Jigsaw, EC Member Concerns

An analysis of the Jigsaw technology and its relationship to JPMS (JSR-376)

Rev. 6 2017-04-18

Table of Contents

Summary

Reinvention, Not Standardization

The Jigsaw implementation is a new module system which is has worked successfully for modularising Java itself, but is largely untried in wider production deployments of any real applications on top of the JVM. Many application deployment use cases which are widely implemented today are not possible under Jigsaw, or would require a significant re-architecture.

Reductive Design Principles

Jigsaw's key design points are predicated on a reductive[1] approach to forward compatibility, which works well for modularising Java itself, but becomes restrictive for the broader use cases that application deployments have. By enforcing the philosophies that make sense for modularization and encapsulation of the Java platform itself into the application domain, the specification actually reduces the ability for application developers to easily adapt to this particular implementation of a module system.

As a result of drawing requirements from the prototype implementation's primary behaviors, the set of use cases which are now considered acceptable have been limited to conform to implementation preference. We would prefer to have extracted the requirements and design from existing application deployment use cases. Many practices which were considered routine and useful in Java are now redefined as anti-patterns in Jigsaw, as described in the Technical Challenge Points section of this document (e.g. "Cyclic Dependencies", "Concealed package conflicts", "Reflection Behavioral Changes", "Module Naming Restrictions", "Adding packages is necessary", "Service Loading Changes" "Resources and Modules").

This results in a subtraction of capabilities for any code consuming Jigsaw. Conversely, we believe that JPMS should be conceived as a fixed set of *added* capabilities which allow for *new* use cases, without excluding existing use cases from being able to migrate to or take advantage of modularity.

A Disrupted Ecosystem

Jigsaw's implementation will eventually require millions of users and authors in the Java ecosystem to face major changes to their applications and libraries, especially if they deal with services, class loading, or reflection in any way. Most of these changes are derived from the implementation choices of Jigsaw and the requirements that were drawn from it.

[1] That is, the specification is based on the idea of subtracting capabilities and adding restrictions

The specification was written to promote certain best practices (e.g. modules are the ultimate authority for determining package access and dependency information, modules should be immutable with a complete eagerly resolvable dependency set, packages should never be duplicated, dependencies should never contain cycles, etc) . This works well for modularising Java itself but is a new, untested, and unproven architecture for deploying applications in a modular manner. In some cases the implementation of Jigsaw contradict years of modular application deployment best practices that are already commonly employed by the ecosystem as a whole.

Fragmentation of the Java community

Due to lack of one to one mapping of use cases (or sufficient interoperability capabilities) and other restrictions, we are concerned that there will likely be two worlds of Java software development: the Jigsaw world, and the “everything else” world (Java SE Classloaders, OSGi, JBoss Modules, Java EE, etc). A library developer will either need to pick which world to support, or deal with the burdens of a 'maintaining both' strategy.

Failure to Meet Major JSR Submission Goals

The JSR submission goals outline certain expectations that are integral to acceptance of the JPMS final release. Several of these goals are not met by the current Jigsaw implementation.

Approachable, yet scalable

The JSR submission specifically expresses that the implementation should support large-scale development. The submission states that:

> *"This JSR will define an approachable yet scalable module system for the Java Platform. It will be approachable, i.e., easy to learn and easy to use,* ***so that developers can use it to construct and maintain libraries and large applications for both the Java SE and Java EE Platforms****."*

For the purpose of evaluating this subjective goal, we define “easy to use” as:

- Having *equal or greater* robustness (tolerance of user input) to Java today
- Having *equal or lesser* effort required by the user to “construct and maintain libraries and large applications” in Java today

Constructing a Jigsaw application is definitively *less* robust than Java today as Jigsaw imposes a number of additional restrictions that will result in errors not previously encountered (see sections covering *concealed package conflicts*, *split packages*, *duplicate packages*, *multiple module versions*, *module naming restrictions*, *cyclic dependencies*, *JSR-250’s awkward place*, *service loader changes*, *reflection behavior changes*, etc).

Also, constructing a Jigsaw application definitively requires more effort by the user to “construct and maintain libraries and large applications”. A Jigsaw application/library must either define one or more additional module descriptors (`module-info.java`) that accurately define the semantics of the respective module, or utilize automatic modules, which involve the use of additional special rules that must be taken into account by the user.

Additionally, there is a an impedance mismatch with widely adopted practices for assembling software in the Java ecosystem, as expanded on in the Impedance Mismatch with Maven section.

The extra burden imposed, logically scales proportional to the size of an application, as does the probability of an error generating input / restriction violation Therefore, we believe that this JSR goal appears unmet, in particular for the target class of “large applications” (that will commonly involve the blending of multiple independent projects).

Leveraged by Java EE 9

It has been made clear since the beginning of the JSR process that it is expected to provide a basis upon which Java EE 9 can be built. As stated in the submission:

> *"This JSR targets Java SE 9.* ***We expect the module system to be leveraged by Java EE 9****, so we will make sure to take Java EE requirements into account."*

The limitations in Jigsaw almost certainly prevent the possibility of Java EE 9 from being based on Jigsaw, as to do so would require existing Java EE vendors to completely throw out compatibility, interoperability, and feature parity with past versions of the Java EE specification.

Concerns about Jigsaw as a complete solution

The patterns introduced within Jigsaw are (in some cases) going to be extremely difficult to fix even in a later release, and will create backwards- and forwards-compatibility problems that will be very difficult to unwind. The result will be a weakened Java ecosystem at a time when rapid change is occurring in the server space with increasing use of languages like Go.

These problems, which are outlined in detail this document, range from adoption issues, to changes to distribution models, to fragmentation of the ecosystem and more.

A Visual Comparison Between Modular Implementations

The following table serves as a high-level summary of some of the more significant capabilities which are not met by the Jigsaw approach relative to existing modular system approaches. The individual points are expanded upon in greater detail in the technical points section of this document.

