Current Tutorial
Introduction to Modules in Java

➜

Next in the Series
Reflective Access with Open Modules and Open Packages

Introduction to Modules in Java

The Java APIs are organized into methods, classes, packages, and - at the highest level - modules. A module has a number of essential pieces of information attached to it:

- a name
- a list of dependencies on other modules
- a public API (with everything else being module internal and inaccessible)
- a list of services it uses and provides

Not only the Java APIs come with these information, you also have the option to create modules for your own projects. By deploying your project as modules, you increase reliability and maintainability, prevent accidental use of internal APIs, and can more easily create runtime images that contain just the JDK code you need (with the option to include your own app in the image to make it standalone).

Before we come to these benefits, we will explore how to define a module and its properties, how to turn it into a deliverable JAR, and how the module system handles them. To make that easier, we're going to assume that everything (code in the JDK, libraries, frameworks, apps) is a module - why that is not necessary and how modules can be created incrementally is covered in dedicated articles.

Module Declarations

At the core of each module is the *module declaration*, a file with the name `module-info.java` that defines all properties of a module. As an example, here's the one for `java.sql`, the platform module that defines the JDBC API:

```
module java.sql {
    requires transitive java.logging;
    requires transitive java.transaction.xa;
    requires transitive java.xml;

    exports java.sql;
    exports javax.sql;

    uses java.sql.Driver;
}
```

It defines the module's name (*java.sql*), its dependencies on other modules (`java.logging`, `java.transaction.xa`, `java.xml`), the packages that make up its public API (`java.sql` and `javax.sql`), and which services it uses (`java.sql.Driver`). This module already employs a more refined form to define dependencies by adding the keyword `transitive` but does of course not use all capabilities of the module system. Generally speaking, a module declaration has the following basic form:

```
module $NAME {
    // for each dependency:
    requires $MODULE;

    // for each API package:
    exports $PACKAGE

    // for each package intended for reflection:
    opens $PACKAGE;

    // for each used service:
    uses $TYPE;

    // for each provided service:
    provides $TYPE with $CLASS;
}
```

(The entire module can be `open` and the `requires`, `exports`, and `opens` directives can be further refined, but those are topics for later.)

You can create module declarations for your projects. Their recommended location - where all tools will pick them up the easiest - is in a project's source root folder, i.e. the folder containing your package directories, often `src/main/java`. For a library, a module declaration might look like this:

```
module com.example.lib {
    requires java.sql;
    requires com.sample.other;

    exports com.example.lib;
    exports com.example.lib.db;

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

For an app, it might be something like this:

```
module com.example.app {
    requires com.example.lib;

    opens com.example.app.entities;

    provides com.example.lib.Service
        with com.example.app.MyService;
}
```

Let's quickly go over the details. This section focuses on what needs to go into the module declaration:

- module name
- dependencies
- exported packages
- used and provided services

The effects are discussed in a later section.

Module Name

Module names have the same requirements and guidelines as package names:

- legal letters include `A`-`Z`, `a`-`z`, `0`-`9`, `_`, and `$`, separated by `.`
- by convention module names are all lower case and `$` is only used for mechanically generated code
- names should be globally unique

In the earlier example, the JDK module's declaration started with `module java.sql`, which defined the module with the name *java.sql*. The two custom modules were named `com.example.lib` and `com.example.app`.

Given a JAR, the corresponding module name can be inferred from the project's documentation, with a look into the `module-info.class` file in the JAR (more on that later), with the help of an IDE, or by running `jar --describe-module --file $FILE` against the JAR file.

Regarding module name uniqueness, the recommendation is the same as for packages: Pick a URL that's associated with the project and invert it to come up with the first part of the module name, then refine from there. (This implies that the two example modules are associated with the domain example.com.) If you apply this process to module names and package names, the former will usually be a prefix of the latter because modules are more general than packages. That is by no means required but an indicator that the names were chosen well.

Requiring Dependencies

The `requires` directives list all direct dependencies by their module names. Take a look at those from the three modules above:

```
// from java.sql, but without `transitive`
requires java.logging;
requires java.transaction.xa;
requires java.xml;

// from com.example.lib
requires java.sql;
requires com.sample.other;

