# Java Hybrid Module System Specification

2022-12-11

# 1 Introduction

The reader should be familiar with *Java Platform Module System* [JPMS]. Java version 9 or later is assumed.

The *Java Hybrid Module System* (*JHMS*) is a way of compiling and running Java code similar to JPMS, except that it adds version support: the goal of JHMS is to provide a consistent view of the dependencies:

*The types, fields, and methods that are visible and accessible,*
*and the versions of these are the same at compile time and run time.*

"Run time" means both unit testing time and application run time. JHMS is able to achieve the goal using techniques similar to [OSGi].

This document defines JHMS from developer's point of view in 2 Hybrid Modules, and guides implementations in 3 Implementation. 4 Relaxations defines various relaxations an implementation is allowed to make to be more flexible in the real world. Motivations and discussions are deferred to a supplemental document [SUPL]

# 2 Hybrid Modules

This section contains the specification of the Java Hybrid Module System (JHMS) from the point of view of the application/library developer.

There are 2 types of modules in JHMS:

1. Platform modules that are part of the Java SE Platform
2. Hybrid modules that are user-defined modules packaged in modular JARs

Relaxation (§4.2) allows a third kind: *automatic hybrid modules*, that are backed by plain (non-modular) JARs, and may be helpful during migration to hybrid modules. "hybrid modules" in this specification always refers to explicit non-automatic hybrid modules, unless otherwise stated.

This specification describes hybrid modules: how to declare them, how they access platform modules, etc. It does not go into details on *how* the given semantics can be achieved on the Java platform: that's deferred to the 3 Implementation section.

Hybrid modules are identical to JPMS modules in all aspects, including requirements on source code, compilation, packaging, linking, testing, launching, and run time, unless otherwise stated in this specification.

Hybrid modules mainly affect the following areas:

1. Module declarations, detailed in 2.1 Module declaration.
2. Compilation, detailed in 2.4 Compilation.

3. Packaging, detailed in 2.5 Packaging.
4. Bootstrapping the execution of the hybrid modules, detailed in 2.7 Launching application and 2.8 Hybrid module container.
5. Run time, as derived from 2.3 Observability, visibility, and accessibility.

## 2.1 Module declaration, descriptor, and version

Any JPMS module declaration (*module-info.java*), or descriptor (*module-info.class*), is also a valid hybrid module declaration (descriptor), however:

1. The *open* modifier on the module, and the *opens* directives, are ignored.
2. The *provides* and *uses* directives are ignored.
3. The *requires* directives with the *static* modifier are ignored at run time.

The semantics of *exports* in JHMS is roughly that of *exports+opens* in JPMS, and the effect of not exporting a package is that it will not be *visible* outside the module. The precise semantics are defined in (§2.2).

The hybrid module version is determined by compilation (§2.4) and packaging (§2.5), and *must* be set according to *ModuleDescriptor.Version*. A hybrid module is allowed to have no version, in case the version is said to be *null*, *empty*, or *absent*.

The version of platform modules are ignored in JHMS.

The module descriptor of a module *M* contains

1. The version of each module it *requires* at the time *M* was compiled. The version of such a module *N* is
    a. The version of *N* used during compilation of *M*, if *N* is a hybrid module.
    b. The Java compiler version used during compilation of *M*, if *N* is a platform module, e.g. 11.0.3.
2. The main class, if specified during packaging (§2.5).

A "*module*" will be used to refer to a specific version of a hybrid module, *or* a platform module. Modules are identified by a name and version pair of the form *M@V*, even though…

a. the version should be ignored for platform modules, and
b. the version may be absent for hybrid modules.

## 2.2 Readability and the module graph

The modules *read* by a module *M@V*[1] is the set of these:

1. *M@V*
2. For each module *N* that *M requires* (non-*static*): the *read closure* of *N@U*, where *U* is the version *N* had at the time *M@V* was compiled.

[1] As a reminder, from (§2.1), ignore the version if the module is a platform module.

The *read closure* of a module *N@U* is the set of these modules:

1. *N@U*.
2. The read closure of any module *K@W* that *N requires transitive* (non-*static*), where *W* is the version *K* had at the time *N@U* was compiled.