	Jigsaw	Class-Loader	OSGi	Java EE (Spec)	ext/lib
Allows cycles between packages in different modules	✘	✔	✔	✔	✔
Isolated package namespaces	✘	✔	✔	✔	✘
Allows lazy loading	✘	✔	✔	✔	✔
Allows dynamic package addition	✘	✔	✔	✔	✘
Unrestricted naming	✘	✔	✘	✔	✔
Allows multiple versions of an artifact	✘	✔	✔	✔	✔
Allows split packages	✘	✔	✔	✔	✔
Module redefinition	✘	✔	✔	✔	✔
Allows textual descriptor	✘	✔	✔	✔	✔
Theoretically Possible to AOT-compile	✔	✔	✔	✔	✔

Refining Jigsaw and timelines

Many of the issues could be fixed in a short amount of time, (e.g. layer primitives, circularity, version restrictions, etc.). Others might require a bit more time to get right, but would lead to a much better overall platform and user experience. A small delay is worth the cost if the alternative is rushing a solution that doesn't cover all use cases. It might also be possible to add additional hooks that could be leveraged by third-party code to improve the experience.

Technical Contention Points

Cyclic Dependences

The implementation forbids dependency cycles among modules during compilation, link, and run time. Disallowing cycles during compilation is an accepted and historical behavior in Java, however disallowing cycles at run time is not, and will cause surprising problems for the user at deployment time. Such cycles might even reflect engineering choices that are required to fulfill certain use cases.

Current specification

The Public Review specification has the following to say on the matter:

> *"It is a compile-time error if the declaration of a module expresses a dependence on itself, either directly or indirectly."* - proposed JLS § 7.7.1
>
> "*When all modules have been resolved then the resulting dependency graph is checked to ensure that it does not contain cycles. A readability graph is constructed, and in conjunction with the module exports and service use, checked for consistency.*" - proposed JDK specification for class `java.lang.module.Configuration`

The proposed JVM specification does not specify that module cycles are forbidden during class resolution or initialization.

Run-time problems

When modules are built, they are compiled against a set of classes which form the Application Binary Interfaces (ABIs) that the module requires in order to function. But it would often be the case that the final module is then included in a different environment entirely - either in a container, or else as a result of reuse. Nontrivial module environments can easily contain "long cycles" where a number of innocuous dependency relationships exist, but happen to form a cycle when certain combinations of modules are assembled.

Bypass the resolver

JPMS authors recommend that runtime support for circularity be added by container providers such as OSGi, JBoss Modules, or Java EE containers by bypassing the Jigsaw resolver completely, and using a custom class loader implementation to resolve class linkage questions.

This solution is completely functional for containers, but it is not functional for stand alone modular applications. In addition, containers will suffer from the deficiency that any software which inspects such a module's dependencies using the `java.lang.reflect` API, including a module inspecting its own, will see only a subset of them (typically, an empty set).

Compromise proposal

A compromise proposal is to continue to forbid cyclic dependences at compilation time (as this behavior is consistent with current practice and `javac` behavior), but to relax restrictions at link and run time so that assemblies of modules will not fail unexpectedly when the dependency graph changes between the build environment and the production environment.

This proposal has not yet been addressed.

Quotes:

> *"The JPMS resolver does not allow cycles amongst modules; this has long been the case. (Circularity amongst classes is allowed, as it must be.) If you want to allow cycles amongst your own modules then you can resolve them yourself and add whatever cycle-inducing readability edges you need."* - Spec Lead, in this post (2016)

Ideology

This is at its heart an ideological disagreement. It has been posited that the presence of circular dependencies is an anti-pattern and a design error: http://openjdk.java.net/projects/jigsaw/spec/issues/#CyclicDependences.

The primary supporting argument is that all modules which form a cycle are logically one module. However this at best applies only to limited cycles of a small fixed number of modules which come from the same author and are produced at the same time. Applications, even relatively small ones, now consist of dozens or hundreds of distinct pieces from a multitude of sources. Maven has been a big enabler of this: By allowing application dependencies to be managed automatically the friction of doing so has been greatly reduced, and it has been observed that including substantial dependency graphs in an application as a common practice has increased in pace.

Automatic Modules

Automatic modules are purported as a compatibility mechanism allowing JARs to naturally grow into modules in a modular environment.

The idea is that a module would be automatically generated from a JAR which has a name that is derived from the name of that JAR. The name would undergo various transformations to make it align with the proposed naming convention of modules.

The proposed behavior suffer from various undesirable side-effects. Many participants in the discussion seem to agree that automatic modules bring more harm than good.

Quotes:

> *"... automatic modules in general are not a good solution to the problem space in general"* - Stephen Colebourne, JSR 310 spec lead in this post
>
> *"I regard automatic modules as one of the most dangerous and poorly specified areas of the current spec, and will be taking this up with the other members of the EG."* - Neil Bartlett, current JSR 376 EG member in this post

New Tooling Required

Users will be relying on build tooling like Maven to create their modules and their distribution environments. Already today there is at least one tool (https://github.com/moditect/moditect) which can modularize an archive, and it is expected that more will appear.

If the other issues listed in this document can be resolved, modularizing a JAR could be as simple as choosing a name and reviewing the results of the calculation of existing modules and Maven dependency metadata. Even manually specifying dependencies could be a fast and easy way to modularize an existing artifact.

Tooling Prevented from helping bridge the module name chasm

Recently the `Module-Name` metadata field was removed from the proposal. This field would have allowed a developer to express their intended module name separately from fully modularizing their own code. This would allow someone to avoid their otherwise legacy module from being subjected to the default automodule name algorithm which only uses elements of the filename as the module name. Not having Module-Name available creates an inherently unstable automatic module naming solution, and will likely cause conflicts between otherwise properly namespaced modules.

For the reasons of name instability, the current guidance (for adopters of Jigsaw) is to block or discourage publishing of libraries that depend upon automodules. This leads to the problem where no library creator can ever fully modularize until **all** of his/her dependencies have also done so!

