The State of the Module System

AUTOMATIC EDITION

Mark Reinhold

2016/3/8 08:18 -0800 [e36c992f7fd1]

This version: http://openjdk.java.net/projects/jigsaw/spec/sotms/2016-03-08
Latest version: http://openjdk.java.net/projects/jigsaw/spec/sotms/
Comments to: jpms-spec-comments@openjdk.java.net

> *This document is slightly out of date. None of the fundamental concepts has changed but the* requires public *directive has been renamed to* requires transitive*, and several additional capabilities have been added. An update is in preparation and will be posted here when ready.*

This is an informal overview of enhancements to the Java SE Platform prototyped in Project Jigsaw and proposed for JSR 376: The Java Platform Module System. A related document describes enhancements to JDK-specific tools and APIs, which are outside the scope of the JSR.

As described in the JSR, the specific goals of the module system are to provide

- *Reliable configuration*, to replace the brittle, error-prone class-path mechanism with a means for program components to declare explicit dependences upon one another, along with
- *Strong encapsulation*, to allow a component to declare which of its public types are accessible to other components, and which are not.

These features will benefit application developers, library developers, and implementors of the Java SE Platform itself directly and, also, indirectly, since they will enable a scalable platform, greater platform integrity, and improved performance.

Contents

This is the second edition of this document. Relative to the initial edition this edition introduces material on compatibility and migration, revises the description of reflective readability, reorders the text to improve the flow of the narrative, and is organized into a two-level hierarchy of sections and subsections for easier navigation.

There are still many open issues in the design, the resolutions of which will be reflected in future versions of this document.

1 DEFINING MODULES

In order to provide reliable configuration and strong encapsulation in a way that is both approachable to developers and supportable by existing tool chains we treat modules as a fundamental new kind of Java program component. A *module* is a named, self-describing collection of code and data. Its code is organized as a set of packages containing types, *i.e.*, Java classes and interfaces; its data includes resources and other kinds of static information.

1.1 Module declarations

A module's self-description is expressed in its *module declaration*, a new construct of the Java programming language. The simplest possible module declaration merely specifies the name of its module:

```
module com.foo.bar { }
```

One or more `requires` clauses can be added to declare that the module depends, by name, upon some other modules, at both compile time and run time:

```
module com.foo.bar {
    requires org.baz.qux;
}
```

Finally, `exports` clauses can be added to declare that the module makes all, and only, the public types in specific packages available for use by other modules:

```
module com.foo.bar {
    requires org.baz.qux;
    exports com.foo.bar.alpha;
    exports com.foo.bar.beta;
}
```

If a module's declaration contains no `exports` clauses then it will not export any types at all to any other modules.

The source code for a module declaration is, by convention, placed in a file named `module-info.java` at the root of the module's source-file hierarchy. The source files for the `com.foo.bar` module, *e.g.*, might include:

```
module-info.java
com/foo/bar/alpha/AlphaFactory.java
com/foo/bar/alpha/Alpha.java
...
```

A module declaration is compiled, by convention, into a file named module-info.class, placed similarly in the class-file output directory.

Module names, like package names, must not conflict. The recommended way to name a module is to use the reverse-domain-name pattern that has long been recommended for naming packages. The name of a module will, therefore, often be a prefix of the names of its exported packages, but this relationship is not mandatory.

A module's declaration does not include a version string, nor constraints upon the version strings of the modules upon which it depends. This is intentional: It is not a goal of the module system to solve the version-selection problem, which is best left to build tools and container applications.

Module declarations are part of the Java programming language, rather than a language or notation of their own, for several reasons. One of the most important is that module information must be available at both compile time and run time in order to achieve *fidelity across phases*, *i.e.*, to ensure that the module system works in the same way at both compile time and run time. This, in turn, allows many kinds of errors to be prevented or, at least, reported earlier—at compile time—when they are easier to diagnose and repair.

Expressing module declarations in a source file which is compiled, along with the other source files in a module, into a class file for consumption by the Java virtual machine is the natural way in which to establish fidelity. This approach will be immediately familiar to developers, and not difficult for IDEs and build tools to support. An IDE, in particular, could suggest an initial module declaration for an existing component by synthesizing requires clauses from information already available in the component's project description.

