JEP 261: Module System

Authors	Alan Bateman, Alex Buckley, Jonathan Gibbons, Mark Reinhold
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	L
Blocks	JEP 200: The Modular JDK JEP 282: jlink: The Java Linker
Depends	JEP 220: Modular Run-Time Images JEP 260: Encapsulate Most Internal APIs
Relates to	JEP 396: Strongly Encapsulate JDK Internals by Default JEP 403: Strongly Encapsulate JDK Internals
Reviewed by	Alan Bateman, Alex Buckley, Chris Hegarty, Jonathan Gibbons, Mandy Chung, Paul Sandoz
Endorsed by	Brian Goetz
Created	2014/10/23 15:05
Updated	2024/08/19 18:30
Issue	8061972

Summary

Implement the Java Platform Module System, as specified by JSR 376, together with related JDK-specific changes and enhancements.

Description

The Java Platform Module System (JSR 376) specifies changes and extensions to the Java programming language, the Java virtual machine, and the standard Java APIs. This JEP implements that specification. As a consequence, the `javac` compiler, the HotSpot virtual machine, and the run-time libraries implement modules as a fundamental new kind of Java program component and provide for the reliable configuration and strong encapsulation of modules in all phases of development.

This JEP also changes, extends, and adds JDK-specific tools and APIs, which are outside the scope of the JSR, that are related to compilation, linking, and execution. Related changes to other tools and APIs, *e.g.*, the `javadoc` tool and the Doclet API, are the subject of separate JEPs.

This JEP assumes that the reader is familiar with the latest *State of the Module System* document and also the other Project Jigsaw JEPs:

- 200: The Modular JDK
- 201: Modular Source Code
- 220: Modular Run-Time Images
- 260: Encapsulate Most Internal APIs
- 282: jlink: The Java Linker

Phases

To the familiar phases of compile time (the `javac` command) and run time (the `java` run-time launcher) we add the notion of *link time*, an optional phase between the two in which a set of modules can be assembled and optimized into a custom run-time image. The linking tool, `jlink`, is the subject of JEP 282; many of the new command-line options implemented by `javac` and `java` are also implemented by `jlink`.

Module paths

The `javac`, `jlink`, and `java` commands, as well as several others, now accept options to specify various *module paths*. A module path is a sequence, each element of which is either a *module definition* or a directory containing module definitions. Each module definition is either

- A *module artifact*, *i.e.*, a modular JAR file or a JMOD file containing a compiled module definition, or else
- An *exploded-module directory* whose name is, by convention, the module's name and whose content is an "exploded" directory tree corresponding to a package hierarchy.

In the latter case the directory tree can be a compiled module definition, populated with individual class and resource files and a `module-info.class` file at the root or, at compile time, a source module definition, populated with individual source files and a `module-info.java` file at the root.

A module path, like other kinds of paths, is specified by a string of path names separated by the host platform's path-separator character (`':'` on most platforms, `';'` on Windows).

Module paths are very different from class paths: Class paths are a means to locate definitions of individual types and resources, whereas module paths are a means to locate definitions of whole modules. Each element of a class path is a container of type and resource definitions, *i.e.*, either a JAR file or an exploded, package-hierarchical directory tree. Each element of a module path, by contrast, is a module definition or a directory which each element in the directory is a module definition, *i.e.*, a container of type and resource definitions, *i.e.*, either a modular JAR file, a JMOD file, or an exploded module directory.

During the resolution process the module system locates a module by searching along several different paths, dependent upon the phase, and also by searching

the compiled modules built-in to the environment, in the following order:

- The *compilation module path* (specified by the command-line option `--module-source-path`) contains module definitions in source form (compile time only).
- The *upgrade module path* (`--upgrade-module-path`) contains compiled definitions of modules intended to be used in preference to the compiled definitions of any upgradeable modules present amongst the system modules or on the application module path (compile time and run time).
- The *system modules* are the compiled modules built-in to the environment (compile time and run time). These typically include Java SE and JDK modules but, in the case of a custom linked image, can also include library and application modules. At compile time the system modules can be overridden via the `--system` option, which specifies a JDK image from which to load system modules.
- The *application module path* (`--module-path`, or `-p` for short) contains compiled definitions of library and application modules (all phases). At link time this path can also contain Java SE and JDK modules.

The module definitions present on these paths, together with the system modules, define the universe of *observable modules*.

When searching a module path for a module of a particular name, the module system takes the first definition of a module of that name. Version strings, if present, are ignored; if an element of a module path contains definitions of multiple modules with the same name then resolution fails and the compiler, linker, or virtual machine will report an error and exit. It is the responsibility of build tools and container applications to configure module paths so as to avoid version conflicts; it is not a goal of the module system to address the version-selection problem.

Root modules

The module system constructs a module graph by resolving the transitive closure of the dependences of a set of *root modules* with respect to the set of observable modules.

When the compiler compiles code in the unnamed module, or the `java` launcher is invoked and the main class of the application is loaded from the class path into the unnamed module of the application class loader, then the *default set of root modules for the unnamed module* is computed as follows in JDK 9:

- The `java.se` module is a root, if it exists. If it does not exist then every `java.*` module on the upgrade module path or among the system modules that exports at least one package, without qualification, is a root.
- Every non-`java.*` module on the upgrade module path or among the system modules that exports at least one package, without qualification, is also a root.

Update, June 2018: In JDK 11, the default set of root modules for the unnamed module changed. The default set is now computed as follows:

- *Every module on the upgrade module path or among the system modules that exports at least one package, without qualification, is a root.*

The `java.se` module still exists in JDK 11 and later, but it is no longer a root.

Otherwise, the default set of root modules depends upon the phase:

- At compile time it is usually the set of modules being compiled (more on this below);
- At link time it is empty; and
- At run time it is the application's main module, as specified via the `--module` (or `-m` for short) launcher option.

