JEP 200: The Modular JDK

Owner	Mark Reinhold
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	9
JSR	376
Discussion	jigsaw dash dev at openjdk dot java dot net
Effort	XL
Duration	XL
Depends	JEP 201: Modular Source Code JEP 220: Modular Run-Time Images JEP 261: Module System
Reviewed by	Alan Bateman, Alex Buckley, Paul Sandoz
Endorsed by	Brian Goetz
Created	2014/07/22 14:08
Updated	2017/09/21 14:30
Issue	8051618

Summary

Use the Java Platform Module System, specified by JSR 376 and implemented by JEP 261, to modularize the JDK.

Goals

Divide the JDK into a set of modules that can be combined at compile time, build time, and run time into a variety of configurations including, but not limited to:

- Configurations corresponding to the full Java SE Platform, the full JRE, and the full JDK;
- Configurations roughly equivalent in content to each of the Compact Profiles defined in Java SE 8; and
- Custom configurations which contain only a specified set of modules possibly augmented by external library and application modules, and the modules transitively required by all of these modules.

The definition of the modular structure should make a clear distinction between standard modules, whose specifications are governed by the Java Community Process, and modules that are specific to the JDK. It should also distinguish modules that are included in the Java SE Platform Specification, and thereby made mandatory in every Platform Implementation, from all other modules.

Motivation

Project Jigsaw aims to design and implement a standard module system for the Java SE Platform and to apply that system to the Platform itself, and to the JDK. Its primary goals are to make implementations of the Platform more easily scalable down to small devices, improve security and maintainability, enable improved application performance, and provide developers with better tools for programming in the large.

Description

Design principles

The modular structure of the JDK implements the following principles:

1. Standard modules, whose specifications are governed by the JCP, have names starting with the string `"java."`.

Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
SCTP
Shenandoah
Skara
Sumatra
Tsan
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

ORACLE

2. All other modules are merely part of the JDK, and have names starting with the string `"jdk."`.
3. If a module exports a package that contains a type that contains a public or protected member that, in turn, refers to a type from some other module, then the first module must grant implied readability to the second, via `requires transitive`. (This ensures that method-invocation chaining works in the obvious way.)
4. A standard module may contain both standard and non-standard API packages. If a standard module exports a standard API package then the export may be qualified; if a standard module exports a non-standard API package then the export must be qualified. In either case, if a standard module exports a package with qualification then the export must be to some subset of the modules in the JDK. If a standard module is a Java SE module, *i.e.*, is included in the Java SE Platform Specification, then it must not export any non-SE API packages, at least not without qualification.
5. A standard module may depend upon one or more non-standard modules. It must not grant implied readability to any non-standard module. If it is a Java SE module then it must not grant implied readability to any non-SE module.
6. A non-standard module must not export any standard API packages. A non-standard module may grant implied readability to a standard module.

An important consequence of principles 4 and 5 is that code that depends only upon Java SE modules will depend only upon standard Java SE types, and thus be portable to all Implementations of the Java SE Platform.

The module graph

The modular structure of the JDK can be visualized as a graph: Each module is a node, and there is a directed edge from one module to another if the first depends upon the second. The full module graph has too many edges to be displayed easily; here is the *transitive reduction* of the graph, in which redundant edges are omitted (click to enlarge):

Herewith a guided tour of the module graph:

- Standard Java SE modules are colored orange; non-SE modules are colored blue.
- If one module depends upon another, and it grants implied readability to that module, then the edge from the first module to the second is solid; otherwise, the edge is dashed.
- At the very bottom is the `java.base` module, which contains essential classes such as `java.lang.Object` and `java.lang.String`. The base module depends upon no module, and every other module depends upon the base module. Edges to the base module are lighter than other edges.
- Near the top is the `java.se.ee` module, which gathers together all of the modules that comprise the Java SE Platform, including modules that overlap with the Java EE Platform Specification. This is an example of an *aggregator* module, which collects and re-exports the content of other modules but adds no content of its own. A run-time system configured to contain the `java.se.ee` module will contain all of the API packages of the Java SE Platform. A module is included in the Java SE Platform Specification

if, and only if, it is a standard module reachable from the `java.se.ee` module.

- The `java.se` aggregator module gathers together those parts of the Java SE Platform that do not overlap with Java EE.
- The non-standard modules include debugging and serviceability tools and APIs (*e.g.*, `jdk.jdi`, `jdk.jcmd` and `jdk.jconsole`), development tools (*e.g.*, `jdk.compiler`, `jdk.javadoc`, and `jdk.xml.bind`), and various service providers (*e.g.*, `jdk.charsets`, `jdk.scripting.nashorn`, and `jdk.crypto.ec`), which are made available to other modules via the existing `java.util.ServiceLoader` mechanism.
- The `java.smartcardio` module is standard but not part of the Java SE Platform Specification, hence its name starts with the string `"java."` but it is colored blue, and it is not reachable from the `java.se` module.

The module graph is, in effect, a new kind of API, and is specified and evolved as such. The subgraph of the module graph rooted at the `java.se.ee` module, with all non-SE modules and corresponding edges removed, is specified in the Java SE Platform Specification; its evolution will hereinafter be governed by the JCP. The evolution of the remainder of the graph will be covered by future JEPs. In either case, if a module is specified as being available for general use then it will be subject to the same evolutionary constraints as other APIs. Removing such a module or changing it incompatibly, in particular, will require public notice at least one major release in advance.

A tabular summary of all of the modules, including footprint metrics for a Linux/AMD64 build, is available here.

Testing

The unit and regression tests in the JDK and `jtreg`, the harness used to run them, now allow tests to be selected on the basis of the modules which they test and upon which they depend, so that arbitrary configurations of JDK modules can be tested.

The primary functional test of this enhancement inspects a configured set of modules to ensure that it is a valid combination of the modules defined herein, that each module has the expected content and exports the expected API packages, and that the modules have the expected dependence relationships.

The JCK can now test those aspects of the module graph which become part of the Java SE Platform Specification. This includes the names of the SE modules, their exported API packages, and the dependences amongst them that cause SE API packages to be re-exported. The JCK can also test arbitrary configurations of the SE modules present in a Platform Implementation.

Risks and Assumptions

The modular structure defined here does not support at least one known use case, namely that of wanting to use the `java.beans` package without having to require the very large `java.desktop` module. It might not address other use cases, as yet unknown. If a critical use case is not supported in the final implementation of this JEP then we expect to be able address it in a later release by refactoring the module graph.

Dependences

This JEP is the one of several for Project Jigsaw. The other JEPs are:

- 201: Modular Source Code
- 220: Modular Run-Time Images
- 260: Encapsulate Most Internal APIs
- 261: Module System; see also JSR 376
- 282: jlink: The Java Linker