Und

What th

By Paul

In this article, I introduce the Java 9 Platform Module System (JPMS), the most important new software engineering technology in Java since its inception. Modularity—the result of Project Jigsaw—helps developers at all levels be more productive as they build, maintain, and evolve software systems, especially large systems.

Paul Deitel

What Is a Module?

Modularity adds a higher level of aggregation above packages. The key new language element is the *module*—a uniquely named, reusable group of related packages, as well as resources (such as images and XML files) and a *module descriptor* specifying

- the module's *name*
- the module's *dependencies* (that is, other modules this module depends on)
- the packages it explicitly makes available to other modules (all other packages in the module are *implicitly unavailable* to other modules)
- the *services it offers*
- the *services it consumes*
- to what other modules it allows *reflection*

History

The Java SE platform has been around since 1995. There are now approximately 10 million developers using it to build everything from small apps for resource-constrained devices—like those in the Internet of Things (IoT) and other embedded devices—to large-scale business-critical and mission-critical systems. There are massive amounts of legacy code out there, but until now, the Java platform has primarily been a monolithic one-size-fits-all solution. Over the years, there have been various efforts geared to modularizing Java, but none is widely used—and none could be used to modularize the Java platform.

Modularizing the Java SE platform has been challenging to implement, and the effort has taken many years. JSR 277: Java Module System was originally proposed in 2005 for Java 7. This JSR was later superseded by JSR 376: Java Platform Module System and targeted for Java 8. The Java SE platform is now modularized in Java 9, but only after Java 9 was delayed until September 2017.

Goals

According to JSR 376, the key goals of modularizing the Java SE platform are

> "Each module must explicitly state its dependencies."

- Reliable configuration—Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner that's recognized both at compile time and execution time. The system can walk through these dependencies to determine the subset of all modules required to support your app.
- Strong encapsulation—The packages in a module are accessible to other modules only if the module explicitly exports them. Even then, another module cannot use those packages unless it explicitly states that it requires the other module's capabilities. This improves platform security because fewer classes are accessible to potential attackers. You may find that considering modularity helps you come up with cleaner, more logical designs.
- Scalable Java platform—Previously, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain and evolve. It couldn't be easily subsetted. The platform is now modularized into 95 modules (this number might change as Java evolves). You can create custom runtimes consisting of only modules you need for your apps or the devices you're targeting. For example, if a device does not support GUIs, you could create a runtime that does not include the GUI modules, significantly reducing the runtime's size.
- Greater platform integrity—Before Java 9, it was possible to use many classes in the platform that were not meant for use by an app's classes. With strong encapsulation, these internal APIs are truly encapsulated and hidden from apps using the platform. This can make migrating legacy code to modularized Java 9 problematic if your code depends on internal APIs.
- Improved performance—The JVM uses various optimization techniques to improve application performance. JSR 376 indicates that these techniques are more effective when it's known in advance that required types are located only in specific modules.

Listing the JDK's Modules

A crucial aspect of Java 9 is dividing the JDK into modules to support various configurations. (Consult "JEP 200: The Modular JDK." All the Java modularity JEPs and JSRs are shown in **Table 1.**) Using the java command from the JDK's bin folder with the `--list-modules` option, as in:

```
java --list-modules
```

lists the JDK's set of modules, which includes the standard modules that implement the Java Language SE Specification (names starting with `java`), JavaFX modules (names starting with `javafx`), JDK-specific modules (names starting with `jdk`) and Oracle-specific modules (names starting with `oracle`). Each module name is followed by a version string—`@` indicates that the module belongs to Java 9.

JEP 200: **THE MODULAR JDK**
JEP 201: **MODULAR SOURCE CODE**
JEP 220: **MODULAR RUN-TIME IMAGES**
JEP 260: **ENCAPSULATE MOST INTERNAL APIS**
JEP 261: **MODULE SYSTEM**
JEP 275: **MODULAR JAVA APPLICATION PACKAGING**
JEP 282: **JLINK: THE JAVA LINKER**
JSR 376: **JAVA PLATFORM MODULE SYSTEM**
JSR 379: **JAVA SE 9**

Table 1. Java Modularity JEPs and JSRs

Module Declarations

As we mentioned, a module must provide a module descriptor—metadata that specifies the module's dependencies, the packages the module makes available to other modules, and more. A module descriptor is the compiled version of a module declaration that's defined in a file named `module-info.java`. Each module declaration begins with the keyword `module`, followed by a unique module name and a module body enclosed in braces, as in:

```
module modulename {
}
```

> "A key motivation of the module system is strong encapsulation."

The module declaration's body can be empty or may contain various *module directives*, including `requires`, `exports`, `provides…with`, `uses` and `opens` (each of which we discuss). As you'll see later, compiling the module declaration creates the module descriptor, which is stored in a file named `module-info.class` in the module's root folder. Here we briefly introduce each module directive. After that, we'll present actual module declarations.