1.2 Module artifacts

Existing tools can already create, manipulate, and consume JAR files, so for ease of adoption and migration we define *modular JAR files*. A modular JAR file is like an ordinary JAR file in all possible ways, except that it also includes a module-info.class file in its root directory. A modular JAR file for the above com.foo.bar module, *e.g.*, might have the content:

```
META-INF/
META-INF/MANIFEST.MF
module-info.class
com/foo/bar/alpha/AlphaFactory.class
com/foo/bar/alpha/Alpha.class
...
```

A modular JAR file can be used as a module, in which case its module-info.class file is taken to contain the module's declaration. It can, alternatively, be placed on the ordinary class path, in which case its module-info.class file is ignored. Modular JAR files allow the maintainer of a library to ship a single artifact that works both as a module, on Java SE 9 and later, and as a regular JAR file on the class path, on all

releases. We expect that implementations of Java SE 9 which include a `jar` tool will enhance that tool to make it easy to create modular JAR files.

For the purpose of modularizing the Java SE Platform's reference implementation, the JDK, we will introduce a new artifact format that goes beyond JAR files to accommodate native code, configuration files, and other kinds of data that do not fit naturally, if at all, into JAR files. This format leverages another advantage of expressing module declarations in source files and compiling them into class files, namely that class files are independent of any particular artifact format. Whether this new format, provisionally named "JMOD," should be standardized is an open question.

1.3 Module descriptors

A final advantage of compiling module declarations into class files is that class files already have a precisely-defined and extensible format. We can thus consider `module-info.class` files in a more general light, as *module descriptors* which include the compiled forms of source-level module declarations but also additional kinds of information recorded in class-file attributes which are inserted after the declaration is initially compiled.

An IDE or a build-time packaging tool, *e.g.*, can insert attributes containing documentary information such as a module's version, title, description, and license. This information can be read at compile time and run time via the module system's reflection facilities for use in documentation, diagnosis, and debugging. It can also be used by downstream tools in the construction of OS-specific package artifacts. A specific set of attributes will be standardized but, since the Java class-file format is extensible, other tools and frameworks will be able to define additional attributes as needed. Non-standard attributes will have no effect upon the behavior of the module system itself.

1.4 Platform modules

The Java SE 9 Platform Specification will use the module system to divide the platform into a set of modules. An implementation of the Java SE 9 Platform might contain all of the platform modules or, possibly, just some of them.

The only module known specifically to the module system, in any case, is the base module, which is named `java.base`. The base module defines and exports all of the platform's core packages, including the module system itself:

```
module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
```

```
    ...
}
```

The base module is always present. Every other module depends implicitly upon the base module, while the base module depends upon no other modules.

The remaining platform modules will share the "java." name prefix and are likely to include, *e.g.*, java.sql for database connectivity, java.xml for XML processing, and java.logging for logging. Modules that are not defined in the Java SE 9 Platform Specification but instead specific to the JDK will, by convention, share the "jdk." name prefix.

2 USING MODULES

Individual modules can be defined in module artifacts, or else built-in to the compile-time or run-time environment. To make use of them in either phase the module system must locate them and, then, determine how they relate to each other so as to provide reliable configuration and strong encapsulation.

2.1 The module path

In order to locate modules defined in artifacts the module system searches the *module path*, which is defined by the host system. The module path is a sequence, each element of which is either a module artifact or a directory containing module artifacts. The elements of the module path are searched, in order, for the first artifact that defines a suitable module.

The module path is materially different from the class path, and more robust. The inherent brittleness of the class path is due to the fact that it is a means to locate individual types in all the artifacts on the path, making no distinction amongst the artifacts themselves. This makes it impossible to tell, in advance, when an artifact is missing. It also allows different artifacts to define types in the same packages, even if those artifacts represent different versions of the same logical program component, or different components entirely.

The module path, by contrast, is a means to locate whole modules rather than individual types. If the module system cannot fulfill a particular dependence with an artifact from the module path, or if it encounters two artifacts in the same directory that define modules of the same name, then the compiler or virtual machine will report an error and exit.

The modules built-in to the compile-time or run-time environment, together with those defined by artifacts on the module path, are collectively referred to as the universe of *observable modules*.