It is occasionally necessary to add modules to the default root set in order to ensure that specific platform, library, or service-provider modules will be present in the resulting module graph. In any phase the option

```
--add-modules <module>(,<module>)*
```

where `<module>` is a module name, adds the indicated modules to the default set of root modules. This option may be used more than once.

As a special case at run time, if `<module>` is `ALL-DEFAULT` then the default set of root modules for the unnamed module, as defined above, is added to the root set. This is useful when the application is a container that hosts other applications which can, in turn, depend upon modules not required by the container itself.

As a further special case at run time, if `<module>` is `ALL-SYSTEM` then all system modules are added to the root set, whether or not they are in the default set. This is sometimes needed by test harnesses. This option will cause many modules to be resolved; in general, `ALL-DEFAULT` should be preferred.

As a final special case, at both run time and link time, if `<module>` is `ALL-MODULE-PATH` then all observable modules found on the relevant module paths are added to the root set. `ALL-MODULE-PATH` is valid at both compile time and run time. This is provided for use by build tools such as Maven, which already ensure that all modules on the module path are needed. It is also a convenient means to add automatic modules to the root set.

Limiting the observable modules

It is sometimes useful to limit the observable modules for, *e.g.*, debugging, or to reduce the number of modules resolved when the main module is the unnamed module defined by the application class loader for the class path. The `--limit-modules` option can be used, in any phase, to do this. Its syntax is:

```
--limit-modules <module>(,<module>)*
```

where <module> is a module name. The effect of this option is to limit the observable modules to those in the transitive closure of the named modules plus the main module, if any, plus any further modules specified via the --add-modules option.

(The transitive closure computed for the interpretation of the --limit-modules option is a temporary result, used only to compute the limited set of observable modules. The resolver will be invoked again in order to compute the actual module graph.)

Increasing readability

When testing and debugging it is sometimes necessary to arrange for one module to read some other module, even though the first module does not depend upon the second via a requires clause in its module declaration. This may be needed to, *e.g.*, enable a module under test to access the test harness itself, or to access libraries related to the harness. The --add-reads option can be used, at both compile time and run time, to do this. Its syntax is:

```
--add-reads <source-module>=<target-module>
```

where <source-module> and <target-module> are module names.

The --add-reads option can be used more than once. The effect of each instance is to add a readability edge from the source module to the target module. This is, essentially, a command-line form of a requires clause in a module declaration, or an invocation of an unrestricted form of the Module::addReads method. As a consequence, code in the source module will be able to access types in a package of the target module at both compile time and run time if that package is exported via an exports clause in the source module's declaration, an invocation of the Module::addExports method, or an instance of the --add-exports option (defined below). Such code will, additionally, be able to access types in a package of the target module at run time if that module is declared to be open or if that package is opened via an opens clause in the source module's declaration, an invocation of the Module::addOpens method, or an instance of the --add-opens option (also defined below).

If, for example, a test harness injects a white-box test class into the java.management module, and that class extends an exported utility class in the (hypothetical) testng module, then the access it requires can be granted via the option

```
--add-reads java.management=testng
```

As a special case, if the <target-module> is ALL-UNNAMED then readability edges will be added from the source module to all present and future unnamed modules, including that corresponding to the class path. This allows code in modules to be tested by test frameworks that have not, themselves, yet been converted to modular form.

Breaking encapsulation

It is sometimes necessary to violate the access-control boundaries defined by the module system, and enforced by the compiler and virtual machine, in order to allow one module to access some of the unexported types of another module. This may be desirable in order to, *e.g.*, enable white-box testing of internal types, or to expose unsupported internal APIs to code that has come to depend upon them. The --add-exports option can be used, at both compile time and run time, to do this. Its syntax is:

```
--add-exports <source-module>/<package>=<target-module>(,<target-module>)*
```

where <source-module> and <target-module> are module names and <package> is the name of a package.

The --add-exports option can be used more than once, but at most once for any particular combination of source module and package name. The effect of each instance is to add a qualified export of the named package from the source module to the target module. This is, essentially, a command-line form of an exports clause in a module declaration, or an invocation of an unrestricted form of the Module::addExports method. As a consequence, code in the target module will be able to access public types in the named package of the source module if the target module reads the source module, either via a requires clause in its module declaration, an invocation of the Module::addReads method, or an instance of the --add-reads option.

If, for example, the module jmx.wbtest contains a white-box test for the unexported com.sun.jmx.remote.internal package of the java.management module, then the access it requires can be granted via the option

```
--add-exports java.management/com.sun.jmx.remote.internal=jmx.wbtest
```

As a special case, if the <target-module> is ALL-UNNAMED then the source package will be exported to all unnamed modules, whether they exist initially or are created later on. Thus access to the sun.management package of the java.management module can be granted to all code on the class path via the option

```
--add-exports java.management/sun.management=ALL-UNNAMED
```

The --add-exports option enables access to the public types of a specified package. It is sometimes necessary to go further and enable access to all non-public elements via the setAccessible method of the core reflection API. The --add-opens option can be used, at run time, to do this. It has the same syntax as the --add-exports option:

```
--add-opens <source-module>/<package>=<target-module>(,<target-module>)*
```

where <source-module> and <target-module> are module names and <package> is the name of a package.

The --add-opens option can be used more than once, but at most once for any particular combination of source module and package name. The effect of each instance is to add a qualified open of the named package from the source module

to the target module. This is, essentially, a command-line form of an opens clause in a module declaration, or an invocation of an unrestricted form of the `Module::addOpens` method. As a consequence, code in the target module will be able to use the core reflection API to access all types, public and otherwise, in the named package of the source module so long as the target module reads the source module.

Open packages are indistinguishable from non-exported packages at compile time, so the `--add-opens` option may not be used in that phase.