With an ecosystem that has transitive dependencies (sometimes dozens to hundreds of layers deep) and with some of those very deep dependencies quite stable and therefore infrequently updated, this will likely mean that some components will never be able to be fully modularised.

The vision behind the `Module-Name` metadata was simply that it would make it easy for module authors starting with Jigsaw to immediately to choose their module name. It could make choosing and declaring a name easy, (maybe even required) very quickly for library authors. That means that the ecosystem could start to build up the metadata that is missing. The JSR could still do so (starting now), and as Jigsaw starts to hit critical mass, there would hopefully be very few important libraries that aren't properly named by their authors as intended.

The bar to picking a good name is clearly much lower than fully modularizing, especially if there is social pressure into doing so before all your dependencies have.

If developers could start declaring their `Module-Name` early, and the rule against automodule dependencies is redefined such that it's OK to lean on something with a `Module-Name`, it becomes much easier and quicker for the ecosystem to get to a building point for full Jigsaw modularization.

Without the `Module-Name` metadata or some equivalent, build systems are effectively barred from helping with the conversion to achieve the very goal of this entire process.

Inconsistent behavior

Automatic modules have special behaviors that are not shared by the classpath or by modules, including allowing cycles, having access to all modules, and being unable to restrict visibility or accessibility in the way that named modules can. Thus as a migration tool, it is problematic to rely upon them.

Automatic module naming follows new patterns and relies on JAR naming conventions, with no option to customize the automatic module's name unless the JAR is renamed during assembly.

Inadequate Isolation

An expectation of a module system is that a module’s implementation choices are independent of other modules in the system. Java EE, OSGI, and other plugin systems incorporate isolation systems characterized by such concepts as fully isolated package namespaces and separated module classloaders. Another example is Dynamic libraries (e.g. DLLs, SOs) which support isolation of symbols. A module system without adequate isolation will be unlikely to cope with

an ecosystem which consists of modules produced by many different authors with different design parameters.

Multiple module versions

The JPMS Spec lead specifically chose not to solve multiple version resolution situations..

> *#MultipleModuleVersions* — Allow multiple distinct modules of a given name to be loaded in a convenient fashion, without using reflection. This could be done by creating new layers automatically, or by relaxing the constraints on multiple versions within a layer, or by some other means (*cf.* #StaticLayerConfiguration, #AvoidConcealedPackageConflicts). Addressing this issue may entail reconsidering the multiple versions non-requirement. [Mike Hearn]
>
> > **Resolution** These overlapping issues do reflect actual, practical problems. There are, however, already effective -- if somewhat crude -- solutions to these problems via techniques such as shading (in Maven) and shadowing (Gradle). More sophisticated solutions could be designed and implemented in a future release.
> >
> > The lack of immediate solutions to these problems should not block a developer who wants to modularize an existing class-path application. If such an application works properly on the class path today then it likely does not have conflicting packages anyway, since conflicting packages on the class path almost always lead to trouble.
> >
> > [Spec Lead]

This decision appears to be the result of the implementation choice of the Jigsaw authors to attempt to use a single class loader for all JDK modules, and then reuse that approach for application modules on the module path (a problem which is addressed elsewhere in this document).

One critical specification problem is that there is no clear definition of what constitutes "multiple versions" of a module. Jigsaw uses the following interpretation:

- Two modules with the same package names in them are considered to be different versions of the same module.
- Two modules with the same name are considered to be two versions of the same module.

The problem with both of these rules is that there are cases where the two modules in question are not different versions of the same module. Examples include usage of generic common names as an identifier (“util”, “beans”, “logger”,”client”, etc), and competing distributions / variations of a standard (e.g. JSR) or common API. Therefore, the aforementioned restriction

not allowing any situation that could be interpreted as multiple versions causes a serious problem for these cases.

Concealed package conflicts

When two modules have the same package name in them, but the package is private in both modules, the module system cannot load both modules into the same layer. This situation is known as a "concealed" package conflict, because although there is no user-visible reason for a conflict, it exists nonetheless due to inadequate module isolation. This also implies that any future tool (Maven plugin, etc) that seeks to assemble a coherent set of modules for the modulepath **cannot** rely only on the published metadata of the modules. It must introspect within each and every module to the package level to determine whether any conflicts exist.

Handling this situation is a primary characteristic of existing module and plugin systems, including the built-in Java SE ClassLoaders While Jigsaw can support a ClassLoader per module configuration, doing so requires a user to develop a custom bootstrap process. The standard JVM launch (using *-p*) will fail immediately if any module contains the same package, even if it is not exported.

Non-concealed, non-conflicting duplicate package names

A similar case is where two modules have exported public packages of the same name. A scenario where this can occur is when dependencies require two ABI incompatible versions that share the same name. For example, one library might use methods in Guava 18 that were dropped in Guava 20, and another library might use methods in Guava 20 that do not exist in Guava 18. As with the concealed case, existing module systems are able to handle this use case, but it will fail on a standard JVM launch under the current Jigsaw implementation.

Split packages

"Split packages" is historically a controversial topic in Java. This case arises when there are non-concealed and non-conflicting duplicate package names, and there is a module which consumes both of the duplicated packages. In this case, some classes may bind to classes in one package, and some may bind to the other, or they may only bind to one or the other.

This is indisputably an advanced use case, and there are many approaches to handling this at an application level. However we believe that at a specification level, there is no technical reason to restrict this situation on a basis more strict than opt-in.

A simple solution: class loaders

Most of the issues described in this section derive from the design decision to force all modules from the module path into a single class loader, and to a lesser extent, the design decision to force platform and application modules into a single layer.