2.2 Resolution

Suppose we have an application that uses the above com.foo.bar module and also the platform's java.sql module. The module that contains the core of the application is declared as follows:

```
module com.foo.app {
    requires com.foo.bar;
    requires java.sql;
}
```

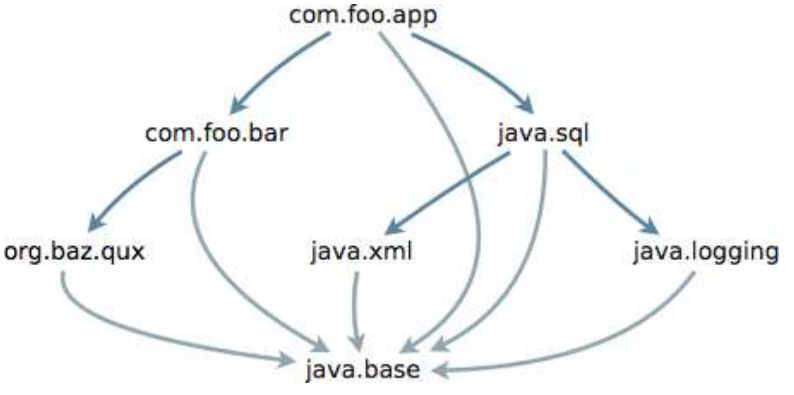
Given this initial application module, the module system *resolves* the dependences expressed in its requires clauses by locating additional observable modules to fulfill those dependences, and then resolves the dependences of those modules, and so forth, until every dependence of every module is fulfilled. The result of this transitive-closure computation is a *module graph* which, for each module with a dependence that is fulfilled by some other module, contains a directed edge from the first module to the second.

To construct a module graph for the com.foo.app module, the module system inspects the declaration of the java.sql module, which is:

```
module java.sql {
    requires java.logging;
    requires java.xml;
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
}
```

It also inspects the declaration of the com.foo.bar module, already shown above, and also those of the org.baz.qux, java.logging, and java.xml modules; for brevity, these last three are not shown here since they do not declare dependences upon any other modules.

Based upon all of these module declarations, the graph computed for the com.foo.app module contains the following nodes and edges:



In this figure the dark blue lines represent explicit dependence relationships, as expressed in requires clauses, while the light blue lines represent the implicit dependences of every module upon the base module.

2.3 Readability

When one module depends directly upon another in the module graph then code in the first module will be able to refer to types in the second module. We therefore say that the first module *reads* the second or, equivalently, that the second module is *readable* by the first. Thus, in the above graph, the com.foo.app module reads the com.foo.bar and java.sql modules but not the org.baz.qux, java.xml, or java.logging modules. The java.logging module is readable by the java.sql module, but no others. (Every module, by definition, reads itself.)

The readability relationships defined in a module graph are the basis of *reliable configuration*: The module system ensures that every

dependence is fulfilled by precisely one other module, that the module graph is acyclic, that every module reads at most one module defining a given package, and that modules defining identically-named packages do not interfere with each other.

Reliable configuration is not just more reliable; it can also be faster. When code in a module refers to a type in a package then that package is guaranteed to be defined either in that module or in precisely one of the modules read by that module. When looking for the definition of a specific type there is, therefore, no need to search for it in multiple modules or, worse, along the entire class path.

2.4 Accessibility

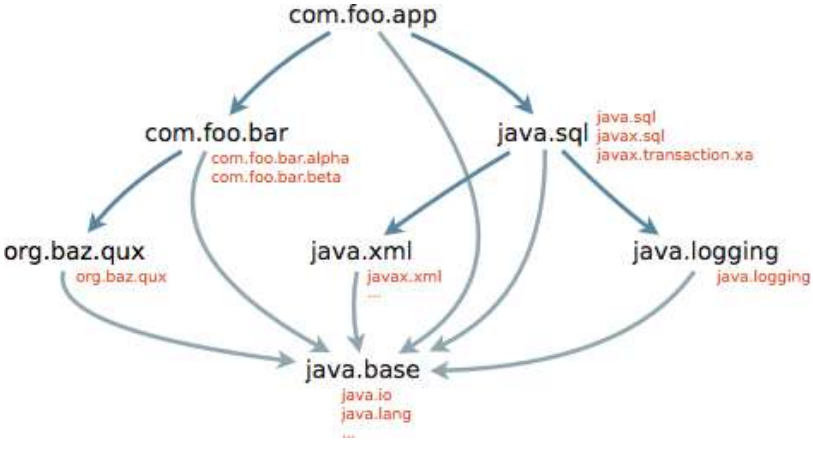
The readability relationships defined in a module graph, combined with the `exports` clauses in module declarations, are the basis of *strong encapsulation*: The Java compiler and virtual machine consider the public types in a package in one module to be *accessible* by code in some other module only when the first module is readable by the second module, in the sense defined above, and the first module exports that package. That is, if two types `S` and `T` are defined in different modules, and `T` is `public`, then code in `S` *can access* `T` if:

1. `S`'s module reads `T`'s module, and
2. `T`'s module exports `T`'s package.

A type referenced across module boundaries that is not accessible in this way is unusable in the same way that a private method or field is unusable: Any attempt to use it will cause an error to be reported by the compiler, or an `IllegalAccessError` to be thrown by the Java virtual machine, or an `IllegalAccessException` to be thrown by the reflective run-time APIs. Thus, even when a type is declared `public`, if its package is not exported in the declaration of its module then it will only be accessible to code in that module.

A method or field referenced across module boundaries is accessible if its enclosing type is accessible, in this sense, and if the declaration of the member itself also allows access.

To see how strong encapsulation works in the case of the above module graph, we label each module with the packages that it exports:



Code in the `com.foo.app` module can access public types declared in the `com.foo.bar.alpha` package because `com.foo.app` depends upon, and

therefore reads, the com.foo.bar module, and because com.foo.bar exports the com.foo.bar.alpha package. If com.foo.bar contains an internal package com.foo.bar.internal then code in com.foo.app cannot access any types in that package, since com.foo.bar does not export it. Code in com.foo.app cannot refer to types in the org.baz.qux package since com.foo.app does not depend upon the org.baz.qux module, and therefore does not read it.

2.5 Implied readability

If one module reads another then, in some situations, it should logically also read some other modules.

The platform's java.sql module, *e.g.*, depends upon the java.logging and java.xml modules, not only because it contains implementation code that uses types in those modules but also because it defines types whose signatures refer to types in those modules. The java.sql.Driver interface, in particular, declares the public method

```
public Logger getParentLogger();
```

where Logger is a type declared in the exported java.util.logging package of the java.logging module.

Suppose, *e.g.*, that code in the com.foo.app module invokes this method in order to acquire a logger and then log a message:

```
String url = ...;
Properties props = ...;
Driver d = DriverManager.getDriver(url);
Connection c = d.connect(url, props);
d.getParentLogger().info("Connection acquired");
```

If the com.foo.app module is declared as above then this will not work: The getParentLogger method returns a Logger, which is a type declared in the java.logging module, which is not readable by the com.foo.app module, and so the invocation of the info method in the Logger class will fail at both compile time and run time because that class, and thus that method, is inaccessible.

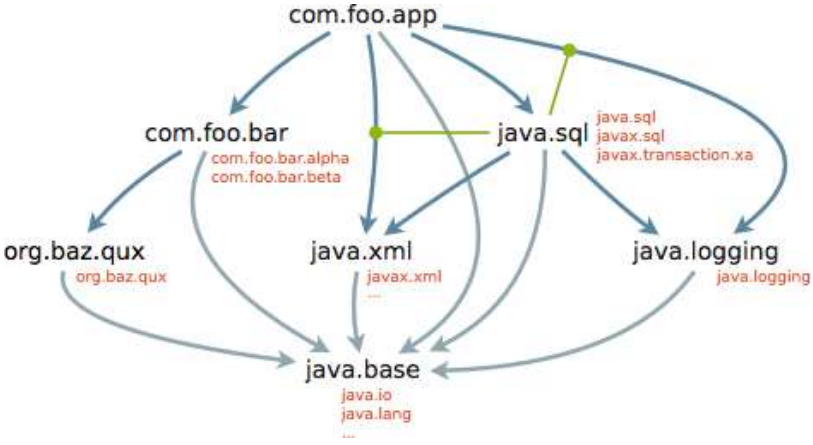
One solution to this problem is to hope that every author of every module that both depends upon the java.sql module and contains code that uses Logger objects returned by the getParentLogger method remembers also to declare a dependence upon the java.logging module. This approach is unreliable, of course, since it violates the principle of least surprise: If one module depends upon a second module then it is natural to expect that every type needed to use the first module, even if the type is defined in the second module, will immediately be accessible to a module that depends only upon the first module.

