Las API de Java se organizan en métodos, clases, paquetes y, en el nivel más alto - módulos. Un módulo contiene una serie de datos esenciales:

- un nombre
- una lista de dependencias de otros módulos
- una API pública (siendo todo lo demás módulo interno e inaccesible)•

una lista de servicios que utiliza y proporciona

No sólo las API de Java vienen con esta información, también tienes la opción de crear módulos para tus propios proyectos. Al desplegar su proyecto como módulos, usted aumentan la fiabilidad y la capacidad de mantenimiento, evitan el uso accidental de API internas y pueden crear más fácilmente imágenes de tiempo de ejecución que contengan sólo el código JDK que necesita (con la opción de incluir su propia aplicación en la imagen para hacerla independiente).

Antes de hablar de estas ventajas, veremos cómo definir un módulo y sus funciones. cómo convertirlo en un JAR entregable, y cómo los maneja el sistema de módulos. Para hacerlo más fácil, vamos a suponer que todo (código en el JDK, bibliotecas, frameworks, aplicaciones) es un módulo - por qué no es necesario y cómo se pueden crear módulos de forma incremental se trata en artículos dedicados.

Declaraciones de módulos

En el núcleo de cada módulo se encuentra la *declaración del módulo*, un archivo con el nombre module- info.java que define todas las propiedades de un módulo. Como ejemplo, aquí está la de java.sql, el módulo de plataforma que define la API JDBC:

```
1  módulo java.sql {
2      requiere java.logging;
3      requiere java.transaction.xa transitivo;
4      requiere java.xml;
5
6      exporta java.sql;
7      exporta javax.sql;
8
9      utiliza java.sql.Driver;
10 }
```

Define el nombre del módulo (*java.sql*), sus dependencias de otros módulos (java.logging, java.transaction.xa, java.xml), los paquetes que componen su API pública (java.sql y javax.sql) y los servicios que utiliza (java.sql.Driver). Este ya emplea una forma más refinada para definir dependencias añadiendo la palabra clave transitive, pero, por supuesto, no utiliza todas las capacidades del sistema de módulos. En general, una declaración de módulo tiene la siguiente forma básica:

```
módulo $NAME {
    // para cada
    dependencia: requiere
    $MODULE;
    // para cada paquete API:
    exporta $PAQUETE

    // para cada paquete destinado a la reflexión:
    opens $PAQUETE;

    // para cada servicio utilizado:
```

```
        utiliza $TYPE;

        // para cada servicio proporcionado:
        proporciona $TYPE con $CLASS;
    }
```

(Todo el módulo puede ser abierto y las directivas requires, exports y opens pueden ser más refinadas, pero esos son temas para más adelante).

Puedes crear declaraciones de módulos para tus proyectos. Su ubicación recomendada -donde todas las herramientas las recogerán más fácilmente- es en la carpeta raíz del proyecto, es decir, la carpeta que contiene los directorios del paquete, a menudo src/main/java. Para una biblioteca, una declaración de módulo podría tener este aspecto:

```
módulo com.example.lib {
    requiere java.sql;
    requiere com.sample.other;

    exporta com.example.lib;
    exporta com.example.lib.db;

    utiliza
    com.example.lib.Service;
}
```

Para una aplicación, podría ser algo así:

```
módulo com.example.app {
    requiere com.example.lib;

    abre com.example.app.entities;

    proporciona
    com.example.lib.Service
        con com.example.app.MyService;
}
```

Repasemos rápidamente los detalles. Esta sección se centra en lo que hay que incluir en la declaración del módulo:

- nombre del módulo
- dependencias
- paquetes exportados
- servicios utilizados y prestados

Los efectos se analizan en una sección posterior.

Nombre del módulo

Los nombres de módulo tienen los mismos requisitos y directrices que los nombres de paquete:

- las letras legales incluyen A-Z, a-z, 0-9, _ y $,
- separadas por .
- por convención, los nombres de los módulos se escriben todos en minúsculas y $ sólo se utiliza para el código generado mecánicamente

- los nombres deben ser globalmente únicos

En el ejemplo anterior, la declaración del módulo JDK comenzaba con module java.sql, que definía el módulo con el nombre *java.sql*. Los dos módulos personalizados eran

llamados com.example.lib y com.example.app.

Dado un JAR, el nombre del módulo correspondiente puede deducirse del archivo con un vistazo al archivo module-info.class en el JAR (más sobre esto más adelante), con la ayuda de un IDE, o ejecutando jar --describe-module --file $FILE contra el archivo JAR.