It is an error if the read closure…

1. contains multiple versions of a module, or
2. contains more than one module that exports the same package.

The *module graph* is the result of *resolving* a set of *root modules R* as follows:

1. To resolve a module *M* at version *V* (*M@V*)
    a. Add a vertex to the module graph identified by the module name and version (*M@V*).
    b. For each *requires* of a module *N@W*[2]:
        i. Resolve *N@W* unless it has already been resolved.
        ii. Add a *requires edge* from *M@V* to *N@W*, annotated with whether the *requires* is *transitive* or not.
        iii. For each module *K@U* in the read closure of *N@W*:
            1. Do nothing if there is a *read edge* from *M@V* to *K@U* already, otherwise
            2. fail if there is a *read edge* from *M@V* to a different version of *K*, otherwise
            3. add a *read edge* from *M@V* to *K@U*.

The graph made from just the read edges is sometimes also called the *readability graph*.

The root modules at compile time is the module being compiled. The root modules at run time is the *main module* determined in (§2.7) and (§2.8).

The *root graph* of a module *M* is the module graph resulting from resolving with *M* as the sole root module.

Readability in JPMS, and readability of platform modules in JHMS (which JHMS cannot change), is defined in [READ] and consistent with the above when 1. references to versions are ignored from above (as they should be according to (§2.1)), and 2. automatic modules are ignored (which is deferred to relaxation (§4.2)).

A JHMS implementation *should* provide a way to print the module graph as dot (graphviz).

## 2.3 Observability, visibility, and accessibility

Observability, visibility, and access control is covered in the Java Language Specification [JLS11] (§6.6, §7.3, §7.4.3), and familiarity is assumed. The *javac* compiler adds additional constraints at compile time. This section defines observability, visibility, and accessibility only

[2] The version of a required module is described in (§2.1).

as it relates to modules in JHMS. The rules are the same at compile and run time, unless otherwise stated.

### 2.3.1 Observability

A module *N* at version *V* is *observable* if

1. *N* is a platform module,[3] or
2. there is exactly one modular JAR with module name *N* and version *V* on the *hybrid module path*. The hybrid module path specifies a list of JAR files and parent directories of JAR files.
    a. An implementation *should* provide *--module-path*/*-p* options to specify a colon (:) separated list of paths to locate hybrid modules when launching a JHMS application, similar to JPMS (§2.7).

All modules in the readability graph must be observable at compile time. If such a hybrid module *N*@*V* is unobservable, it causes compilation to fail with e.g. “error: module not found: *N*”.

All modules in the module graph must be observable at run time. If such a hybrid module *N*@*V* is not observable, a FindException “Hybrid module *N*@*V* not found on the module path” is thrown, analogous to FindException “Module *N* not found” thrown for an readable but unobservable platform module in JPMS.

If an unreadable hybrid module *N* is observable, a reference (in module *M*) to a package *P* in *N* causes compilation to fail with e.g. “error: package *P* is not visible” and “package *P* is declared in module *N*, but module *M* does not read it”. Unless otherwise stated, we will assume readable hybrid modules are observable.

### 2.3.2 Visibility

A package *P* of (platform or hybrid) module *N* is *visible* to code in hybrid module *M* iff

1. *N* = *M*, or
2. *P* is exported (to *M* if qualified) by *N* and *N* is readable by *M*.

The visibility of a type follows the visibility of its package.

Referring to an invisible type causes compilation to fail with “error: *cannot find symbol*”. It is a compile and run time error if two types *T* and *T’* of modules *N* and *N’* (*N* ≠ *N’*) are visible to *M* with the same package *P* = *P’* (split package). *Class.forName()* returns the *Class* object for a visible type, and throws a *ClassNotFoundException* for an invisible type.

### 2.3.3 Accessibility

A type *T* is *accessible* to a hybrid module if it is visible and public. A member *F* in type *T* is *accessible* to a hybrid module if

1. *T* is accessible, and

[3] The version *V* is ignored.