> **The `--add-exports` and `--add-opens` options must be used with great care. You can use them to gain access to an internal API of a library module, or even of the JDK itself, but you do so at your own risk: If that internal API is changed or removed then your library or application will fail.**

Patching module content

When testing and debugging it is sometimes useful to replace selected class files or resources of specific modules with alternate or experimental versions, or to provide entirely new class files, resources, and even packages. This can be done via the `--patch-module` option, at both compile time and run time. Its syntax is:

```
--patch-module <module>=<file>(<pathsep><file>)*
```

where `<module>` is a module name, `<file>` is the filesystem path name of a module definition, and `<pathsep>` is the host platform's path-separator character.

The `--patch-module` option can be used more than once, but at most once for any particular module name. The effect of each instance is to change how the module system searches for a type in the specified module. Before it checks the actual module, whether part of the system or defined on a module path, it first checks, in order, each module definition specified to the option. A patch path names a sequence of module definitions but it is not a module path, since it has leaky, class-path-like semantics. This allows a test harness, *e.g.*, to inject multiple tests into the same package without having to copy all of the tests into a single directory.

The `--patch-module` option cannot be used to replace `module-info.class` files. If a `module-info.class` file is found in a module definition on a patch path then a warning will be issued and the file will be ignored.

If a package found in a module definition on a patch path is not already exported or opened by that module then it will, still, not be exported or opened. It can be exported or opened explicitly via either the reflection API or the `--add-exports` or `--add-opens` options.

The `--patch-module` option replaces the `-Xbootclasspath:/p` option, which has been removed (see below).

> **The `--patch-module` option is intended only for testing and debugging. Its use in production settings is strongly discouraged.**

Compile time

The `javac` compiler implements the options described above, as applicable to compile time: `--module-source-path`, `--upgrade-module-path`, `--system`, `--module-path`, `--add-modules`, `--limit-modules`, `--add-reads`, `--add-exports`, and `--patch-module`.

The compiler operates in one of three modes, each of which implements additional options.

- *Legacy mode* is enabled when the compilation environment, as defined by the `-source`, `-target`, and `--release` options, is less than or equal to 8. None of the modular options described above may be used.

In legacy mode the compiler behaves in essentially the same way as it does in JDK 8.

- *Single-module mode* is enabled when the compilation environment is 9 or later and the `--module-source-path` option is not used. The other modular options described above may be used; the existing options `-bootclasspath`, `-Xbootclasspath`, `-extdirs`, `-endorseddirs`, and `-XXuserPathsFirst` may not be used.

Single-module mode is used to compile code organized in a traditional package-hierarchical directory tree. It is the natural replacement for simple uses of legacy mode of the form

```
$ javac -d classes -classpath classes -sourcepath src Foo.java
```

If a module descriptor in the form of a `module-info.java` or `module-info.class` file is specified on the command line, or is found on the source path or the class path, then source files will be compiled as members of the module named by that descriptor and that module will be the sole root module. Otherwise if the `--module <module>` option is present then source files will be compiled as members of `<module>`, which will be the root module. Otherwise source files will be compiled as members of the unnamed module, and the root modules will be computed as described above.

It is possible to put arbitrary classes and JAR files on the class path in this mode, but that is not recommended since it amounts to treating those classes and JAR files as part of the module being compiled.

- *Multi-module mode* is enabled when the compilation environment is 9 or later and the `--module-source-path` option is used. The existing `-d` option to name the output directory must also be used; the other modular options described above may be used; the existing options `-bootclasspath`, `-Xbootclasspath`, `-extdirs`, `-endorseddirs`, and `-XXuserPathsFirst` may not be used.

Multi-module mode is used to compile one or more modules, whose source code is laid out in exploded-module directories on the module source path. In this mode the module membership of a type is determined by the position of its source file in the module source path, so each source file specified on the command line must exist within an element of that path. The set of root modules is the set of modules for which at least one source file is specified.

In contrast to the other modes, in this mode an output directory must be specified via the `-d` option. The output directory will be structured as an element of a module path, *i.e.*, it will contain exploded-module directories which themselves contain class and resource files. If the compiler finds a module on the module source path but cannot find the source file for some type in that module then it will search the output directory for the corresponding class file.

In large systems the source code for a particular module may be spread across several different directories. In the JDK itself, *e.g.*, the source files for a module may be found in any one of the directories `src/<module>/share/classes`, `src/<module>/<os>/classes`, or `build/gensrc/<module>`, where `<os>` is the name of the target operating system. To express this in a module source path while preserving module identities we allow each element of such a path to use braces ({ and }) to enclose commas-separated lists of alternatives and a single asterisk (*) to stand for the module name. The module source path for the JDK can then be written as

```
{src/*/{share,<os>}/classes,build/gensrc/*}
```

In both of the modular modes the compiler will, by default, generate various warnings related to the module system; these may be disabled via the option `-Xlint:-module`. More precise control of these warnings is available via the `exports`, `opens`, `requires-automatic`, and `requires-transitive-automatic` keys for the `-Xlint` option.

The new option `--module-version <version>` may be used to specify the version strings of the modules being compiled.

Class-file attributes

A JDK-specific class-file attribute, `ModuleTarget`, optionally records the target operating system and architecture of the module descriptor that contains it. Its format is:

```
ModuleTarget_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 os_arch_index; // index to a CONSTANT_utf8_info structure
}
```

The UTF-8 string in the constant pool at `os_arch_index` has the format `<os>-<arch>`, where `<os>` is typically one of `linux`, `macos`, `solaris`, or `windows`, and `<arch>` is typically one of `x86`, `amd64`, `sparcv9`, `arm`, or `aarch64`.

Packaging: Modular JAR files

The `jar` tool can be used without change to create modular JAR files, since a modular JAR file is just a JAR file with a `module-info.class` file in its root directory.