En cuanto a la unicidad del nombre del módulo, la recomendación es la misma que para paquetes: Elige una URL asociada al proyecto e inviértela para obtener la primera parte del nombre del módulo, luego refina a partir de ahí. (Esto implica que los dos módulos de ejemplo están asociados al dominio ejemplo.com). Si aplica esta a los nombres de módulos y paquetes, el primero suele ser un prefijo del segundo porque los módulos son más generales que los paquetes. Esto no significa en absoluto necesario, sino un indicador de que los nombres se eligieron bien.

Exigir dependencias

Las directivas requires listan todas las dependencias directas por sus nombres de módulo. Echa un vistazo a las de los tres módulos anteriores:

```
// de java.sql, pero sin `transitive`.
requiere java.logging;
requiere java.transaction.xa;
requiere java.xml;

// de com.example.lib
requiere java.sql;
requiere com.sample.other;

// de com.example.app
requiere com.example.lib;
```

Podemos ver que el módulo de aplicación *com.example.app* depende de la biblioteca *com.example.lib*, que a su vez necesita el módulo no relacionado com.sample.*other* y el módulo de plataforma java.*sql*. No tenemos la declaración de com.sample.*other* pero sabemos que *java.sql* depende de *java.logging*, *java.transaction.xa* y *java.xml*. Si los buscamos, veremos que no tienen más dependencias. (O mejor dicho, no tienen dependencias explícitas - consulta la sección sobre el módulo base más abajo para más detalles).

También es posible manejar dependencias opcionales (con requires static) y "reenviar" dependencias que forman parte de la API de un módulo (con requires transitive), pero eso se trata en artículos aparte.

La lista de dependencias externas es, con toda probabilidad, muy similar a las dependencias enumeradas en su configuración de compilación. Esto suele llevar a preguntarse si se trata de
redundante y debe ser auto-generado. No es redundante porque un nombre de módulo no contiene versión o cualquier otra información que una herramienta de compilación necesite para obtener un JAR (como ID de grupo e ID de artefacto) y las configuraciones de compilación listan esa información pero no el nombre de un módulo. Debido a que el nombre del módulo puede ser inferido dado un JAR es
posible generar esta sección de module-info.java. No está claro si
Sin embargo, el esfuerzo merece la pena, sobre todo teniendo en cuenta la complejidad añadida de las dependencias de los módulos de la plataforma y de los

modificadores estáticos y transitivos, así como el hecho de que los IDE ya proponen añadir directivas requires cuando son necesarias (de forma parecida a las importaciones de paquetes), lo que hace que actualizar las declaraciones de los módulos sea muy sencillo.

Exportación y apertura de paquetes

Por defecto, todos los tipos, incluso los públicos, sólo son accesibles dentro de un módulo. Para que código fuera de un módulo acceda a un tipo, el paquete que contiene el tipo necesita ser *exportado* o *abierto*. Esto se consigue utilizando las funciones exports y opens que incluyen el nombre del paquete que contiene el módulo. El efecto exacto de la exportación se discute en la sección sobre encapsulación fuerte a continuación, de la apertura en un artículo sobre la reflexión, pero la esencia es:

- los tipos y miembros públicos de los paquetes exportados están disponibles en tiempo de compilación y de ejecución
- se puede acceder a todos los tipos y miembros de los paquetes abiertos en tiempo

de ejecución a través de reflection Aquí están las directivas exports y opens de los tres módulos de ejemplo:

```
// de módulo java.sql
exportaciones java.sql;
exportaciones javax.sql;

// de com.example.lib
exportaciones com.example.lib;
exportaciones com.example.lib.db;

// de com.example.app
abre com.example.app.entities
```

Esto muestra que *java.sql* exporta un paquete del mismo nombre, así como javax.sql - el módulo contiene muchos más paquetes, por supuesto, pero no son parte de su API y no nos conciernen. El módulo library exporta dos paquetes para ser usados por otros módulos - de nuevo, todos los demás paquetes (potenciales) están a buen recaudo. La aplicación
no exporta paquetes, lo que no es raro ya que el módulo que lanza la aplicación raramente es una dependencia de otros módulos y por lo tanto nadie llama a él.
Sin embargo, abre com.example.app.entities para la reflexión - a juzgar por el nombre, probablemente porque contiene entidades con las que otros módulos quieren interactuar a través de la reflexión (piensa en JPA).

También existen variantes cualificadas de las directivas exports y opens que permiten exportar/abrir un paquete sólo a módulos específicos.