// from com.example.app
requires com.example.lib;
```

We can see that the app module *com.example.app* depends on the library *com.example.lib*, which in turn needs the unrelated module *com.sample.other* and the platform module *java.sql*. We don't have the declaration of *com.sample.other* but we know that *java.sql* depends on *java.logging*, *java.transaction.xa*, and *java.xml*. If we look those up, we see that they have no further dependencies. (Or rather, no explicit dependencies - check the section on the base module below for more details.)

It is also possible to handle optional dependencies (with `requires static`) and to "forward" dependencies that are part of a module's API (with `requires transitive`), but that's covered in separate articles.

The list of external dependencies is in all likelihood very similar to the dependencies listed in your build configuration. This often leads to the question whether this is redundant and should be auto-generated. It's not redundant because a module name contains no version or really any other information that a build tool needs to obtain a JAR (like group ID and artifact ID) and build configurations list those information but not a module's name. Because the module name can be inferred given a JAR it is possible to generate this section of `module-info.java`. It's not clear whether that's worth the effort, though, particularly with the added complexity of dependencies on platform modules and of `static` and `transitive` modifiers as well as the fact that IDEs already propose adding `requires` directives when they're needed (much like package imports), which makes updating module declarations very simple.

Exporting And Opening Packages

By default all types, even `public` ones, are only accessible inside a module. To give code outside of a module access to a type, the package containing the type needs to be either *exported* or *opened*. This is achieved by using the `exports` and `opens` directives, which include the name of a package the module contains. The exact effect of exporting is discussed in the section on strong encapsulation below, of opening in an article on reflection, but the gist is:

- public types and members in exported packages are available at compile and run time
- all types and members in open packages can be accessed at run time via reflection

Here are the `exports` and `opens` directives from the three example modules:

```
// from module java.sql
exports java.sql;
exports javax.sql;

// from com.example.lib
exports com.example.lib;
exports com.example.lib.db;

// from com.example.app
opens com.example.app.entities;
```

This shows that *java.sql* exports a package of the same name as well as `javax.sql` - the module contains a lot more packages of course, but they are not part of its API and don't concern us. The library module exports two packages to be used by other modules - again, all other (potential) packages are safely locked away. The app module exports no packages, which isn't uncommon as the module launching the application is rarely a dependency of other modules and so nobody calls into it. It does open `com.example.app.entities` for reflection though - judging by the name probably because it contains entities that other modules want to interact with via reflection (think JPA).

There are also qualified variants of the `exports` and `opens` directives that allow you to export/open a package to specific modules only.

As a rule of thumb, try to export as few packages as possible - just like keeping fields private, only making methods package-visible or public if needed, and making classes package-visible by default and only public if needed in another package. This reduces the amount of code that is visible elsewhere, which reduces complexity.

Using And Providing Services

Services are their own topic - for now it suffices to say that you can use them to decouple the user of an API from its implementation, making it easier to replace it as late as when launching the application. If a module uses a type (an interface or a class) as a service, it needs to state that in the module declaration with the `uses` directive, which includes the fully qualified type name. Modules providing a service express in their module declaration which of their own types do that (usually by implementing or extending it).

The library and app example modules show the two sides:

```
// in com.example.lib
uses com.example.lib.Service;

// in module com.example.app
provides com.example.lib.Service
    with com.example.app.MyService;
```

The lib module uses `Service`, one of its own types, as a service and the app module, which depends on the lib module, provides it with `MyService`. At run time the lib module will then access all classes that implement / extend the service type by using the `ServiceLoader` API with a call as simple as `ServiceLoader.load(Service.class)`. That means the library module executes behavior defined in the app module even though it doesn't depend on it - that's great to untangle dependencies and keep modules focused in their concerns.

Building and Launching Modules

The module declaration `module-info.java` is a source file like any other and so it takes a few steps before it's running in the JVM. Fortunately, these are the exact same steps that your source code takes and most build tools and IDEs understand that well enough to adapt to its presence. In all likelihood, you don't need to do anything manually to build and launch a modular code base. Of course there is value in understanding the nitty, gritty details, so a dedicated article takes you from source code to running JVM with just command line tools.

Here, we'll stay on a higher level of abstraction and instead discuss a few concepts that play an important role in building and running modular code:

- modular JARs
- module path
- module resolution and module graph
- the base module

Modular JARs

A `module-info.java` file (aka module declaration) gets compiled to `module-info.class` (called *module descriptor*), which can then be placed into a JAR's root directory or in a version-specific directory, if it's a multi-release JAR. A JAR containing a module descriptor is called a *modular JAR* and is ready to be used as a module - JARs without a descriptor are *plain JARs*. If a module JAR is placed on the module path (see below), it becomes a module at run time, but it can still be used on the class path as well, where it becomes part of the unnamed module just like plain JARs on the class path.

Module Path

The *module path* is a new concept that parallels the class path: It is a list of artifacts (JARs or bytecode folders) and directories that contain artifacts. The module system uses it to locate required modules that are not found in the runtime, so usually all app, library, and framework modules. It turns all artifacts on the module path into modules, even plain JARs, which get turned into automatic modules, which is enables incremental modularization. Both `javac` and `java` as well as other module-related commands understand and process the module path.

Side note: This and the previous section together have revealed a possibly surprising behavior of the module system: Whether a JAR is modular or not does not determine whether it is treated as a module! All JARs on the class path are treated as a single barely-a-module, all JARs on the module path get turned into modules. That means the person in charge of a project gets to decide which dependencies end up as individual modules and which don't (as opposed to the dependencies' maintainers).

Module Resolution and Module Graph

To launch a modular app, run the `java` command with a module path and a so-called *initial module* - the module that contains the `main` method:

```
# modules are in `app-jars` | initial module is `com.example.app`
java --module-path app-jars --module com.example.app
```

This will start a process called *module resolution*: Beginning with the initial module's name, the module system will search the module path for it. If it finds it, it will check its `requires` directives to see what modules it needs and then repeats the process for them. If it doesn't find a module, it will throw an error then and there, letting you know a dependency is missing. You can observe this process by adding the command line option `--show-module-resolution`.

The result of this process is the *module graph*. Its nodes are modules, its edges are a bit more complicated: Each `requires` directive spawns an edge between the two modules, called a *readability edge* where the requiring module *reads* the required one. There are other ways to create edges but that doesn't need to concern us now because it doesn't change anything fundamental.

If we imagine an average Java program, for example a backend for a web app, we can picture it's module graph: At the top we'll find the initial module, further down the other app modules as well as the frameworks and libraries they use. Then come their dependencies and at some point the JDK modules with *java.base* at the bottom - read on for details on that.

Base Module

There's one module to rule them all: *java.base*, the so-called *base module*. It contains classes like `Class` and `ClassLoader`, packages like `java.lang` and `java.util`, and the entire module system. Without it, a program on the JVM can't function and so it gets special status:

- the module system is aware of it specifically
- no need to put `requires java.base` in a module declaration - a dependency on the base module comes for free

So when an earlier section discussed the dependencies of the various modules, that wasn't entirely complete. They all also implicitly depend on the base module - because they have to. And when the previous section said that the module system starts with module resolution, that's not 100% correct either. The first thing that happens is that, in a profound chicken-and-egg head-scratcher, the module system resolves the base module and bootstraps itself.

Module System Benefits

So what do you get for your troubles of creating module declarations for your projects? Here are the three most prominent benefits:

- strong encapsulation
- reliable configuration
- scalable platform

Strong Encapsulation

Without modules, every public class or member is free to be used by any other class - no way to make something visible within a JAR but not beyond it's boundaries. And even non-public visibility isn't really a deterrent because there's always reflection that can be used to break into private APIs. The reason is that JARs have no boundaries, they are just containers for the class loader to load classes from.

Modules are different, they *do* have a boundary that compiler and runtime recognize. A type from a module can only be used if:

- the type is public (as before)
- the package is exported
- the module using the type reads the module containing the type

That means the creator of a module has much more control over which types make up the public API. No longer is it *all public types*, now it's *all public types in exported packages*, which finally allows us to lock away functionality that contains public types that are not supposed to be used outside a sub-project.

This is obviously crucial for the JDK APIs itself, whose developers no longer need to plead with us not to use packages like `sun.*` or `com.sun.*` (for more on what happened to those internal APIs and why you can still use `sun.misc.Unsafe`, see this article on strong encapsulation of JDK internals). The JDK also no longer needs to rely on the security manager's manual approach to prevent access to security-sensitive types and methods, thus eliminating an entire class of potential security hazards. Libraries and frameworks can also benefit from clearly communicating and enforcing which APIs are meant to be public and (presumably) stable and which are internal.