The `jar` tool implements the following new options to allow the insertion of additional information into module descriptors as modules are packaged:

- `--main-class=<class-name>`, or `-e <class-name>` for short, causes `<class-name>` to be recorded in the `module-info.class` file as the class containing the module's `public static void main` entry point. (This is not a new option; it already records the main class in the JAR file's manifest.)
- `--module-version=<version>` causes `<version>` to be recorded in the `module-info.class` file as the module's version string.
- `--hash-modules=<pattern>` causes hashes of the content of the specific modules that depend upon this module, in a particular set of observable modules, to be recorded in the `module-info.class` file for later use in the validation of dependencies. Hashes are only recorded for modules whose names match the regular expression `<pattern>`. If this option is used then the `---module-path` option, or `-p` for short, must also be used to specify the set of observable modules for the purpose of computing the modules that depend upon this module.
- `--describe-module`, or `-d` for short, displays the module descriptor, if any, of the specified JAR file.

The `jar` tool's `--help` option can be used to show a complete summary of its command-line options.

Two new JDK-specific JAR-file manifest attributes are defined to correspond to the `--add-exports` and `--add-opens` command-line options:

- `Add-Exports: <module>/<package>( <module>/<package>)*`
- `Add-Opens: <module>/<package>( <module>/<package>)*`

The value of each attribute is a space-separated list of slash-separated module-name/package-name pairs. A `<module>/<package>` pair in the value of an `Add-Exports` attribute has the same meaning as the command-line option `--add-exports <module>/<package>=ALL-UNNAMED`. A `<module>/<package>` pair in the value of an `Add-Opens` attribute has the same meaning as the command-line option `--add-opens <module>/<package>=ALL-UNNAMED`.

Each attribute can occur at most once, in the main section of a `MANIFEST.MF` file. A particular pair can be listed more than once. If a specified module was not resolved, or if a specified package does not exist, then the corresponding pair is ignored. These attributes are interpreted only in the main executable JAR file of an application, i.e., in the JAR file specified to the `-jar` option of the Java run-time launcher; they are ignored in all other JAR files.

Packaging: JMOD files

The new JMOD format goes beyond JAR files to include native code, configuration files, and other kinds of data that do not fit naturally, if at all, into JAR files. JMOD files are used to package the modules of the JDK itself; they can also be used by developers to package their own modules, if desired.

JMOD files can be used at compile time and link time, but not at run time. To support them at run time would require, in general, that we be prepared to extract and link native-code libraries on-the-fly. This is feasible on most platforms, though it can be very tricky, and we have not seen many use cases that require this capability, so for simplicity we have chosen to limit the utility of JMOD files in this release.

A new command-line tool, `jmod`, can be used to create, manipulate, and examine JMOD files. Its general syntax is:

```
$ jmod (create|extract|list|describe|hash) <options> <jmod-file>
```

For the `create` subcommand, `<options>` can include the `--main-class`, `--module-version`, `--hash-modules`, and `---module-path` options described above for the `jar` tool, and also:

- `--class-path <path>` specifies a class path whose content will be copied into the resulting JMOD file.
- `--cmds <path>` specifies one or more directories containing native commands to be copied.
- `--config <path>` specifies one or more directories containing configuration files to be copied.
- `--exclude <pattern-list>` specifies files to be excluded, where `<pattern-list>` is a comma-separated list of patterns of the form `<glob-pattern>`, `glob:<glob-pattern>`, or `regex:<regex-pattern>`.
- `--header-files <path>` specifies one or more directories containing C and C++ header files to be copied.
- `--legal-notices <path>` specifies one or more directories containing legal notices to be copied.
- `--libs <path>` specifies one or more directories containing native libraries to be copied.
- `--man-pages <path>` specifies one or more directories containing manual pages to be copied.
- `--target-platform <os>-<arch>` specifies the target operating system and architecture, to be recorded in the `ModuleTarget` attribute of the `module-info.class` file.

The `extract` subcommand accepts a single option, `--dir`, to indicate the directory into which the content of the specified JMOD file should be written. The directory will be created if it does not exist. If this option is not present then the content will be extracted into the current directory.

The `list` subcommand lists the content of the specified JMOD file; the `describe` subcommand displays the module descriptor of the specified JMOD file, in the same format as the `--describe-module` options of the `jar` and `java` commands. These subcommands accept no options.

The `hash` subcommand can be used to hash an existing set of JMOD files. It requires both the `--module-path` and `--hash-modules` options.

The `jmod` tool's `--help` option can be used to show a complete summary of its command-line options.

Link time

The details of the command-line linking tool, `jlink`, are described in JEP 282. At a high level its general syntax is:

```
$ jlink <options> ---module-path <modulepath> --output <path>
```

where the `---module-path` option specifies the set of observable modules to be considered by the linker and the `--output` option specifies the path of the directory that will contain the resulting run-time image. The other `<options>` can include the `---limit-modules` and `---add-modules` options, described above, as well as additional linker-specific options.

The `jlink` tool's `--help` option can be used to show a complete summary of its command-line options.

Run time

The HotSpot virtual machine implements the options described above, as applicable to run time: `--upgrade-module-path`, `--module-path`, `--add-modules`, `--limit-modules`, `--add-reads`, `--add-exports`, `--add-opens`, and `--patch-module`. These options can be passed to the command-line launcher, `java`, and also to the JNI invocation API.

An additional option specific to this phase and supported by the launcher is:

- `--module <module>`, or `-m <module>` for short, specifies the main module of a modular application. This will be the default root module for the purpose of constructing the application's initial module graph. If the main module's descriptor does not indicate a main class then the syntax `<module>/<class>` can be used, where `<class>` names the class that contains the application's `public static void main` entry point.

Additional diagnostic options supported by the launcher include:

- `--list-modules` displays the names and version strings of the observable modules and then exits, in the same manner as `java --version`.
- `--describe-module <module>`, or `-d <module>` for short, displays the module descriptor of the specified module, in the same format as the `jar`

`-d` option and the `jmod describe` subcommand, and then exits.