The module API provides methods to construct layers which map each module into its own class loader. However this mechanism is not used by the JDK for applications, even though it is able to solve all of the issues in this section. The primary argument for this situation revolves around a predicted compatibility issue, that applications, once converted to Jigsaw, may be surprised that `getClassLoader()` returns a different value respective to the jar file that contains it. However, there are other more severe (and more common) compatibility breakages introduced by Jigsaw in the same situation that expose the weakness of this argument:

> High-level JEP that describes CL topology restructuring
>
> Example breakage caused in Gradle
>
> Example breakage caused in Eclipse
>
> Example breakage caused in CDI

No "Current Module"

Existing systems rely on identifying the current application by using the Thread Context Class Loader (TCCL). Because modules in Jigsaw are not represented by class loaders, programs which rely on this behavior of the TCCL will begin to exhibit subtly incorrect behavior.

No corresponding concept exists for modules, which means that any software relying on this concept must be redesigned and re-implemented to use some different approach.

Lack of Mutability

A desired characteristic of existing modular runtimes is that modules can be dynamically installed and redefined (often referred to as hot and/or incremental deployment). Jigsaw does not support this directly (http://openjdk.java.net/projects/jigsaw/spec/issues/#MutableConfigurations), but introduces a hierarchical grouping called Layers. We believe the hierarchical nature of this solution is a poor fit for supporting updates to modules, which other module systems have commonly found to be nodes with peer-to-peer relationships that form a graph.

While Layers were enhanced to support multiple parents, the solution cannot be used to model a graph (since layers cannot have cycles). Non-trivial usage of the Layers capability does not

scale, as it creates very large search paths instead of a desired O(1) resolution. Several applications will be forced to bypass and / or reimplement Jigsaw's class loading and resolution in order to get their desired behaviour.

Hierarchical layers don't meet application needs

Existing Module systems have all been built on the experience that hierarchical linkage systems (such as the traditional classloader relationship) don't meet modern application deployment needs. Hierarchical implementations introduce locking problems, visibility issues, complex resolution for parent / child-first dilemmas and other issues have demonstrated the weakness of that approach.

Hierarchical Layer relationships in Jigsaw also suffer from similar problems, e.g., a linear scan of all parents for all modules.

Modules always loaded eagerly

In order to support the restrictions imposed by Jigsaw, modules are always loaded and resolved eagerly within a layer - even if there are hundreds or thousands of modules on the module path. This is opposite with the classical behavior of classes on the classpath, which are always loaded, resolved, and initialized on an as-needed basis. We believe that the as-needed loading mechanism is a useful model. Existing, module frameworks such as JBoss Modules and OSGi resolve lazily.

In Jigsaw, platform modules must be divided into two groups: the eagerly resolved platform modules, and a set of optional modules that are only loaded if explicitly specified on the command line. This can be awkward, particularly if a module's requirement is only discovered late in execution.

In Jigsaw the JVM module path cannot have modules added to it at runtime. This is more restrictive than the existing model with classes in a classloader, i.e., a package can always have more classes dynamically added to it, which has been repeatedly proven to be a very useful mechanism for library and framework authors..

Layer Primitives

There are proposed primitives for Jigsaw (from EG members) that add the ability to dynamically modify a module in a few specific ways, is necessary and useful to developers and users of existing containers and plugin systems.

Primitives already exist, modules are already dynamic

All of the proposed primitives already exist within Jigsaw. Modules themselves have the ability to do things like add exports which the layer controller cannot do without injecting bytecode into the module to call these methods.

Many frameworks generate proxies and other bytecode with security needs that would entail using new private packages, but the spec lead and other EG members disagree with these use cases (see A small change: a dozen lines).

Adding packages is necessary

Many frameworks, containers, tools, and libraries (including the JDK itself) make use of dynamic code generation to implement various types of functionality. These frameworks have the same need as the JDK to generate classes in non-public packages and may not be able to function if unable to do so.

Containers and plugin systems also often adhere to the practice of lazy discovery of classes. In these cases, in order to properly interoperate with Jigsaw, such frameworks must be able to dynamically add packages and other module characteristics as they are discovered.

A small change: a dozen lines

The code to make this change is a very small patch that exposes a small number of methods already present in the implementation. This was proposed in: (http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2016-December/000501.html and http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2016-December/000507.html) and ultimately rejected without a technical justification:

> *"I have too often seen APIs that seemed like a good idea at the time but were, in fact, woefully deficient, baked into the Java Platform where they fester for ages, cause pain to all who use it, and torment those who maintain it. I will not let that happen here"* - Spec Lead *in* JPMS posting rejecting the change

Module Naming Restrictions

Module Names

Since the initiation of Jigsaw into JPMS, module names have been restricted by the rule that they must be, (or approximate), valid Java language names. This rule excludes a large number of artifacts in existing module systems and in Maven.

Many artifacts within Maven contain hyphen ("-") characters, which are not allowed by the module naming rules. Also, the colon (":") delimiter (used to separate artifact IDs from group IDs in Maven) is also disallowed.

Containers have the ability to bypass these naming restrictions, but in order to do so, they must generate bytecode for their module descriptors (as the descriptor building API enforces the `javac` naming rules).

Module Version Strings

Module version strings in Jigsaw are constrained by a format which does not reflect any current versioning practices. They are therefore incompatible with most existing Java-based versioning schemes.

The implementation of version strings in Jigsaw involves several ists of Objects and extensive usage of boxed types.

We believe that a module system should support versioning schemes that reflect a users' or containers' best practices in common use, while also making recommendations for those cases where a practice is not established. We believe each module loading layer should be able to establish its own policy for syntax, semantics, and ordering which operate solely within the realm of that layer and do not interfere with that of other layers.

Module Descriptors Are Bytecode

The Jigsaw implementation mandates that module descriptors should be established and loaded in bytecode format.

Architectural concern

Binary descriptor formats are an uncommon implementation choice. Text-based descriptor formats (particularly those based on common meta formats like properties, MANIFEST.MF, or XML) are easier to read, modify, and programmatically manipulate using standard tools. The

cost of parsing such files is minimal and in some cases nearly indistinguishable from their binary counterparts.

