ToiletPaper #91

Java goes module

Author: Arnim Kreutzer / Business Division Media

✖ Problem

Since its release in March 1995, Java has become one of the most popular and widely used programming languages. During its development, the requirements for the behavior of codebases with seven-digit LOC numbers weren't considered. As a result, the implemented mechanisms for loading classes – keyword Jar Hell – and the access control for such systems are inadequate. Between ***package*** and ***public*** it lacks an Access Modifier that restricts access to classes from certain packages.
Partly for this reason, Sun launched the project ***jigsaw*** in 2008 to simplify the development and maintenance of large applications and libraries. For this purpose, a modular system was developed and integrated into the Java platform.

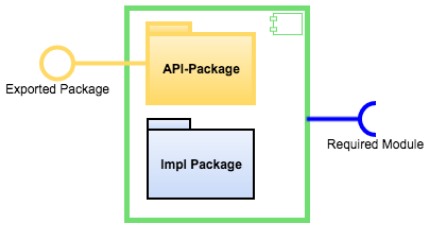
✔ Solution

Using modules, systems are disassembled into closed, loosely coupled software components according to their technical and / or functional aspects. At the Java code level, module packages combine to form a closed or encapsulated group. If using a module, the API must be specified.

➔ Example

The definition of a module is made by the file ***module-info.java***. This must be in the root directory of the source tree and looks like this in a simple case:

```
module my.module.name {
    exports my.exported.api.pckg;
    requires name.of.required.module;
}
```



A module can export several packages and (of course) depend on several modules. For each exported package or needed module, a line with an export or requires directive is entered. The API of a module consists of all public elements of the exported packages.
Already at compile time, the module system checks the dependencies and creates the module graph, starting from a root module, in which nodes represent the modules and edges the dependencies. Cyclic dependencies between modules are not allowed and result in an error when detected. The module graph is therefore guaranteed to be acyclic.
By embedding the above properties in the JVM, you get the following benefits of the Java Platform Module System over the previous JVM versions:

1. Reliable configuration: when starting the JVM, the dependencies are checked. So, it is already certain at this time that all necessary modules are available.

2. Severe encapsulation: encapsulation prevents internal classes from being accessed from outside the module (in case need to be public for cross-package / in-module use).

3. Scalable development: modules can be developed in independent teams. The modules are delimited automatically by the module system via the exchange of the explicitly exported parts.

4. Security: the encapsulation is realized on the lowest level of the JVM. Access via Reflection to sensitive internal classes is no longer possible.

5. Optimizations: the module graph defines which modules (incl. the platform modules) are required for the execution of the application. When starting the JVM, only these modules will be considered. Further, a minimal configuration can be created for the distribution. The always necessary linear search of the Classpath when loading classes is omitted, since the combination module / package system-wide is unique.

✚ Further Aspects

- Implementation of services
- Testing modules
- Migration & automated modules
- Reflection vs. strong encapsulation
- Module descriptor advanced
- Modulelayer/ configuration
- API design
- Runtime images with jlink (JEP-220, JEP-282)
- jamb.it/coffeetalks-2018-3