- `--validate-modules` validates all the observable modules, checking for conflicts and other potential errors, and then exits.
- `--dry-run` initializes the virtual machine and loads the main class but does not invoke the main method; this is useful for validating the configuration of the module system.
- `--show-module-resolution` causes the module system to describe its activities as it constructs the initial module graph.
- `-Dsun.reflect.debugModuleAccessChecks` causes a thread dump to be shown whenever an access check in the `java.lang.reflect` API fails with an `IllegalAccessException` or an `InaccessibleObjectException`. This is useful for debugging when the underlying reason for a failure is hidden because the exception is caught and not re-thrown.
- `-Xlog:module+[load|unload][=[debug|trace]]` causes the VM to log debug or trace messages as modules are defined and changed in the run-time module graph. These options generate voluminous output during startup.
- `-verbose:module` is a shorthand for `-Xlog:module+load -Xlog:module+unload`.
- `-Xlog:init`=debug causes a stack trace to be displayed if the initialization of the module system fails.
- `--version`, `--show-version`, `--help`, and `--help-extra` display the same information and work in the same way as the existing `-version`, `-show-version`, `-help`, and `-Xhelp` options, respectively, except that they write the help text to the standard output stream rather than the standard error stream.

The stack traces generated for exceptions at run time have been extended to include, when present, the names and version strings of relevant modules. The detail strings of exceptions such as `ClassCastException`, `IllegalAccessException`, and `IllegalAccessError` have also been updated to include module information.

The existing `-jar` option has been enhanced so that if the manifest file of the JAR file being launched contains a `Launcher-Agent-Class` attribute then the JAR file is launched as both an application and as an agent for that application. This allows `java -jar foo.jar` to be used in place of the more verbose `java -javaagent:foo.jar -jar foo.jar`.

Relaxed strong encapsulation

In this release the strong encapsulation of some of the JDK's packages is relaxed by default, as permitted by the Java SE 9 Platform Specification. This relaxation is controlled at run time by a new launcher option, `--illegal-access`, which works as follows:

- `--illegal-access=permit` opens each package in each module in the run-time image to code in all unnamed modules, *i.e.*, to code on the class path, if that package existed in JDK 8. This enables both static access, *i.e.*, by compiled bytecode, and deep reflective access, via the platform's various reflection APIs.

  The first reflective-access operation to any such package causes a warning to be issued, but no warnings are issued after that point. This single warning describes how to enable further warnings. This warning cannot be suppressed.

  This mode is the default in JDK 9. It will be phased out in a future release and, eventually, removed.
- `--illegal-access=warn` is identical to `permit` except that a warning message is issued for each illegal reflective-access operation.
- `--illegal-access=debug` is identical to `warn` except both a warning message and a stack trace are issued for each illegal reflective-access operation.
- `--illegal-access=deny` disables all illegal-access operations except for those enabled by other command-line options, e.g., `--add-opens`.

  This mode will become the default in a future release.

When deny becomes the default illegal-access mode then `permit` will likely remain supported for at least one release, so that developers can continue to migrate their code. The `permit`, `warn`, and `debug` modes will, over time, be removed, as will the `--illegal-access` option itself. (For launch-script compatibility the unsupported modes will most likely just be ignored, after issuing a warning to that effect.)

The default mode, `--illegal-access=permit`, is intended to make you aware when you have code on the class path that reflectively accesses some JDK-internal API at least once. To prepare for the future you can use the `warn` or `debug` modes to learn about all such accesses. For each library or framework on the class path that requires illegal access you have two options:

- If the component's maintainers have already released a new, fixed version that no longer uses JDK-internal APIs then you can consider upgrading to that version.
- If the component still needs to be fixed then we encourage you to contact its maintainers and ask them to replace their use of JDK-internal APIs with proper exported APIs, if available.

If you must continue to use a component that requires illegal access then you can eliminate the warning messages by using one or more `--add-opens` options to open just those internal packages to which access is required.

To verify that your application is ready for the future, run it with `--illegal-access=deny` along with any necessary `--add-opens` options. Any remaining illegal-access errors will most likely be due to static references from compiled code to JDK-internal APIs. You can identify those by running the `jdeps` tool with the `--jdk-internals` option. (The run-time system does not issue warnings for illegal static-access operations because that would require deep VM changes and degrade performance.)

The warning message issued when an illegal reflective-access operation is detected has the following form:

```
WARNING: Illegal reflective access by $PERPETRATOR to $VICTIM
```

where:

- $PERPETRATOR is the fully-qualified name of the type containing the code that invoked the reflective operation in question plus the code source (i.e., JAR-file path), if available, and
- $VICTIM is a string that describes the member being accessed, including the fully-qualified name of the enclosing type

In the default mode, `--illegal-access=permit`, at most one of these warning messages will be issued, accompanied by additional instructive text. Here is an example, from running Jython:

```
$ java -jar jython-standalone-2.7.0.jar
WARNING: An illegal reflective access operation has occurred
WARNING: Illegal reflective access by jnr.posix.JavaLibCHelper (file:/tmp/jython-standalone-2.7.0.jar) to method sun.nio.ch.S
WARNING: Please consider reporting this to the maintainers of jnr.posix.JavaLibCHelper
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
Jython 2.7.0 (default:9987c746f838, Apr 29 2015, 02:25:11)
[OpenJDK 64-Bit Server VM (Oracle Corporation)] on java9
Type "help", "copyright", "credits" or "license" for more information.
>>> ^D
```

The run-time system makes a best-effort attempt to suppress duplicate warnings for the same $PERPETRATOR and $VICTIM.