Como regla general, intenta exportar el menor número de paquetes posible, como mantener los campos privados, hacer que los métodos sólo sean visibles desde el paquete o públicos si es necesario, y hacer que las clases sean visibles desde el paquete por defecto y sólo públicas si se necesitan en otro paquete. Esto reduce la cantidad de código que es visible en otro lugar, lo que reduce la complejidad.

Utilización y prestación de servicios

Los servicios son un tema aparte; por ahora basta decir que puede utilizarlos para desvincular al usuario de una API de su implementación, lo que facilita su sustitución a medida que tarde como al lanzar la aplicación. Si un módulo utiliza un tipo (una interfaz o una clase)

como servicio, debe indicarlo en la declaración del módulo con la directiva uses, que incluye el nombre completo del tipo. Los módulos que proporcionan un servicio expresan en su declaración de módulo cuáles de sus propios tipos lo hacen (normalmente implementándolo o extendiéndolo).

Los módulos de ejemplo de la biblioteca y la aplicación muestran las dos caras:

```
// en com.example.lib
utiliza
com.example.lib.Service;
// en el módulo com.example.app
proporciona
com.example.lib.Service
    con com.example.app.MyService;
```

El módulo lib utiliza Service, uno de sus propios tipos, como servicio y el módulo app, que depende del módulo lib, le proporciona MyService. En tiempo de ejecución, el módulo lib
accederá a todas las clases que implementen / extiendan el tipo de servicio utilizando la API ServiceLoader con una llamada tan simple como ServiceLoader.load(Service.class).
Esto significa que el módulo de la librería ejecuta el comportamiento definido en el módulo de la aplicación aunque no dependa de él - esto es genial para desenredar las dependencias y mantener los módulos centrados en sus preocupaciones.

Creación y lanzamiento de módulos

La declaración del módulo module-info.java es un archivo fuente como cualquier otro y, por tanto, sigue unos pasos antes de ejecutarse en la JVM. Afortunadamente, estos son exactamente los mismos pasos que toma su código fuente y la mayoría de las herramientas de compilación y los IDEs entienden eso bien
suficiente para adaptarse a su presencia. Lo más probable es que no necesites hacer nada manualmente para construir y lanzar una base de código modular. Por supuesto, hay valor en
para entender los detalles, por lo que un artículo dedicado le llevará desde el código fuente hasta la ejecución de JVM con sólo herramientas de línea de comandos.

Aquí, nos quedaremos en un nivel más alto de abstracción y en su lugar discutiremos algunos conceptos que juegan un papel importante en la construcción y ejecución de código modular:

- modular JARs
- module path
- resolución del módulo y gráfico del módulo
- el módulo base

JAR modulares

Un archivo module-info.java (también conocido como declaración de módulo) se compila en module-info.class (llamado descriptor *de módulo*), que puede colocarse en el directorio raíz de un JAR o en un directorio específico de la versión, si se trata de un JAR multilanzamiento. Un JAR que contiene un descriptor de módulo se denomina *JAR modular* y está listo para ser utilizado como módulo; los JAR sin descriptor son *JAR simples*. Si un JAR modular se coloca en la ruta de módulos (ver más abajo), se convierte en un módulo en tiempo de ejecución, pero también se puede utilizar en la ruta de clases,
donde pasa a formar parte del módulo sin nombre al igual que los JAR normales en la ruta de clases.

Módulo Ruta

La *ruta de módulo* es un nuevo concepto paralelo a la ruta de clase: Es una lista de artefactos (JARs o carpetas de código de bytes) y directorios que contienen artefactos. El sistema de módulos
lo utiliza para localizar los módulos necesarios que no se encuentran en el tiempo de ejecución, por lo que normalmente todos los módulos de aplicaciones, bibliotecas y marcos. Convierte todos los artefactos de la ruta de módulos en módulos, incluso los JAR simples, que se convierten en módulos automáticos, lo que permite que

modularización incremental. Tanto javac y java como otros comandos relacionados con módulos entienden y procesan la ruta del módulo.

Nota al margen: Esta sección y la anterior juntas han revelado un comportamiento posiblemente sorprendente del sistema de módulos: El hecho de que un JAR sea modular o no no determina que sea tratado como un módulo. Todos los JAR de la ruta de clases se tratan como un único barely-a-module, todos los JAR de la ruta del módulo se convierten en módulos. Eso significa que la persona a cargo de un proyecto decide qué dependencias terminan como módulos individuales y cuáles no (a diferencia de los mantenedores de las dependencias).