Modules should not be a part of the Java language specification

It has been suggested on multiple occasions that module descriptors do not make sense as bytecode for a variety of reasons (http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2015-December/000212.html, http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2015-September/000125.html). However, these arguments were met with the assertion that modules are "fundamentally, a new kind of Java program component" and that it therefore has to be "specified in both the Java Language Specification (JLS) and the Java Virtual Machine Specification (JVMS)." This assertion was subsequently contested in a post to the JPMS spec experts list in 2015.

The argument that Jigsaw behaviors must be part of the JLS is used to justify the storage format and compilation behavior of Jigsaw descriptors. However, the argument that application modularity on top of the JVM must be part of the JLS has not been strongly supported by technical arguments. A number of successful application module, plugin, and class loading systems exist without the necessity of elevating modules to a programming language level.

Even if the enhanced security and diagnostic features that the JVM provides are brought into consideration, there is no new behavior which has been shown to be required or otherwise made possible by the current implementation or JLS modification as these are all run-time behaviors and enhancements.

Service Loading Changes

Behavioral changes

The contract of ServiceLoader was established over 10 years ago in Java 6 and is now considered a standard way to locate providers for interfaces. Jigsaw changes the behavior of this API in substantial and compatibility-affecting ways, e.g. http://download.java.net/java/jigsaw/docs/api/java/util/ServiceLoader.html and http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2016-December/000524.html. The changes are discussed in the following sections.

No relative services

With traditional ServiceLoader, the services you load would be based on what the current class's class loader, or the specified target class loader, could locate. This allows services to be intuitively defined on a relative basis in class loader-oriented systems.

This behavior is removed for modules under Jigsaw. A different, module-based service locator is used by default which does **not** have relative behavior.

Ordering is lost

In Java 6 through 8, ServiceLoader reported services in the order they are discovered by the class loader, meaning the class loader could implement a reasonable and predictable policy for returning implementations.

Jigsaw does not specify the order that services are returned within a layer, which will cause unforeseen stability problems as applications may be relying on a preferred load order.

Global namespace

In Jigsaw, all service interfaces and implementations on the module path are flattened into a single, global namespace.

This means that it is impossible to selectively assign service implementations, or to get a predictable result if the same interface exists in more than one location in the module graph, among other problems.

No extensibility / customizability

In Jigsaw, there is no API by which the behavior of service loading can be customized or modified back to its original behavior, unless Jigsaw modules are not used at all. That is, the special behavior and privilege of service loaders cannot be replicated by user code. Even when a customized layer is used, the layer provider must provide a fixed mapping of available services up front.

Strong restrictions

Every module that uses a service must also declare that the service is being used in the module descriptor. Most existing service wiring frameworks are moving away from multiple-site declarations, as this has been found to cause issues.

Failure to declare a service that is being used results in a run-time error, which can be surprising, and also prevents any sort of dynamicity in terms of finding an implementation.

Java 9-aware software can dynamically add a `uses` declaration to their own module before loading a service, but adding this at scale will be a difficult task.

These service loader changes were introduced as a balm against the rules regarding circularity. However the changes cause new problems. Sticking with the relative behavior would allow modules to choose their services and their implementations in the same established way that

they always have, using dependency edges to create a predictable set and ordering for services.

We believe that the addition of a global or layer-wide service registry (as a new, supplementary feature) would be an example of a useful (and fully compatible) change that solves the same sorts of configuration problems.

Reflection Behavioral Changes

Jigsaw introduces new restrictions on private reflection which entail disallowing the `setAccessible()` method of reflection entities from being invoked from modules which are not specifically granted access to the module in which the corresponding member exists. However this restriction is not consistently applied, i.e legacy classpath-based code, as well as the *unnamed* module, both are exempt from these restrictions.

Security justification

The security justification is clear: less reflective access means fewer CVEs. However, the security justification must be weighed against impact of the new restrictions on compatibility and usability. Increasing security is of less use if existing or new software cannot take advantage of the new capabilities.

Compatibility impact

New module access constructs have been introduced, which allow modules to opt in to allowing inter-module reflection. However, this mechanism has compatibility concerns.

Each existing artifact that is being modularized must consider the reflection accesses made by that artifact and decide what consumers must do in terms of opening access. Because it is the module that is being reflected upon that must grant access, it is not until run time that reflection access problems can be detected (because there is no way for a module to declare that its users must open themselves for private reflective access), or to test for that at build or load time. Examples of how users must deal with runtime errors rather than compile time errors in both JavaFX and GSON have been posted to the JPMS comments list.

Quotes:

> *"I have argued that the Java security model should be brought up to date, but I understand that requires a far reaching redesign that is beyond the remit of the modularity EG. That means that modularity should 'do no harm', while avoiding those land mines you refer to below."* - Tim Ellison (IBM) in this post to the JPMS spec experts list in 2015

Specification and framework impact

It is unclear how other specifications (such as CDI or JPA) are to be granted access to the modules which consume them, especially in an embedded or containerless context (such as one might find in a cloud-style deployment). The consuming module must somehow grant open access to the specification implementation, but the groups responsible for moving such specifications to their Java 9-ready forms are unlikely to be willing to require users to establish dependencies on modules other than the specification API module. This implies that the specification API itself must be tailored to the implementation, or in some other general way be able to relay privileged access to implementations.

Problems with "the big kill switch"

Because of the scale of the compatibility problems, the JDK has lately added an option to blanket-disable the additional reflection security capabilities. The change itself introduces a new problem: log messages are emitted to the error stream regardless of any application use of that stream.
Quotes:

> *"The big kill switch doesn't seem useful, it just hides everything that needs work."* - Keimpe Bronkhosrt (Oracle), in this post to jigsaw-dev

JSR-250 Challenges

In order to remain relevant in the modular world, a module implementing a specification will need to be consumable in a predictable manner by applications; in particular, each specification will require a predictable name and clear requirements for consumption. In the case of JSR-250 (the `javax.annotation` package), the Java SE platform has included the classes for quite some time. However the classes included in the platform have lagged behind the specification in the past, and there is a general desire to move them out of the platform for this reason and for the reason that they do not necessarily belong in the platform. The current Jigsaw proposal seeks to do so but has challenges.