An extended example

Suppose we have an application module, `com.foo.bar`, which depends upon a library module, `com.foo.baz`. If we have the source code for both modules in the module-path directory `src`:

```
src/com.foo.bar/module-info.java
src/com.foo.bar/com/foo/bar/Main.java
src/com.foo.baz/module-info.java
src/com.foo.baz/com/foo/baz/BazGenerator.java
```

then we can compile them, together:

```
$ javac --module-source-path src -d mods $(find src -name '*.java')
```

The output directory, `mods`, is a module-path directory containing exploded, compiled definitions of the two modules:

```
mods/com.foo.bar/module-info.class
mods/com.foo.bar/com/foo/bar/Main.class
mods/com.foo.baz/module-info.class
mods/com.foo.baz/com/foo/baz/BazGenerator.class
```

Assuming that the `com.foo.bar.Main` class contains the application's entry point, we can run these modules as-is:

```
$ java -p mods -m com.foo.bar/com.foo.bar.Main
```

Alternatively, we can package them up into modular JAR files:

```
$ jar --create -f mlib/com.foo.bar-1.0.jar \
      --main-class com.foo.bar.Main --module-version 1.0 \
      -C mods/com.foo.bar .
$ jar --create -f mlib/com.foo.baz-1.0.jar \
      --module-version 1.0 -C mods/com.foo.baz .
```

The `mlib` directory is a module-path directory containing the packaged, compiled definitions of the two modules:

```
$ ls -l mlib
-rw-r--r-- 1501 Sep  6 12:23 com.foo.bar-1.0.jar
-rw-r--r-- 1376 Sep  6 12:23 com.foo.baz-1.0.jar
```

We can now run the packaged modules directly:

```
$ java -p mlib -m com.foo.bar
```

`jtreg` enhancements

The jtreg test harness supports a new declarative tag, `@modules`, to express a test's dependences upon the modules in the system being tested. It takes a series of space-separated arguments, each of which can be of the form

- `<module>`, where `<module>` is a module name, to indicate that the specified module must be present;
- `<module>/<package>`, to indicate that the specified module must be present and the specified package must be exported to the test's module; or
- `<module>/<package>:<flag>`, to indicate that the specified module must be present and, if the flag is open then the specified package must be opened to the test's module, or else if the flag is +open then the specified package must be both exported and opened to the test's module.

A default set of `@modules` arguments, which will be used for all tests in a directory hierarchy that do not include such a tag, can be specified as the value of the

`modules` property in a `TEST.ROOT` file or in any `TEST.properties` file.

The existing `@compile` tag accepts a new option, `/module=<module>`. This has the effect of invoking `javac` with the `--module <module>` option, defined above, to compile the specified classes as members of the indicated module.

Class loaders

The Java SE Platform API historically specified two class loaders: The *bootstrap class loader*, which loads classes from the bootstrap class path, and the *system class loader*, which is the default delegation parent for new class loaders and, typically, the class loader used to load and start the application. The specification does not mandate the concrete types of either of these class loaders, nor their precise delegation relationship.

The JDK has, since the 1.2 release, implemented a three-level hierarchy of class loaders, where each loader delegates to the next:

- The application class loader, an instance of `java.net.URLClassLoader`, loads classes from the class path and is installed as the system class loader unless an alternate system loader is specified via the system property `java.system.class.loader`.
- The extension class loader, also an instance of `URLClassLoader`, loads classes available via the extension mechanism and, also, some resources and service providers built-in to the JDK. (This loader is not mentioned explicitly in the Java SE Platform API Specification.)
- The bootstrap class loader, which is implemented solely within the virtual machine and is represented by `null` in the `ClassLoader` API, loads classes from the bootstrap class path.

JDK 9 retains this three-level hierarchy, in order to preserve compatibility, while making the following changes to implement the module system:

- The application class loader is no longer an instance of `URLClassLoader` but, rather, of an internal class. It is the default loader for named modules that are neither Java SE nor JDK modules.
- The extension class loader is no longer an instance of `URLClassLoader` but, rather, of an internal class. It no longer loads classes via the extension mechanism, which was removed by JEP 220. It does, however, define selected Java SE and JDK modules, about which more below. In its new role this loader is known as the *platform class loader*, it is available via the new `ClassLoader::getPlatformClassLoader` method, and it will be required by the Java SE Platform API Specification.
- The bootstrap class loader is implemented in both library code and within the virtual machine, but for compatibility it is still represented by `null` in the `ClassLoader` API. It defines the core Java SE and JDK modules.

The platform class loader is retained not only for compatibility but, also, to improve security. Types loaded by the bootstrap class loader are implicitly granted all security permissions (`AllPermission`), but many of these types do not actually require all permissions. We have *de-privileged* modules that do not require all permissions by defining them to the platform loader rather than the bootstrap class loader, and by granting them the permissions they actually need in the default security policy file. The Java SE and JDK modules defined to the platform class loader are:

```
java.activation*           jdk.accessibility
java.compiler*             jdk.charsets
java.corba*                jdk.crypto.cryptoki
java.scripting             jdk.crypto.ec
java.se                    jdk.dynalink
java.se.ee                 jdk.incubator.httpclient
java.security.jgss         jdk.internal.vm.compiler*
java.smartcardio           jdk.jsobject
java.sql                   jdk.localedata
java.sql.rowset            jdk.naming.dns
java.transaction*          jdk.scripting.nashorn
java.xml.bind*             jdk.security.auth
java.xml.crypto            jdk.security.jgss
java.xml.ws*               jdk.xml.dom
java.xml.ws.annotation*    jdk.zipfs
```

(An asterisk, `'*'`, in these lists indicates an upgradeable module.)

JDK modules that provide tools or export tool APIs are defined to the application class loader:

```
jdk.aot                    jdk.jdeps
jdk.attach                 jdk.jdi
jdk.compiler               jdk.jdwp.agent
jdk.editpad                jdk.jlink
jdk.hotspot.agent          jdk.jshell
jdk.internal.ed            jdk.jstatd
jdk.internal.jvmstat       jdk.pack
jdk.internal.le            jdk.policytool
jdk.internal.opt           jdk.rmic
jdk.jartool                jdk.scripting.nashorn.shell
jdk.javadoc                jdk.xml.bind*
jdk.jcmd                   jdk.xml.ws*
jdk.jconsole
```

All other Java SE and JDK modules are defined to the bootstrap class loader:

```
java.base                  java.security.sasl
java.datatransfer          java.xml
java.desktop               jdk.httpserver
java.instrument            jdk.internal.vm.ci
```

```
java.logging                jdk.management
java.management             jdk.management.agent
java.management.rmi         jdk.naming.rmi
java.naming                 jdk.net
java.prefs                  jdk.sctp
java.rmi                    jdk.unsupported
```