2. either *F* is public, or *F* is *protected* and the access is from a subclass of *T*.

A member *F* in type *T* is also accessible to a hybrid module via reflection at run time if

1. *T* is visible, and
2. *setAccessible(true)* is called on *F*.

Referring to an inaccessible type *T* in package *P* causes compilation to fail with "*error:* T *is not public in* P*; cannot be accessed from outside package*". Referring to an inaccessible private field *F* of public type *T* causes compilation to fail with "*error:* F *has private access in* T". *Field::get* returns the value of the field as an *Object* for an accessible field, or throws an *IllegalAccessException* for an inaccessible field.

### 2.3.4 Summary

This section can be summarized as:

> *Accessibility is granted with pre-JPMS rules for readable packages[4], and is otherwise denied by invisibility.*

## 2.4 Compilation

Compilation must include a module declaration in *module-info.java*.

The version *should* be set with *--module-version* during compilation (or packaging (§2.5)) according to *ModuleDescriptor.Version* (§2.1).

All readable hybrid modules (§2.2) must be made observable (§2.3) during compilation, i.e. put on *--module-path* as a modular JAR (that contains a *module-info.class*).

When compiling hybrid module *M*, the modules it *requires* in the module declaration (§2.1) have their version stored in the module descriptor (*module-info.class*) as the *compiled version*, if they have a non-null version. The indirect transitively required modules (§2.1) are not stored in the module descriptor.

If a module *M* reads both *N* and *K*, and *N* has a *requires transitive K*, then

1. the version $V_M$ of *K* during the compilation of *M*, must be equal to
2. the version $V_N$ of *K* during the compilation of *N*.

Note that this is the same error as having two different versions of a module readable by *M*, and forbidden in (§2.2). It's just that *javac* does not enforce this. A JHMS implementation *should* provide a *javac wrapper* (§3.1) that would fail compilation if this is violated.

## 2.5 Packaging

A hybrid module is packaged into a *hybrid modular JAR* similar to how a JPMS module is packaged into a modular *JAR* [MJAR].

[4] A readable package *P* is a package exported from a readable module to the hybrid module.

The *main class* of the hybrid module can be set with the *jar* command's *--main-class/-e*.

The version *may* be set with *--module-version* when packaging the modular JAR, and if so *should* set it to a version according to *ModuleDescriptor.Version*, see (§2.1, §2.4).

## 2.6 Linking

A JHMS implementation can use *jlink* to reduce the set of platform modules as long as it still includes the platform modules required by the application. *jlink must not* be used to include any library or application modules.

## 2.7 Launching application

A JHMS implementation *must* provide a way to launch a *hybrid module application* with *java* in a way that is similar to launching module mode when passing *--module*/*-m* to *java*, for instance with a *java wrapper* (§3.2). It must support the following:

1. There must be a way to specify the paths to look for hybrid modules, preferably *--module-path*/*-p*. All modules in the module graph (§2.2) must be made observable (§2.3), i.e. added to the hybrid module path.
2. There must be a way to specify the main module and main class, preferably *--module*/*-m*. If the main class is not specified explicitly, the module must have a main class (§2.5).

The *main class* of a hybrid module application must be in an exported package.[5]

The current thread's context class loader must be the *main* hybrid module class loader upon invocation of the *main* method, by default.

## 2.8 Hybrid module container

A JHMS implementation *should* provide a library for *bootstrapping* a *hybrid module container* that can be used to execute hybrid module code according to this specification. The rest of this section describes such a library.

It must be possible to create a new hybrid module container:

1. The hybrid module container must be created with a module path, as in (§2.7).
2. The hybrid module container must contain an API for resolving a root hybrid module according to (§2.2). The version of the hybrid module may be omitted[6] if there is only one version of the hybrid module on the module path. An interface to the root hybrid module is returned, and must contain:

---

[5] Unlike JPMS which allows *main* in unexported packages. This seemed counterintuitive.