The current proposal to rename the bundled JSR-250 module from `java.annotation` to `javax.ws.annotation`, citing the history of JAX-WS within the platform. The user must then manually enable that module, along with any other JAX-WS support modules, to use the container-bundled JAX-WS implementation. The module will also be deprecated, encouraging use of an external version.

However this poses a challenge: How does a module distribution employ an updated version of this module at its defined name? One option is to ignore the provided module and bundle a new `java.annotation` module. However, this option causes a problem when the built-in JAX-WS support is in use, as the packages in the new module will conflict with the module in the JDK.

To get around this, a developer will need to upgrade the `javax.ws.annotation` module *and* establish a pseudo-module that aliases `javax.ws.annotation` to `java.annotation`.

This is awkward for the developer. Ideally, the specification classes should be included under their specification name, and made upgradeable. They can then be deprecated from the platform if necessary.

Resources and Modules

Resources are used for a variety of purposes, from data supplementation to configuration to service description. Historically, a class could use the Thread Context ClassLoader or its own class loader (depending on circumstance) to locate such resources, and this model works consistently.

In a multiple-module system (whether or not the module is backed with a dedicated class loader), it is useful to be able to find resources from other modules, and to know which module each found resource originated in, encapsulated in a single object which provides access to the content as well as the size and origin of the resource. This idea was proposed for the JDK in a 2009 bug report but has found little traction.

Under Jigsaw, inter-module resources has no modular support for this function (despite a number of new module-aware and classloader-incompatible resource APIs were added).

The service support which previously used the general resource support could leverage such a mechanism for added flexibility in a modular setting. At the moment only a one-off function that uses module implementation details is provided.

Distribution Model

An important aspect of a module system is how it manages independently developed, versioned, and packaged units of software. Two common approaches to this problem are overridable descriptors and flexible resolution systems.

Overridable Descriptor Approach

The overridable descriptor approach allows for a module to redefine the module specification of its full tree of dependencies. This allows for modules to potentially publish their details on a best effort basis, with a built in mechanism for consumers to adapt to conflicts as necessary. The ability to adapt allows for organic evolution of the system without requiring any form of coordination between participants. Examples of this approach include Maven and JBoss Modules.

Flexible Resolution System Approach

Another approach relies on a flexible resolution system that analyzes detailed data about the modules in the system (dependencies, available packages, etc), and produces a solution accounting for the requirements of each module and the variations available (e.g. versions). This approach also has the ability to adapt to independent life-cycles. OSGi utilizes this approach.

Jigsaw - A new approach with challenges

Jigsaw, on the other hand takes a different approach. It does not have a flexible resolution system, nor adequate metadata to generate solutions for independent software composition. It also explicitly seeks to avoid an override ability, as the security benefits would be defeated (a module could override another module's access restrictions). This would break Jigsaw's design principle labeled "fidelity", where the intended flow from compile to assembly to test to distribution to run has a dependency graph that is universally consistent.

Satisfying such aims is difficult to achieve in environments other than those composed of software with a collectively coordinated life-cycle, such as the JVM itself. Carrying this outside of isolated islands of software would require a centralized and specially curated repository of some form. This was expounded upon in great detail on the JPMS experts list in 2015, but was not resolved.

Decentralized artifact repositories versus centralized module registry

It was originally implied that Maven would eventually evolve into a centralized repository for Jigsaw modules:

> *"We're not trying to establish a new ecosystem of component distribution; we are, rather, trying to fit into existing ones, and in particular the existing Maven-based ecosystem."* - Spec Lead in this post to jpms-spec-experts

The single, global module namespace, cannot be met by Maven Central without a fundamental and complex change to the way that submissions are curated.

The reason for this is that today, a Maven artifact in Maven Central only has to resolve consistently relative to the set of artifacts it consumes, and (to a lesser extent because there's

some flexibility here) the set of artifacts it is likely to coexist with. This flexibility and relativity goes most of the way to mitigate the fact that many Maven artifacts have conflicting packages and version requirements.

In the Jigsaw modular world a set of artifacts must resolve in a mutually consistent way, yet are 100% non-conflicting in terms of module specification. They also have to be 100% mutually consistent in terms of dependency mesh. In order to have any sort of guarantee of consistency for any given module artifact, consistency must be guaranteed for **all** artifacts.

The Maven Central model for artifacts fails in this regard for the exact same reason that there isn't, for example, one unified Linux package "mega-repository". Packaging issues aside, there are many competing implementations of the same specifications and solutions to the same problems; these things have rippling effects on compatibility. In order to create one, single, unified module repository for *everything* in Maven Central that is internally consistent would be a extremely large undertaking for the Java ecosystem as well as requiring major maintenance.

It was eventually acknowledged that Maven can't meet this need(http://mail.openjdk.java.net/pipermail/jpms-spec-experts/2017-April/000667.html), yet the design and implementation constraints (described above under Distribution Model), which lead to a universal repository, still remain.

Impedance Mismatch with Maven

Since using Maven as a universal repository is not possibleusible, supporting Jigsaw requires Maven to port it's override approach, in a way that works around the constraints Jigsaw imposes.

As mentioned above, Java libraries and applications are commonly composed of multiple different projects produced independently by multiple different parties. This can be readily observed by inspecting *pom.xml* files in the maven central repository. A common problem encountered in assembling software produced by different parties with different lifecycles is a transitive dependency conflict. Maven provides multiple mechanisms to resolve these conflicts such as: excluding deps, overriding versions, utilizing Bill Of Materials (BOMs) etc. Additionally, the nature of the current Java classpath is such that even in the presence of a conflict (duplicate version, duplicate package etc), these cases may execute fine (although cause unexpected behaviour).