The three built-in class loaders work together to load classes as follows:

- The application class loader first searches the named modules defined to all of the built-in loaders. If a suitable module is defined to one of these loaders then that loader will load the class. If a class is not found in a named module defined to one of these loaders then the application class loader delegates to its parent. If a class is not found by its parent then the application class loader searches the class path. Classes found on the class path are loaded as members of this loader's unnamed module.
- The platform class loader searches the named modules defined to all of the built-in loaders. If a suitable module is defined to one of these loaders then that loader will load the class. (The platform class loader can, consequently, now delegate to the application class loader, which can be useful when a module on the upgrade module path depends upon a module on the application module path.) If a class is not found in a named module defined to one of these loaders then the platform class loader delegates to its parent.
- The bootstrap class loader searches the named modules defined to itself. If a class is not found in a named module defined to the bootstrap loader then the bootstrap class loader searches the files and directories added to the bootstrap class path via the `-Xbootclasspath/a` option. Classes found on the bootstrap class path are loaded as members of this loader's unnamed module.

The application and platform class loaders delegate to their respective parent loaders in order to ensure that the bootstrap class path is still searched when a class is not found in a module defined to one of the built-in loaders.

Removed: Bootstrap class-path options

In earlier releases the `-Xbootclasspath` option allows the default bootstrap class path to be overridden, and the `-Xbootclasspath/p` option allows a sequence of files and directories to be prepended to the default path. The computed value of this path is reported via the JDK-specific system property `sun.boot.class.path`.

With the module system in place the bootstrap class path is empty by default, since bootstrap classes are loaded from their respective modules. The `javac` compiler only supports the `-Xbootclasspath` option in legacy mode, the `java` launcher no longer supports either of these options, and the system property `sun.boot.class.path` has been removed.

The compiler's `--system` option can be used to specify an alternate source of system modules, as described above, and its `-release` option can be used to specify an alternate platform version, as described in JEP 247 (Compile for Older Platform Versions). At run time the `--patch-module` option, mentioned above, can be used to inject content into modules in the initial module graph.

A related option, `-Xbootclasspath/a`, allows files and directories to be appended to the default bootstrap class path. This option, and the related API in the `java.lang.instrument` package, is sometimes used by instrumentation agents, so for compatibility it is still supported at run time. Its value, if specified, is reported via the JDK-specific system property `jdk.boot.class.path.append`. This option can be passed to the command-line launcher, `java`, and also to the JNI invocation API.

Testing

Many existing tests were affected by the introduction of the module system. In JDK 9 the `@modules` tag, described above, was added to the unit and regression tests as needed, and tests that used the `-Xbootclasspath/p` option or assumed that the system class loader is a `URLClassLoader` were updated.

There is, of course, an extensive set of unit tests for the module system itself. In the JDK 9 source forest the run-time tests are in the test/jdk/modules directory of the `jdk` repository and the runtime/modules directory of the `hotspot` repository; the compile-time tests are in the tools/javac/modules directory of the `langtools` repository.

Early-access builds containing the changes described here were available throughout the development of the module system. Members of the Java community were strongly encouraged to test their tools, libraries, and applications against these builds to help identify compatibility issues.

Risks and Assumptions

The primary risks of this proposal are ones of compatibility due to changes to existing language constructs, APIs, and tools.

Changes due primarily to the introduction of the Java Platform Module System (JSR 376) include:

- Applying the `public` modifier to an API element no longer guarantees that the element will be everywhere accessible. Accessibility now depends also upon whether the package containing that element is exported or opened by its defining module, and whether that module is readable by the module containing the code that is attempting to access it. For example, code of the following form might not work correctly:

```
Class<?> c = Class.forName(...);
if (Modifier.isPublic(c.getModifiers()) {
```

```
        // Assume that c is accessible
    }
```

- If a package is defined in both a named module and on the class path then the package on the class path will be ignored. The class path can, therefore, no longer be used to augment packages that are built into the environment. The `javax.transaction` package, *e.g.*, is defined by the `java.transaction` module, so the class path will not be searched for types in that package. This restriction is important to avoid splitting packages across class loaders and across modules. At compile time and run time the upgrade module path can be used to upgrade modules that are built-in into the environment. The `--patch-module` option can be used for other ad-hoc patching.
- The `ClassLoader::getResource*` methods can no longer be used to locate JDK-internal resources other than class files. Module-private non-class resources can be read via the `Class::getResource*` methods, the `Module::getResourceAsStream` method or via the `jrt:` URL scheme and filesystem defined in JEP 220.
- The `java.lang.reflect.AccessibleObject::setAccessible` method cannot be used to gain access to public members of packages that are not exported or opened by their defining modules, or to non-public members of packages that are not opened by their defining modules; in either case, an `InaccessibleObjectException` will be thrown. If a framework library, such as a serializer, needs access to such members at run time then the relevant packages must be opened to the framework module by declaring the containing module to be open, by declaring that package to be open, or by opening that package via the `--add-opens` command-line option.
- JVM TI agents can no longer instrument Java code that runs early in the startup of the run-time environment. The `ClassFileLoadHook` event, in particular, is no longer sent during the primordial phase. The `VMStart` event, which signals the beginning of the start phase, is only posted after the the VM is initialized to the point where it can load classes in modules other than `java.base`. Two new capabilities, `can_generate_early_class_hook_events` and `can_generate_early_vmstart`, can be added by agents that are carefully written to handle events early in VM initialization. More details can be found in the updated description of the class file load hook event and the start event.
- The == syntax in security policy files has been revised to augment the permissions granted to standard and JDK modules rather than override them. Applications that override other aspects of the JDK's default policy file therefore do not need to copy the default permissions granted to the standard and JDK modules.

