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

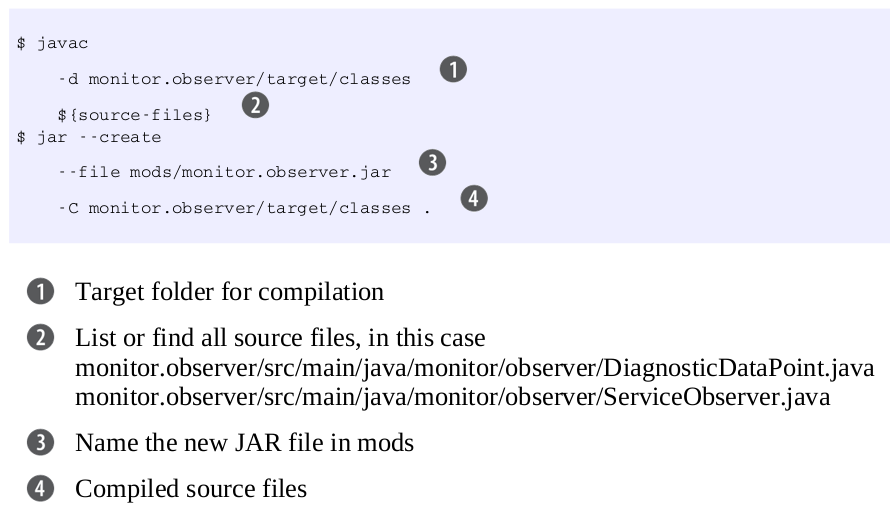
Inside the module block, requires and exports directives define dependencies and exported packages

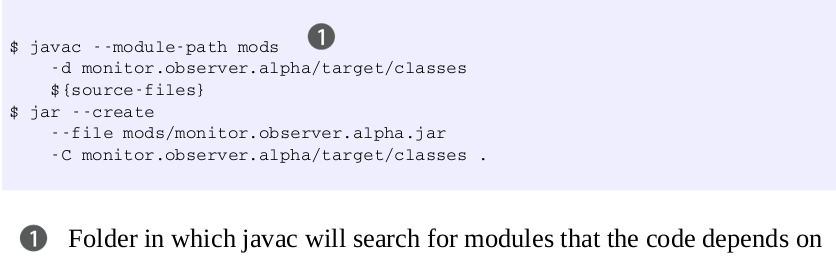
A requires directive contains a module name and tells the JVM that the declaring module depends on the one given by the directive.

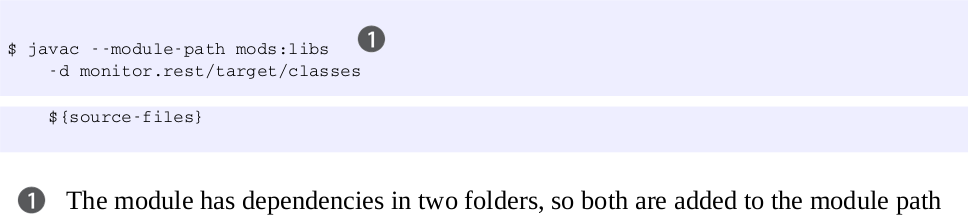
the module system forbids declaring cyclic dependencies.

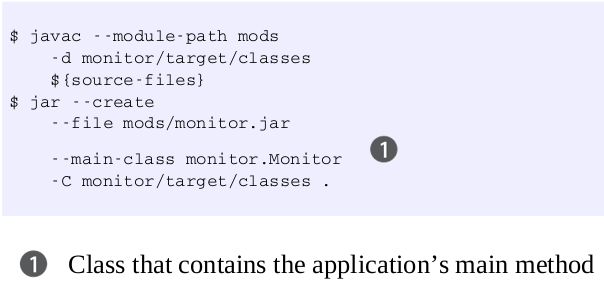
An exports directive contains a package name and informs the JVM that other modules depending on the declaring one can see public types in

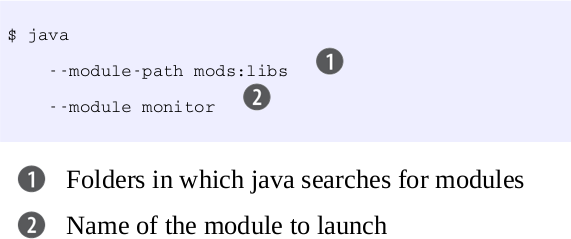
that package.











**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:

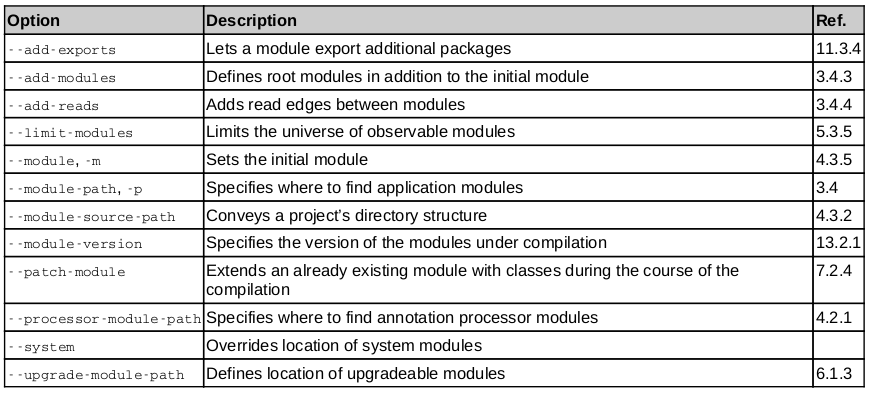
– ModuleFinder before: consultado para ubicar modulos antes de mirar en el parent configuration

– 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/ .

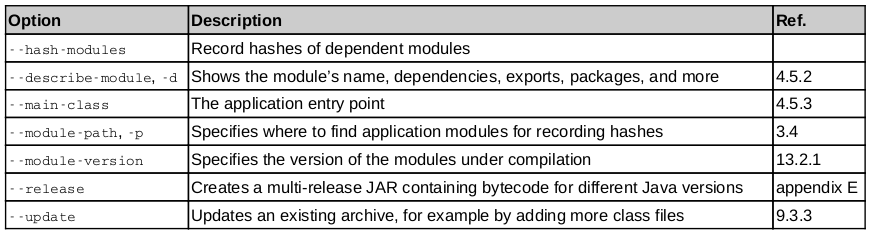
4 → java --module-path target/comun.jar --module modulos.comun/com.sebastian.modulos.comun.Main

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

–

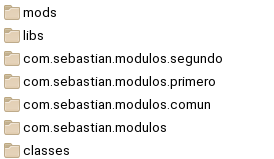
The java command has an option --module ${module} , which specifies the initial module ${module} . Module resolution will start from there and it is also the one from which a main class, meaning one with a public static void main method, will be launched. The specific class is either defined by the initial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

–