[6] Not to be confused with an absent/*null* version: The version of the root hybrid module must be omitted, or absent/*null*, or a (non-*null*) *String*.

a. A method to return a *Class* object of a *public* class in an unqualified exported package.
b. It *must* contain a convenience method for invoking a *public static void main(String…)* method in such a class (a).[7] When invoking such a method, the container must *not* change the thread's context class loader, see §2.7.

## 2.9 Resource loading

The *Class::getResource(name)* and *Class::getResourceAsStream(name)* instance methods invoked on a class instance defined in a hybrid module *M*, e.g. with *instance.getClass().getResource()*, provides a way to load resources in an analogous way to how classes are loaded. Both methods have similar semantics:

1. *name* is resolved to an absolute name *qname* as follows:
    a. If *name* starts with a "*/*", then *qname* is the suffix. Otherwise, *qname* is *pkg/name*, where *pkg* is the package name of the class with "." replaced by "*/*".
2. The part before the last "/" of *qname*, with all "/" replaced by "." …
    a. may be a valid package name exported by another hybrid module *N* readable to *M*, may be a package of *M* (let *N* = *M)*, or may be an invalid package name (let *N* = *M*). The resource is loaded from *N*, returning *null* if not found.
    b. may be a valid package name exported by a platform module *N* readable by *M*…
        i. If *qname* ends in "*.class*" or the package is open,[8] the resource is returned if it exists.
        ii. Otherwise, null is returned.

If *getResource(name)* / *getResourceAsStream(name)* is called on an instance of a platform class, then

*3.* *name* is resolved to an absolute name *qname* as in (1) above.
4. The resource is loaded from the platform module if it exists in the module and at least one of the following is true, otherwise *null* is returned.
    a. *qname* ends in "*.class*", or
    b. the part before the last "/" of *qname*, with all "/" replaced by ".", is *not* a valid package name, or
    c. the package is not a package of the platform module, or
    d. the package is open.

## 2.10 Unit testing

Black-box unit testing of a hybrid module *M* produces a hybrid module *T*:

1. The hybrid modular JAR *T.jar* contains everything in *M.jar*, in addition to test classes and resources, and an updated module declaration with additional dependencies (*requires*).

---

[7] Since the *main* method may be in an unexported package, it would not otherwise be accessible without special support from the implementation.
[8] In OpenJDK 17, no platform modules contain non-*.class* files in *open* packages, so the "or the package is open" part can be ignored.

2. The tests are written to be run by a test framework. The test framework JARs must be converted to and compatible with hybrid modular JARs.
3. A test framework specific *booter* creates a hybrid module container, invokes a *main* stub that sets up the test framework and invokes the test framework.

For a working example, see Implementation, Unit testing (§3.7).

There is an alternative black-box unit testing strategy, which may work with more test frameworks, but requires special support in the hybrid module container:

A. The hybrid modular JAR *T.jar* contains everything in *M.jar*, in addition to test classes and resources, and an updated module declaration with additional dependencies (*requires*).
B. The tests are written to be run by a test framework. The additional test dependencies, and their transitive dependencies, does *not* have to be converted to hybrid modular JARs, but that transitive set of JARs cannot conflict with any of *M*'s transitive required dependencies. Compilation can be done with the JAR files as automatic modules.
C. The hybrid module *T* class loader must try to load classes from the test framework JARs, if a class is not found by the standard lookup mechanism (class is neither in *T* nor in exported packages in transitively required modules).
D. A test framework specific *booter* creates a hybrid module container, invokes a *main* stub, sets up the test framework, and passes control on to the test framework.

This will likely work with more frameworks because the classes and resources seen by the test framework includes all the test framework JARs, the unit tests, and the classes in the module being tested. In this respect, the class space is indistinguishable from running the tests with the normal class path (pre-JPMS).

# 3 Implementation

## 3.1 *javac* wrapper

A JHMS implementation *should* provide a *javac* wrapper to implement (§2.3) to help with verifying the versions used during compilation are consistent among compilations of the different modules.

## 3.2 *java* wrapper

A JHMS implementation *should* provide a *java* wrapper that can be used to launch a *hybrid module application* (§2.6), similar to how *java* can be used to launch a JPMS application.

