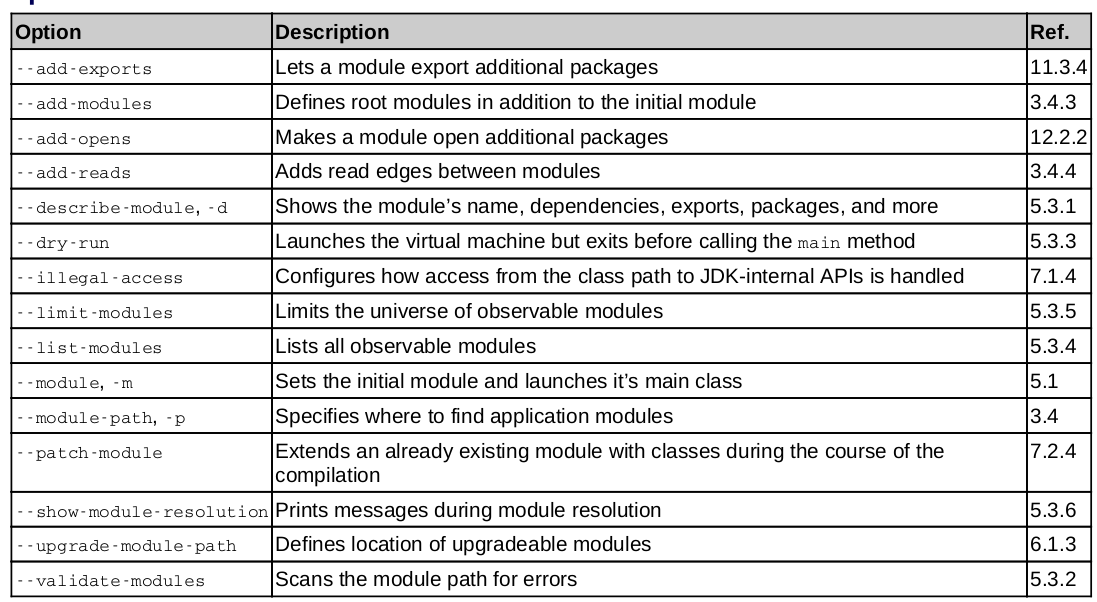
→ los parametros son opcionales PERO si se usa uno HAY que usar los anteriores

**[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" \l "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.

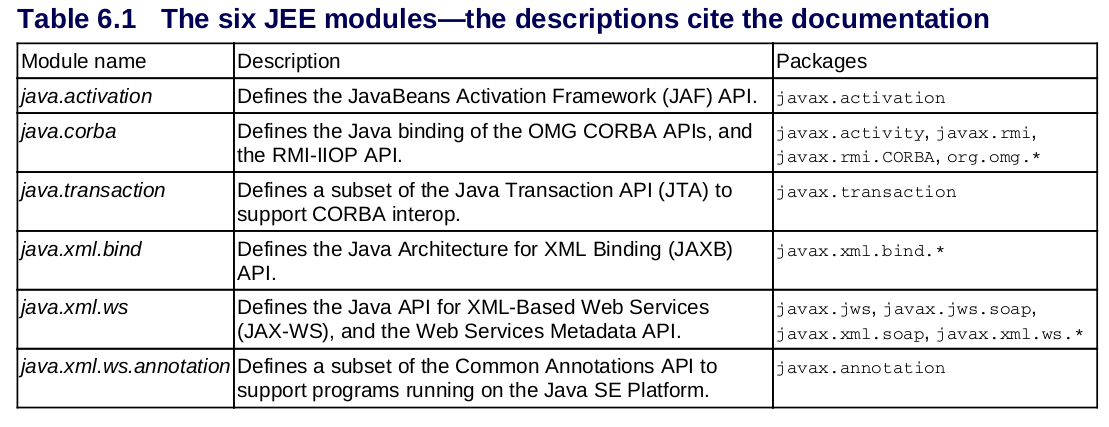
– A module can use opens directives to give reflective access to packages, which exposes resources located therein and thus allows other modules to load them. Unfortunately, this solution invites other code to depend on the module’s internal structure.

– 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 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