Resolución del módulo y gráfico del módulo

Para lanzar una aplicación modular, ejecute el comando java con una ruta de módulo y una llamada *módulo inicial* - el módulo que contiene el método principal:

```
# los módulos están en `app-jars` | el módulo inicial es `com.example.app`
java --module-path app-jars --module com.example.app
```

Esto iniciará un proceso llamado *resolución de módulos*: Comenzando con el nombre del módulo inicial, el sistema de módulos lo buscará en la ruta de módulos. Si lo encuentra, comprobará sus directivas requires para ver qué módulos necesita y repetirá el proceso para ellos. Si no encuentra un módulo, lanzará un error en ese momento, permitiéndote
saber que falta una dependencia. Puede observar este proceso añadiendo la opción de línea de comandos --show-module-resolution.

El resultado de este proceso es el *grafo de módulos*. Sus nodos son módulos, sus aristas son un poco más complicadas: Cada directiva require genera una arista entre los dos módulos.
módulos, lo que se denomina un *borde de legibilidad* en el que el módulo solicitante *lee* al requerido. Hay otras formas de crear bordes, pero eso no tiene por qué preocuparnos ahora porque no cambia nada fundamental.

Si imaginamos un programa Java normal, por ejemplo un backend para una aplicación web, podemos imaginar su gráfico de módulos: En la parte superior encontraremos el módulo inicial, más abajo los otros módulos de la aplicación así como los frameworks y librerías que utilizan. Luego vienen sus dependencias y en algún momento los módulos JDK con *java.base* en la parte inferior - sigue leyendo para más detalles sobre esto.

Módulo base

Hay un módulo que los gobierna a todos: *java.base*, el llamado *módulo base*. Contiene clases como Class y ClassLoader, paquetes como java.lang y java.util, y los módulos
todo el sistema de módulos. Sin él, un programa en la JVM no puede funcionar y por eso recibe un estatus especial:

- el sistema de módulos lo conoce específicamente
- no hay necesidad de poner requiere java.base en una declaración de módulo - una dependencia en el módulo base viene de forma gratuita

Por eso, cuando en una sección anterior se hablaba de las dependencias de los distintos módulos, no estaba del todo completo. Todos ellos también dependen implícitamente del módulo base -
porque tienen que hacerlo. Y cuando en la sección anterior se decía que el sistema de módulos empieza con la resolución de módulos, tampoco es 100% correcto. Lo primero que

ocurre es que, en un profundo rompecabezas de gallina y huevo, el sistema de módulos resuelve el módulo base y se arranca a sí mismo.

Ventajas del sistema de módulos

Entonces, ¿qué obtienes por las molestias de crear declaraciones de módulos para tus proyectos? He aquí las tres ventajas más destacadas:

- fuerte encapsulación•
  configuración fiable•
  plataforma escalable

Encapsulación fuerte

Sin módulos, cada clase pública o miembro es libre de ser utilizado por cualquier otra clase - no hay manera de hacer algo visible dentro de un JAR, pero no más allá de sus límites. Y
incluso la visibilidad no pública no es realmente un elemento de disuasión porque siempre hay reflexión que se puede utilizar para romper en las API privadas. La razón es que los JAR no tienen límites,
son sólo contenedores para que el cargador de clases cargue las clases.

Los módulos son diferentes, tienen un límite que el compilador y el tiempo de ejecución reconocen. Un tipo de un módulo sólo se puede utilizar si:

- el tipo es público (como
  antes)• el paquete se exporta
- el módulo que utiliza el tipo lee el módulo que contiene el tipo

Esto significa que el creador de un módulo tiene mucho más control sobre los tipos que componen la API pública. Ya no son *todos los tipos públicos*, ahora son *todos los tipos públicos exportados.*
que finalmente nos permite bloquear la funcionalidad que contiene tipos públicos que se supone que no deben utilizarse fuera de un subproyecto.

Obviamente, esto es crucial para las propias API del JDK, cuyos desarrolladores ya no tienen que suplicarnos que no utilicemos paquetes como sun.* o com.sun.* (para más información sobre lo que
ocurrido con esas API internas y por qué puede seguir utilizando sun.misc.Unsafe, consulte esto artículo sobre encapsulación fuerte de las partes internas del JDK). Además, el JDK ya no necesita confiar en el enfoque manual del gestor de seguridad para impedir el acceso a las aplicaciones sensibles desde el punto de vista de la seguridad.
tipos y métodos, eliminando así toda una clase de riesgos potenciales para la seguridad.
Las bibliotecas y los marcos de trabajo también pueden beneficiarse de comunicar claramente y hacer cumplir qué API están destinadas a ser públicas y (presumiblemente) estables y cuáles son internas.