We therefore extend module declarations so that one module can grant readability to additional modules, upon which it depends, to any module that depends upon it. Such *implied readability* is expressed by including the public modifier in a requires clause. The declaration of the java.sql module actually reads:

```
module java.sql {
    requires public java.logging;
    requires public java.xml;
    exports java.sql;
```

```
        exports javax.sql;
        exports javax.transaction.xa;
    }
```

The public modifiers mean that any module that depends upon the java.sql module will read not only the java.sql module but also the java.logging and java.xml modules. The module graph for the com.foo.app module, shown above, thus contains two additional dark-blue edges, linked by green edges to the java.sql module since they are implied by that module:



The com.foo.app module can now include code that accesses all of the public types in the exported packages of the java.logging and java.xml modules, even though its declaration does not mention those modules.

In general, if one module exports a package containing a type whose signature refers to a package in a second module then the declaration of the first module should include a requires public dependence upon the second. This will ensure that other modules that depend upon the first module will automatically be able to read the second module and, hence, access all the types in that module's exported packages.

3 COMPATIBILITY & MIGRATION

Thus far we have seen how to define modules from scratch, package them into module artifacts, and use them together with other modules that are either built-in to the platform or also defined in artifacts.

Most Java code was, of course, written prior to the introduction of the module system and must continue to work just as it does today, without change. The module system can, therefore, compile and run applications composed of JAR files on the class path even though the platform itself is composed of modules. It also allows existing applications to be migrated to modular form in a flexible and gradual manner.

3.1 The unnamed module

If a request is made to load a type whose package is not defined in any known module then the module system will attempt to load it from the class path. If this succeeds then the type is considered to be a member of a special module known as *the unnamed module*, so as to ensure that

every type is associated with some module. The unnamed module is, at a high level, akin to the existing concept of the unnamed package. All other modules have names, of course, so we will henceforth refer to those as *named modules*.

The unnamed module reads every other module. Code in any type loaded from the class path will thus be able to access the exported types of all other readable modules, which by default will include all of the named, built-in platform modules. An existing class-path application that compiles and runs on Java SE 8 will, thus, compile and run in exactly the same way on Java SE 9, so long as it only uses standard, non-deprecated Java SE APIs.

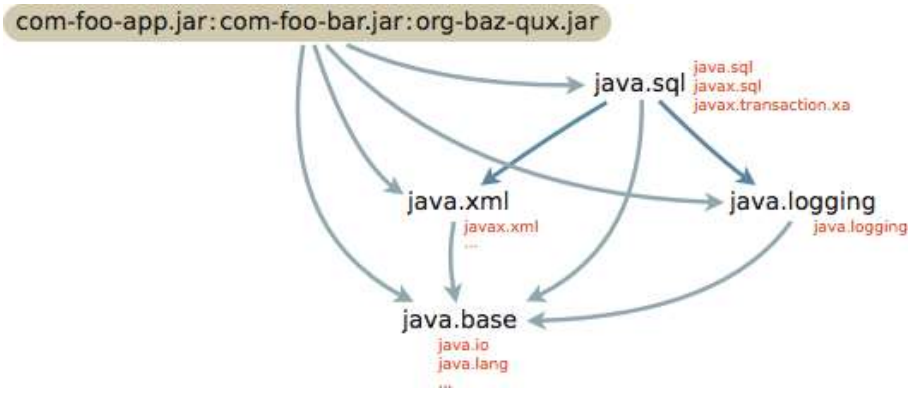
The unnamed module exports all of its packages. This enables flexible migration, as we shall see below. It does not, however, mean that code in a named module can access types in the unnamed module. A named module cannot, in fact, even declare a dependence upon the unnamed module. This restriction is intentional, since allowing named modules to depend upon the arbitrary content of the class path would make reliable configuration impossible.