Application code bases small and large can be sure not to accidentally use internal APIs of their dependencies that my change in any patch release. Larger code bases can further benefit from creating multiple modules with strong boundaries. That way, developers that implement a feature can clearly communicate to their colleagues which parts of the added code are meant for use in other parts of the app and which are just internal scaffolding - no more accidental use of an API that "was never intended for that use case".

All of that said, if you absolutely have to use internal APIs, of the JDK or other modules, you still can with these two command line flags, assuming you're in control of the application's launch command.

Reliable Configuration

During module resolution, the module system checks whether all required dependencies, direct and transitive, are present and reports an error if something's missing. But it goes beyond just checking presence.

There must be no ambiguity, i.e. no two artifacts can claim they're the same module. This is particularly interesting in the case where two versions of the same module are present. Because the module system has no concept of versions (beyond recording them as a string), it treats this as a duplicate module. Accordingly, it reports an error if it runs into this situation.

There must be no static dependency cycles between modules. At run time, it's possible and even necessary for modules to access each other (think about code using Spring annotations and Spring reflecting over that code), but these must not be compile dependencies (Spring is obviously not compiled against the code it reflects over).

Packages should have a unique origin, so no two modules can contain types in the same package. If they do, this is called a *split package*, and the module system will refuse to compile or launch such configurations.

This verification isn't airtight of course and it's possible for problems to hide long enough to crash a running application. If, for example, the wrong version of a module ends up in the right place, the application will launch (all required modules are present) but will crash later, when, for example, a class or method is missing. It does detect a number of common problems early, though, reducing the chance that an application that launched will fail at run time because of dependency issues.

Scalable Platform

With the JDK split into modules for everything from XML handling to JDBC API, it is finally possible to hand-craft a *runtime image* that contains just the JDK features you need and ship that with your app. If your code base is fully modularized, you can go one step further and include your modules in that image, making it a self-contained *application image* that comes with everything it needs, from your code to dependencies to JDK APIs and the JVM. This article explains how to do that.

Further Reading

You now have a basic understanding of the workings and benefits of the module system and are ready to explore many more topics to deepen your knowledge. The following articles don't have to be read in order - each mentions right at the beginning which others you should've read before.

More sophisticated dependency and API Management:

- Reflective Access with Open Modules and Open Packages
- Optional Dependencies with `requires static`
- Implied Readability with `requires transitive`
- Qualified `exports` and `opens`
- Decoupling Modules with Services

From JARs to modules to images:

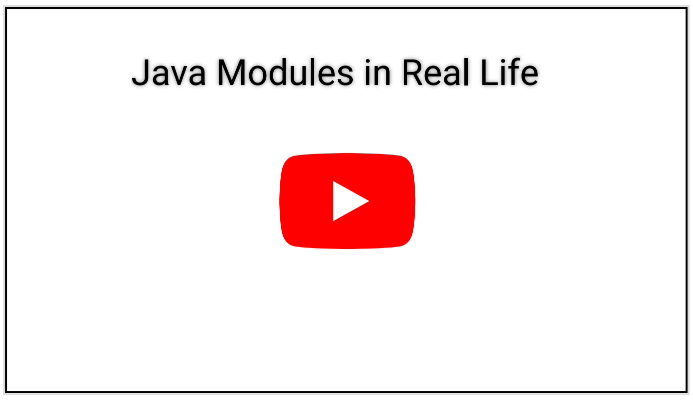
- Code on the Class Path - the Unnamed Module
- Incremental Modularization with Automatic Modules
- Creating Runtime Images and Application Images with jlink

Deeper module system knowledge:

- Building Modules on the Command Line
- Strong Encapsulation (of JDK Internals)
- Circumventing Strong Encapsulation with `--add-exports` and `--add-opens`

- Extending the Module Graph with `--add-modules` and `--add-reads`

More Learning



Last update: *September 14, 2021*

Current Tutorial		Next in the Series
Introduction to Modules in Java	➜	Reflective Access with Open Modules and Open Packages

Home > Tutorials > Modules > Introduction to Modules in Java

About

- About Java
- About OpenJDK
- Getting Started
- Oracle Java SE Subscription

Downloads

- All Releases
- Source Code

Learning Java

- Documentation
- Java 21 API Docs
- Tutorials
- FAQ
- Java YouTube

Community

- Java User Groups
- Java Conferences
- Contributing

Stay Informed

- Inside.java
- Newscast
- Podcast
- Java Magazine
- Java YouTube
- @java on Twitter