Las bases de código de aplicaciones pequeñas y grandes pueden estar seguras de no utilizar accidentalmente API internas de sus dependencias que puedan cambiar en cualquier versión de parche. Bases de código más grandes
puede beneficiarse aún más de la creación de varios módulos con límites bien

definidos. De esta forma, los desarrolladores que implementan una función pueden comunicar claramente a sus colegas
qué partes del código añadido están pensadas para su uso en otras partes de la aplicación y cuáles son sólo andamiaje interno - no más uso accidental de una API que "nunca fue
destinado a ese caso de uso".

Dicho todo esto, si es absolutamente necesario utilizar API internas, del JDK u otras aún puede hacerlo con estos dos indicadores de línea de comandos, siempre que controle el comando de inicio de la aplicación.

Configuración fiable

Durante la resolución del módulo, el sistema de módulos comprueba si todos los dependencias, directas y transitivas, están presentes e informa de un error si falta algo. Pero va más allá de la mera comprobación de la presencia.

No debe haber ambigüedad, es decir, dos artefactos no pueden afirmar que son el mismo módulo. Esto es especialmente interesante en el caso de que existan dos versiones del mismo módulo. Dado que el sistema de módulos no tiene concepto de versiones (más allá de registrar
como una cadena), lo trata como un módulo duplicado. En consecuencia, informa de un error si se encuentra con esta situación.

No debe haber ciclos de dependencia estática entre módulos. En tiempo de ejecución, es es posible e incluso necesario que los módulos accedan entre sí (piense en el código que utiliza anotaciones de Spring y en Spring reflejando sobre ese código), pero no deben ser compilar dependencias (Spring obviamente no se compila contra el código sobre el que se refleja).

Los paquetes deben tener un origen único, por lo que no puede haber dos módulos que contengan tipos en el mismo paquete. Si lo hacen, esto se llama un paquete *dividido*, y el sistema de módulos se negará a compilar o lanzar tales configuraciones.

Por supuesto, esta verificación no es hermética y es posible que los problemas se oculten durante mucho tiempo.
suficiente para bloquear una aplicación en ejecución. Si, por ejemplo, la versión incorrecta de un módulo termina en el lugar correcto, la aplicación se iniciará (todos los módulos necesarios están presentes), pero se bloqueará más tarde, cuando, por ejemplo, falte una clase o un método. Sin embargo, detecta con antelación una serie de problemas comunes, reduciendo la posibilidad de que una
la aplicación lanzada fallará en tiempo de ejecución debido a problemas de dependencia.

Plataforma escalable

Con el JDK dividido en módulos para todo, desde el manejo de XML hasta la API JDBC, es por fin es posible crear una imagen de *tiempo de ejecución* que contenga sólo las características del JDK que necesitas y enviarla con tu aplicación. Si tu código base está totalmente modularizado, puedes ir un paso más allá e incluir tus módulos en esa imagen, convirtiéndola en una *imagen de aplicación* autónoma que viene con todo lo que necesita, desde tu código hasta los módulos.
a las APIs del JDK y a la JVM. Este artículo explica cómo hacerlo.

Lecturas complementarias

Ahora tiene una comprensión básica del funcionamiento y las ventajas del módulo sistema y está listo para explorar muchos más temas para profundizar sus conocimientos. Los siguientes artículos no tienen que leerse en orden - cada uno menciona justo al
comenzando qué otras deberías haber leído antes.

Gestión de dependencias y API más sofisticada:

- Acceso reflexivo con módulos y paquetes abiertos

- Dependencias opcionales con requires static
- Legibilidad implícita con requisitos transitivos
- Exportaciones y aperturas cualificadas
- Desacoplamiento de módulos con servicios

Desde JAR hasta módulos e imágenes:

- Código en la ruta de clase: el módulo sin nombre
- Modularización incremental con módulos automáticos
- Creación de imágenes de tiempo de ejecución e imágenes de aplicación con jlink

Conocimiento más profundo del sistema de módulos:

- Creación de módulos en la línea de comandos
- Encapsulación fuerte (de los componentes internos del JDK)
- Eludir la encapsulación fuerte con --add-exports y --add-opens
- Ampliar el gráfico de módulos con --add-modules y --add-reads