Modules that define Java EE APIs, or APIs primarily of interest to Java EE applications, have been deprecated and will be removed in a future release. They are not resolved by default for code on the class path:

- The default set of root modules for the unnamed module is based, in JDK 9, upon the `java.se` module rather than the `java.se.ee` module. Thus, by default, code in the unnamed module will not have access to APIs in the following modules:

  ```
  java.activation
  java.corba
  java.transaction
  java.xml.bind
  java.xml.ws
  java.xml.ws.annotation
  ```

  This is an intentional, if painful, choice, driven by two goals:

  - To avoid unnecessary conflicts with popular libraries that define types in some of the same packages. The widely-used `jsr305.jar`, *e.g.*, defines annotation types in the `javax.annotation` package, which is also defined by the `java.xml.ws.annotation` module.
  - To make it easier for existing application servers to migrate to JDK 9. Application servers often override the content of one or more of these modules, and in the near term they are most likely to do so by continuing to place the necessary non-modular JAR files on the class path. If these modules were resolved by default then the maintainers of application servers would have to take awkward actions to exclude them in order to override them.

  These modules are still part of JDK 9. Code on the class path can be granted access to one or more of these modules, as needed, via the `--add-modules` option.

The run-time behavior of some Java SE APIs has changed, though in ways that continue to honor their existing specifications:

- The application and platform class loaders are no longer instances of the `java.net.URLClassLoader` class, as noted above. Existing code that invokes `ClassLoader::getSystemClassLoader` and blindly casts the result to `URLClassLoader`, or does the same thing with the parent of that class loader, might not work correctly.
- Some Java SE types have been de-privileged and are now loaded by the platform class loader rather than the bootstrap class loader, as noted above. Existing custom class loaders that delegate directly to the bootstrap class loader might not work correctly; they should be updated to delegate to the platform class loader, which is easily available via the new `ClassLoader::getPlatformClassLoader` method.

- Instances of `java.lang.Package` created by the built-in class loaders for packages in named modules do not have specification or implementation versions. In previous releases this information was read from the manifest of `rt.jar`. Existing code that expects the `Package::getSpecification*` or `Package::getImplementation*` methods always to return non-null values might not work correctly.

There are several source-incompatible Java SE API changes:

- The `java.lang` package includes two new top-level classes, `Module` and `ModuleLayer`. The `java.lang` package is implicitly imported on demand (*i.e.*, `import java.lang.*`). If code in an existing source file imports some other package on demand, and that package declares a `Module` or `ModuleLayer` type, and the existing code refers to that type, then the file will not compile without change.
- The `java.lang.instrument.Instrumentation` interface declares two new abstract methods, `redefineModule` and `isModifiableModule`. This interface is not intended to be implemented outside of the `java.instrument` module. If there are external implementations then they will not compile on JDK 9 without change.
- The five-parameter `transform` method declared in the `java.lang.instrument.ClassFileTransformer` interface is now a default method. The interface now also declares a new `transform` method that makes the relevant `java.lang.reflect.Module` object available to the transformer when instrumenting classes at load time. Existing compiled code will continue to run, but existing source code that uses the existing five-parameter transform method as a functional interface will no longer compile.

Finally, changes due to revisions to JDK-specific APIs and tools include:

- Most of the JDK's internal APIs are inaccessible by default at compile time, as described in JEP 260. Existing code that compiled against these APIs with warnings in previous releases will no longer compile. A workaround is to break encapsulation via the `--add-exports` option, defined above.
- Selected critical internal APIs in the `sun.misc` and `sun.reflect` packages have been moved to the `jdk.unsupported` module, as described in JEP 260. Non-critical internal APIs in these packages, such as `sun.misc.BASE64{De,En}coder`, have been either moved or removed.
- If a security manager is present then the run-time permission `accessSystemModules` is required in order to access JDK-internal resources via the `ClassLoader::getResource*` or `Class::getResource*` methods; in previous releases, permission to read the file `${java.home}/lib/rt.jar` was required.
- The `-Xbootclasspath` and `-Xbootclasspath/p` options have been removed, as noted above. At compile time, the new `--release` option can be used to specify an alternate platform version (see JEP 247). At run time, the new `--patch-module` option, described above, can be used to inject content into system modules.
- The JDK-specific system property `sun.boot.class.path` has been removed, since the bootstrap class path is empty by default. Existing code that uses this property might not work correctly.
- The JDK-specific annotation `@jdk.Exported`, introduced by JEP 179, has been removed since the information it conveys is now recorded in the `exports` declarations of module descriptors. We have seen no evidence of this annotation being used by tools outside of the JDK.
- The `META-INF/services` resource files previously found in `rt.jar` and other internal artifacts are not present in the corresponding system modules, since service providers and dependences are now declared in module descriptors. Existing code that scans for such files might not work correctly.
- The JDK-specific system property `file.encoding` can be set on the command line via the `-D` option, as before, but it must specify a charset defined in the base module. If any other charset is specified then the run-time system will fail to start. Existing launch scripts that specify such charsets might not work correctly.
- The `com.sun.tools.attach` API can no longer be used, by default, to attach an agent to the current process or an ancestor of the current process. Such attachment operations can be enabled by setting the system property `jdk.attach.allowAttachSelf` on the command line.
- The dynamic loading of JVM TI agents will be disabled by default in a future release. To prepare for that change we recommend that applications that allow dynamic agents start using the option `-XX:+EnableDynamicAgentLoading` to enable that loading explicitly. The option `-XX:-EnableDynamicAgentLoading` disables dynamic agent loading.

Dependences

JEP 200 (The Modular JDK) originally defined the modules present in the JDK in an XML document, as an interim measure. This JEP moved those definitions to proper module descriptors, *i.e.*, `module-info.java` and `module-info.class` files, and the `modules.xml` file in the root source-code repository was removed.

The initial implementation of JEP 220 (Modular Run-Time Images) in JDK 9 used a custom build-time tool to construct JRE and JDK images. This JEP replaced that tool with the `jlink` tool.

Modular JAR files can also be Multi-Release JAR files, per JEP 238.