The *java* wrapper *may*

1. launch in class path mode,
2. parse options and arguments,

3. make a hybrid module container (§2.7) with *main hybrid module* according to *--module* as the root hybrid module, and *module path* according to *--module-path*, and then
4. pass execution to the *main* method of the *main class*, the class specified as part of *--module*, or otherwise the main class must have been set on the main hybrid module (§2.5).

## 3.3 Class loading

Each hybrid module *M* is 1:1 with a class loader $L_M$ that loads a type *T* in package *P* by

1. delegating to another hybrid module class loader $L_N$, if *P* is exported by a readable hybrid module $N \neq M$ (§2.2), or
2. delegating to the platform class loader, if *P* is exported by a readable platform module (§2.2), or
3. defining *T* (*defineClass()*) by reading the class file from the hybrid modular JAR (§2.3).

The hybrid module class loader of a hybrid module class *C* is *C.getClassLoader()*.

As a side-effect, it is possible to gain access to an (otherwise) inaccessible type *T* of another hybrid module *N* (§2.2) by using *N*'s class loader and calling *loadClass()*.

## 3.4 Relation to JPMS modules

Each hybrid module *M* is 1:1 with an unnamed JPMS module, the unnamed JPMS module associated with the class loader, that according to JLS11 (§7.7.5)

1. *exports* and *opens* all packages, and
2. *reads* all observable modules, which for JHMS means all readable modules (§2.2).

The JPMS module associated with a hybrid module class *C* is the unnamed JPMS module associated with the hybrid module class loader:

3. *C.getModule() == C.getClassLoader() .getUnnamedModule()*

## 3.5 Module Resolution and Graph

During run time readability resolution, the module graph and its resolution are useful concepts.

The *module graph* is a directed graph of (platform and hybrid) modules (uniquely identified by name and version), and a result of the *resolution* of a set of *root modules R*.

Phase 1:
A module *M* at version *V* (*M@V*) is *resolved* as follows:

a. Add *M@V* to the module graph, unless it has already been added (skip the rest of steps).

b. If *M* is an automatic hybrid module, skip the rest of steps (see phase 2).
c. For each module *N* that *M requires*, resolve *N@W*, where *W* is the version module *N* had at the time *M@V* was compiled.
d. Add an edge from *M*@V to *N@W*.

Phase 2:
If the module graph contains *any* automatic hybrid modules, which by (1.b) was not completely resolved:

a. Resolve *all* observable platform modules according to phase 1.
b. Add the remaining observable automatic hybrid modules, and resolve all automatic hybrid modules according to phase 1, ignoring (1.b). From (§2.1.1), an automatic hybrid module requires *all* modules.

The set of readable modules for a module *M* (§2.2) can be seen as a subgraph of the module graph.

An implementation will typically resolve the module graph as a key component of figuring out readability.

## 3.6 Services

Since services are not supported, and *uses* and *provides* are ignored (§2.1), an implementation *may* warn if it encounters *uses* directives in the module declaration at compile time, and *should* warn at run time. It *may* fail at run time, but only if it is possible to disable that failure by simple means, e.g. a system property.

## 3.7 Unit testing

Black-box unit testing of a hybrid module *M* can be accomplished as follows:

1. A separate *test source directory T* contains the source files for the unit tests, in a directory tree mirroring the source directory for *M*.
2. A *test module declaration module-info.java* is a copy of *M*'s module declaration *module-info.java*, but in addition, contains *requires* for the *test dependencies*. InjtelliJ, for example, doesn't like to see the *module-info.java* in the root of the test source directory when it already has one in the source directory, so keep it somewhere else.
3. The test source directory and test module declaration is compiled using *–patch-module M=M.jar -d TDIR*.
4. The *test hybrid module M'* is a copy of *M.jar*, but updated with `*jar -u -f M'.jar -C TDIR .*` .
5. A unit test framework specific *boot hybrid module* is run in a hybrid module container, and getting the *test hybrid module M'* class loader e.g. through the context class loader. This boot hybrid module sets up and invokes the unit test framework driver.
6. The unit test framework driver JARs must have been converted to (and compatible with) hybrid modular JARs. The unit test framework driver uses the class loader to find test classes, instantiate them, and invoke test methods as normal.