This conflicts with Jigsaw's design principles of "strong encapsulation" and "fidelity" where the descriptors of all artifacts are non-overridable and generated at compile-time in an augmentation unfriendly format (bytecode).

In order for Maven to support the ability of a dependent to override a dependency in a complete and comprehensive manner, Maven would have to implement a post-build time

module-info.class augmentation facility that remains consistent with already established mechanisms, and is capable of rewriting a full dependency tree. It's not clear that such a facility will be available as it likely requires further research and potential implementation. In the meantime, Java developers will have to resolve conflicting *module-info.class* files themselves, editing the bytecode of and repackaging dependencies on their own as necessary.

Example Scenarios

The following examples are non-exhaustive, and simplified to be illustrative of the types of conflicts developers would encounter

Duplicate spec dependencies

- foo-lib requires apache-JSR-XXX-api (needs jsrxxx package)
- bar-lib requires official-JSR-XXX-api (needs jsrxxx package)
- app requires foo-lib and bar-lib

With Maven and classpath the solution to the problem is to exclude one of the jsrxxx variants. However with Jigsaw you will get a compile (and runtime failure) until you open and edit foo-lib or bar-lib, and edit the descriptor. Alternatively you can create a Maven submodule artifact that builds a false alias, where it pretends to be one of the JSR API modules and “requires public” the other.

Dropped exported transitive

- foo-lib requires transitive guava
- bar-lib requires foo-lib (but not guava because it gets it for free with foo-lib dep, and it just works)

Some time later after a release, a user asks foo-lib’s maintainers to stop exporting guava because it's not necessary, and that conflicts in some other way for their use case, foo-lib agrees they got this wrong and removes the transitive keyword.

Users of bar-lib now will get IllegalAccessError, because bar-lb no longer has access to guava’s packages. To fix this, users will have to either downgrade foo-lib (if it's even possible), or crack open and edit either foo-lib or bar-lib.

Over-eager shading

1. foo-lib exports an API that exposes commons-collections classes, but it doesn’t yet support Jigsaw, so it shades them classes and re-exports them (can’t rename the packages since the types are in the API signature used by users)
2. commons-collection later decides to publish for jigsaw
3. other libs use commons-collections, which then conflicts with anything that also uses foo-lib

Qualified Open

1. foo-lib opens foo.beans to bar-lib (only bar-lib has access, works with 1.1)
2. myapp uses foo-lib from maven
3. zeta-lib uses new bar-lib 1.2, which has new methods it needs
4. bar-lib 1.2 recently refactored and has moved its bean introspection code to a bar-lib-impl module
5. myapp wants zeta-lib, but the upgrade of bar-lib breaks foo-lib
6. myapp must now refactor foo-lib

One could argue bar-lib's refactor is a mistake, and that they should have put reflection access in bar-lib and delegated back from bar-lib-impl. They could in turn argue that they shouldn't have to structure their system around reflective access, and foo-lib shouldn't have qualified the open. Foo-lib could argue that qualification looked like a good practice.

Regardless of whether or not this is an error, and who is at fault, the software is already released, and the disruption can't be undone until everyone upgrades.

Inadequate Compatibility Strategy

Many existing containers, frameworks, and applications are currently incompatible with Jigsaw. The recommended compatibility strategy is to run in a *legacy* mode where the class path is used, and ultimately either rewrite (for Jigsaw) or abandon all the incompatible artifacts. In order to achieve the goals of the JSR, we do not believe this strict position is necessary.

Quote:

> *"Ok. [Forget] Jigsaw compatibility. If doing so requires use of Java 9 tools, I'll have zero [...] level interesting in support for next 2+ years"* - Tatu Saloranta (Author of Jackson, WoodStox, ClassMate, TStore) Mar 16

Unusual API Constructs

Read edges (addReads)

Read edges are a new construct in Jigsaw. They represent the consumer side of the exports/requires relationship; in effect, this is a sort of access permission. However, the permission is not granted by the module being examined; rather, it is granted by the examiner. This access-control feature does not solve any security use cases because a module can always grant itself permission to read anything.

This concept is the source of a major compatibility problem. Any code using reflection requires read access in order to function correctly. Thus a decision was made to automatically add read access any time reflection is used on a member in a module other than the source module. This further adds to the question of the function of this mechanism.

It is unclear what user-visible problem is solved by this mechanism.

Unnamed Module(s)

The *unnamed* module of a ClassLoader is an architectural artifact that results from the mismatch of mapping classes between modules and class loaders. Essentially, any class that isn't explicitly loaded into a module is placed in the *unnamed* module of the module's classloader (note that this implies that there are many *unnamed* modules).

Unnamed modules have unique behavior compared to named modules. They cannot opt in to reflection restrictions, and report no name or version on stack traces.

Conflation of paths as packages

The design of Jigsaw fully isolates and hides resources between modules. This was done so that all linkage decisions between modules could be done solely on a package basis. However, once inter-module resources was introduced, this conflation has become awkward, resulting in rules such as: *"[...] The effective package name of a resource named by the string `"/foo/bar/baz"`, e.g., is 'foo.bar' [...] If a resource's effective package name is not a valid Java language package name (e.g., "META-INF.foo.bar") then the resource can be located by code in any module."*

We believe that modules should be able to control resource access on a similar basis to controlling package access, regardless of what path they are found in. Representing a path name as an invalid package name is an awkward construct.

Module descriptors cannot be easily constructed

The only way to define new modules in software is by creating a module info descriptor. There is a programmatic API for doing so which (by design) only allows a subset of possible module descriptor information to be specified. This is a deliberate choice so that users do not exploit capabilities that deviate from the narrow set of approved use cases.

In order to create descriptors which utilize the full range of Jigsaw capabilities, bytecodes must be generated and fed into the binary descriptor parser. Generally speaking, special support libraries are required to do this. This design deliberately creates difficulties for dynamic programs, frameworks, and containers.