The keywords `exports`, `module`, `open`, `opens`, `provides`, `requires`, `uses`, `with`, as well as `to` and `transitive`, which we introduce later, are restricted keywords. They're keywords only in module declarations and may be used as identifiers anywhere else in your code.

requires. A `requires` module directive specifies that this module depends on another module—this relationship is called a *module dependency*. Each module must explicitly state its dependencies. When module A `requires` module B, module A is said to *read* module B and module B is *read by* module A. To specify a dependency on another module, use `requires`, as in:

```
requires modulename;
```

There is also a `requires static` directive to indicate that a module is required at compile time, but is optional at runtime. This is known as an *optional dependency* and won't be discussed in this introduction.

requires transitive—implied readability. To specify a dependency on another module and to ensure that other modules reading your module also read that dependency—known as *implied readability*—use `requires transitive`, as in:

```
requires transitive modulename;
```

Consider the following directive from the `java.desktop` module declaration:

```
requires transitive java.xml;
```

In this case, any module that reads `java.desktop` also implicitly reads `java.xml`. For example, if a method from the `java.desktop` module returns a type from the `java.xml` module, code in modules that read `java.desktop` becomes dependent on `java.xml`. Without the `requires transitive` directive in `java.desktop`'s module declaration, such dependent modules will not compile unless they *explicitly* read `java.xml`.

According to JSR 379 Java SE's standard modules must grant implied readability in all cases like the one described here. Also, though a Java SE standard module may depend on non-standard modules, it *must not* grant implied readability to them. This ensures that code depending only on Java SE standard modules is portable across Java SE implementations.

exports and exports…to. An `exports` module directive specifies one of the module's packages whose `public` types (and their nested `public` and `protected` types) should be accessible to code in all other modules. An `exports…to` directive enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package—this is known as a *qualified* export.

uses. A `uses` module directive specifies a service used by this module—making the module a service consumer. A *service* is an object of a class that implements the interface or extends the `abstract` class specified in the `uses` directive.

provides…with. A `provides…with` module directive specifies that a module provides a service implementation—making the module a *service provider*. The `provides` part of the directive specifies an interface or `abstract` class listed in a module's `uses` directive and the `with` part of the directive specifies the name of the service provider class that `implements` the interface or `extends` the `abstract` class.

open, opens, and opens…to. Before Java 9, reflection could be used to learn about all types in a package and all members of a type—even its `private` members—whether you wanted to allow this capability or not. Thus, nothing was truly encapsulated.

A key motivation of the module system is strong encapsulation. By default, a type in a module is not accessible to other modules unless it's a public type *and* you export its package. You expose only the packages you want to expose. With Java 9, this also applies to reflection.

Allowing runtime-only access to a package. An opens module directive of the form

```
opens package
```

indicates that a specific *package*'s `public` types (and their nested `public` and `protected` types) are accessible to code in other modules at runtime only. Also, all the types in the specified package (and all of the types' members) are accessible via reflection.

Allowing runtime-only access to a package by specific modules. An `opens…to` module directive of the form

```
opens package to comma-separated-list-of-modules
```

indicates that a specific *package*'s `public` types (and their nested `public` and `protected` types) are accessible to code in the listed modules at runtime only. All of the types in the specified package (and all of the types' members) are accessible via reflection to code in the specified modules.

Allowing runtime-only access to all packages in a module. If all the packages in a given module should be accessible at runtime and via reflection to all other modules, you may `open` the entire module, as in:

```
open module modulename {
   // module directives
}
```

Reflection Defaults

By default, a module with runtime reflective access to a package can see the package's `public` types (and their nested `public` and `protected` types). However, the code in other modules can access *all* types in the exposed package and *all* members within those types, including `private` members via `setAccessible`, as in earlier Java versions.

For more information on `setAccessible` and reflection, see Oracle's documentation.

In the rest of this article, learn more about reflection defaults and create a simple Wecome app that demonstrates module fundamentals.

Paul Deitel, CEO and chief technical officer of Deitel & Associates, is a graduate of MIT with 35 years of experience in computing. He is a Java Champion and has been programming in Java for more than 22 years. He and his coauthor, Dr. Harvey M. Deitel, are the world's best-selling programming-language authors. Paul has delivered Java, Android, iOS, C#, C++, C, and Internet programming courses to industry, government, and academic clients internationally.

NOTE: This article is excerpted from *Java Magazine* September/October 2017. Continue to the full article, which was adapted from the recently published book *Java 9 for Programmers*, by Paul and Harvey Deitel.

Learn More

Project Jigsaw: Module System Quick-Start Guide | Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications | Java Magazine