The above has been verified to work with JUnit 5.9.1.

# 4 Relaxations

The other chapters of this specification describes an ideal and narrow interpretation. Such interpretations may not be practical nor secure: This chapter specifies relaxations of the various rules.

## 4.2 Version relaxation

It is allowed, but discouraged to substitute a module with a different version of the module, both during compilation and run time. This may be important to deliver security fixes.

## 4.2 Automatic hybrid modules

A JAR without a *module-info.class* is a *plain JAR* backing an *automatic hybrid module*.

This section (§4.2) only applies if the automatic hybrid modules relaxation is claimed by the JHMS implementation, *and* automatic hybrid modules has been *activated* (§4.2.2). If so, the rest of this section includes the various amendments to the specification (§2).

### 4.2.1 Auto read set, auto path

All automatic hybrid modules reads the same set of modules called the *auto read set*:

1. All platform modules,
2. All hybrid modules on the *auto path*, including the automatic hybrid modules.

The *auto path* is similar to the module path used to find JARs (§2.3): it is a list of paths to JARs and directories containing JARs. By default, the auto path equals the module path,[9] but it must be possible to set it explicitly, or set it as an exclusion list to be applied to the module path.

It is an error if there is more than one version of a module on the auto path.[10]

### 4.2.2 Activation

Automatic hybrid modules are *activated* if

1. The module graph contains at least one automatic hybrid module, or
2. the auto path has been set.

---

[9] The auto path defaults to the module path, and so if the module path contains multiple version JHMS would fail? No, because this section on automatic hybrid modules only comes into effect if the module graph contains any automatic hybrid modules, as mentioned earlier.

[10] Again, automatic hybrid modules (§4.2) only applies if there is at least one automatic hybrid module on the module path.

### 4.2.3 Module descriptor

Automatic hybrid modules has an implied module descriptor as follows:

1. All packages are exported
2. *requires* of all modules in the auto read set.

The name and version of an automatic hybrid module is derived from the JAR as follows (identical to that of JPMS[11]):

1. The name is based on the prefix of the filename, up to but not including the first "*-DIGITS.*", or if not found up to but not including the *.jar* file extension. The name is then normalized as follows:
    a. All non-alphanumeric characters are replaced by dots (.).
    b. All leading and trailing dots are removed, and sequential dots are collapsed to one.
2. An *Automatic-Module-Name* entry in the JAR manifest (*META-INF/MANIFEST.MF*) overrides the name.
3. The version is based on the suffix of the filename (excluding *.jar* file extension), starting with *DIGITS* in the first occurrence of *-DIGITS.* in the filename (including *.jar* file extension), or *null* (absent) if not found. If the version is unparseable by *ModuleDescriptor.Version*, the version is also *null*.

The *Main-Class* manifest entry will be honored and used if the automatic hybrid module is the main module in (§2.7) and (§2.8), but the main class has not been explicitly provided.

# 5 References


[JPMS] Java Platform Module System, aka JPMS, aka JSR 376: http://openjdk.java.net/projects/jigsaw/spec/
[OSGi] OSGi: https://www.osgi.org/
[SUPL] Java Hybrid Module System Supplemental, https://docs.google.com/document/d/1j9RliG973g4TtD33T1rMZfpPjveuc7nGK-bSaQ2l34w/edit
[MJAR] Modular JAR: https://docs.oracle.com/javase/9/tools/jar.htm
[JLS11] Java Language Specification, Java SE 11 Edition: https://docs.oracle.com/javase/specs/jls/se11/jls11.pdf
[READ] Module resolution and readability graph, *java.lang.module* package, https://docs.oracle.com/javase/9/docs/api/java/lang/module/package-summary.html
[MDVE] *ModuleDescriptor.Version()*, *java.lang.module* package, https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/module/ModuleDescriptor.Version.html


[11] See *deriveModuleDescriptor(ModulePath.java)* in OpenJDK 11 or *ModuleFinder.of(Path...)* Javadoc.