Secondary API for loading classes and resources

Since the beginning of Java, locating resources and class content has been possible in two ways: by class and by class loader. In order to transition to Jigsaw, classes which load resources from specific peers must be rewritten to use the methods on Module instead. This means that they will require distinct implementations for Java 8 versus Java 9 in order to achieve the same behavior.

"Optional" is everywhere

Most of the API changes in Jigsaw use `java.util.Optional` for getter return values as a substitute for null-checking. Opinions of this feature vary widely, and general usage of it in this sort of context remains controversial. Several changes to this class which are intended to

address outstanding issues have been introduced in Java 9, and are not yet considered battle tested best practices.

Quotes:

> *"Of course, people will do what they want. But we did have a clear intention when adding this feature, and it was not to be a general purpose Maybe or Some type, as much as many people would have liked us to do so. Our intention was to provide a limited mechanism for library method return types where there needed to be a clear way to represent "no result", and using null for such was overwhelmingly likely to cause errors.*
>
> *"For example, you probably should never use it for something that returns an array of results, or a list of results; instead return an empty array or list. You should almost never use it as a field of something or a method parameter.*
>
> *"I think routinely using it as a return value for getters would definitely be over-use."* - Brian Goetz in this post to StackOverflow

References:

> Java 8 Optional: What's the Point?
>
> Why java.util.Optional is broken
>
> How Optional Breaks the Monad Laws and Why It Matters
>
> What's Wrong in Java 8, Part IV: Monads

Primary Use Case Considerations and Strategies

Java Developer Strategies for Using Jigsaw In its current form

1. Avoid using Jigsaw, and instead advise Jigsaw users that wish to consume your project to create their own local module to represent it.
2. If you wish to support Jigsaw, and you wish your project to be usable by non-Jigsaw user's (Class-Path, Java 8, Java EE, OSGi, Eclipse, etc), then you can produce a special Jigsaw-only build along with a traditional build of your library.
3. Avoid relying exclusively on the package exclusion security capabilities of Jigsaw, as they can be disabled via the command line, and the traditional build mentioned in step 2 won't utilize them.
4. Reduce the number of Jigsaw dependencies in your project to the smallest possible number, since it's unclear when or if the dependency and package conflict issues could be worked around by build systems such as Maven. Consider other strategies such as:
    a. Define your own modules locally to represent a dependency;
    b. Use the shade plugin to relocate packages and merge the dependency into your module (this avoids the duplicate concealed package issue, as well as version conflicts);
    c. Directly include the source of the dependency in your project.
5. Be prepared to patch and convert dependencies which encounter compatibility issues with Jigsaw.
6. Since there is no multi-module packaging system in Jigsaw, consider just defining everything in one module to simplify distribution.

Standalone SE Application Strategies for Using Jigsaw

1. Avoid using Jigsaw; due to the issues presented in the document.
2. If your application needs to run on Java 8 or earlier, your options will include (note that all options will require multiple launch mechanisms):
    a. Compile your source code as Java 8, but your *module-info.java* as Java 9 for every module shipped (note that this will require either multiple javac build invocations, or generating your own bytecode for the *module-info.class* as an additional step);
    b. Utilize two build pipeline stages that produce two separate target distributions (one Jigsaw Java 9 distribution and one Java 8-or-earlier distribution);
    c. Utilize two build stages as above and then construct a build script to merge the output for each jar to produce a single, multi-version JAR.
3. Avoid relying exclusively on the package exclusion security capabilities of Jigsaw, as they can be disabled via the command line, and the traditional build mentioned in step 2 won't utilize them.
4. Reduce the number of Jigsaw dependencies in your project to the smallest possible, since it's unclear when or if the dependency and package conflict issues could be worked around by build systems such as maven. Consider other strategies such as:
    a. Define your own modules locally to represent a dependency;
    b. Use the shade plugin to relocate packages and merge the dependency into your module (this avoids the duplicate concealed package issue, as well as version conflicts);
    c. Directly include the source of the dependency in your project;
    d. Define your own layer with a custom class-loading facility instead of the standard jigsaw launch to manage conflicts.
5. Be prepared to patch and convert dependencies which encounter compatibility issues with Jigsaw and/or service loader namespace conflicts.
6. Since there is no multi-module packaging system in Jigsaw, consider just defining everything in one module to simplify distribution.

Dynamic Runtime/Container Strategies for Supporting Jigsaw

Note this advice refers to any existing or new greenfield dynamic runtime environment

1. If possible, discourage users from using Jigsaw, and encourage them to produce traditional packaging or utilize other modular technologies. Due to several issues, many of which are presented in the document, this will likely help users of your runtime produce a more reliable system.
2. Due to the limitations expressed in this document (particularly around mutability and hierarchical layers), if support of Jigsaw is desired, it will likely require a complete reimplementation of the Jigsaw contracts utilizing a Jigsaw facade in front of a custom modular class-loading implementation.
3. Due to restrictions in the APIs, in order to adequately support reflective dynamic frameworks, such as dependency injection at runtime, containers will likely need to modify the bytecode of *module-info.class* provided by the user to add appropriate qualified declarations open declarations.
4. If support of lazy loading of packages and/or dynamic package extension is required, a runtime will need to take extreme measures, such as altering the Jigsaw implementation with an agent and/or utilizing Unsafe.
5. In order to support custom serialization frameworks (e.g. *Xstream*), a runtime will need to bypass the package restriction facility in Jigsaw using Unsafe.
6. Since plugin/deployment code built using Jigsaw might have a security model based on Jigsaw package restrictions, a container should try to wall off access as much as possible using any isolation mechanisms available based on the selected class loading strategy.
7. Due to possible conflicts with module names, dependencies, and service names, a runtime should consider rewriting/redefining *module-info.class*.
8. Due to all of the above, runtimes should advise their users that module metadata returned from Java reflection will not match expectation nor what is observed in a standalone Jigsaw execution.