If a package is defined in both a named module and the unnamed module then the package in the unnamed module is ignored. This preserves reliable configuration even in the face of the chaos of the class path, ensuring that every module still reads at most one module defining a given package. If, in our example above, a JAR file on the class path contains a class file named `com/foo/bar/alpha/AlphaFactory.class` then that file will never be loaded, since the `com.foo.bar.alpha` package is exported by the `com.foo.bar` module.

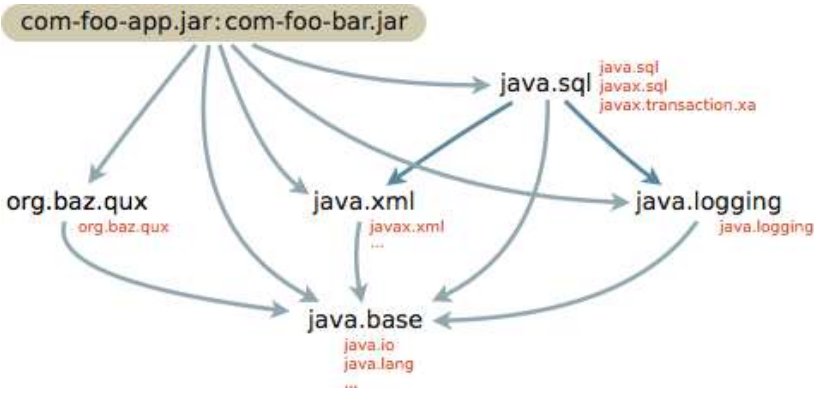
3.2 Bottom-up migration

The treatment of types loaded from the class path as members of the unnamed module allows us to migrate the components of an existing application from JAR files to modules in an incremental, bottom-up fashion.

Suppose, *e.g.*, that the application shown above had originally been built for Java SE 8, as a set of similarly-named JAR files placed on the class path. If we run it as-is on Java SE 9 then the types in the JAR files will be defined in the unnamed module. That module will read every other module, including all of the built-in platform modules; for simplicity, assume those are limited to the `java.sql`, `java.xml`, `java.logging`, and `java.base` modules shown earlier. Thus we obtain the module graph

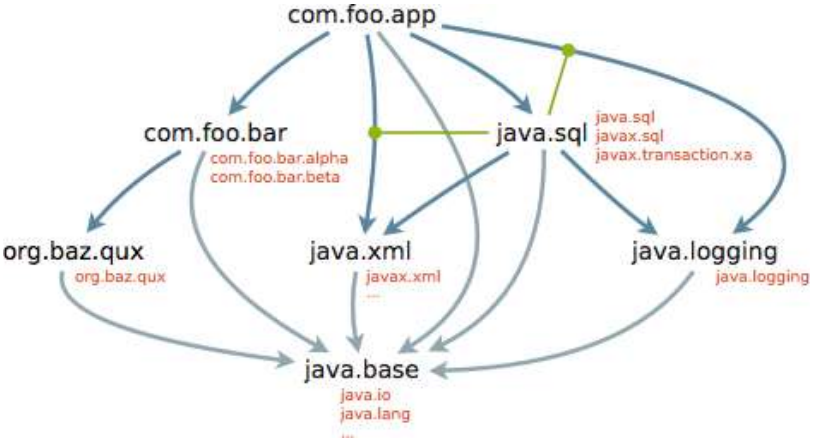


We can immediately convert org-baz-qux.jar into a named module because we know that it does not refer to any types in the other two JAR files, so as a named module it will not refer to any of the types that will be left behind in the unnamed module. (We happen to know this from the original example, but if we did not already know it then we could discover it with the help of a tool such as jdeps.) We write a module declaration for org.baz.qux, add it to the source code for the module, compile that, and package the result as a modular JAR file. If we then place that JAR file on the module path and leave the others on the class path we obtain the improved module graph



The code in com-foo-bar.jar and com-foo-app.jar continues to work because the unnamed module reads every named module, which now includes the new org.baz.qux module.

We can proceed similarly to modularize com-foo-bar.jar, and then com-foo-app.jar, eventually winding up with the intended module graph, shown previously:



Knowing what we do about the types in the original JAR files we could, of course, modularize all three of them in a single step. If, however, org-baz-qux.jar is maintained independently, perhaps by an entirely different team or organization, then it can be modularized before the other two components, and likewise com-foo-bar.jar can be modularized before com-foo-app.jar.

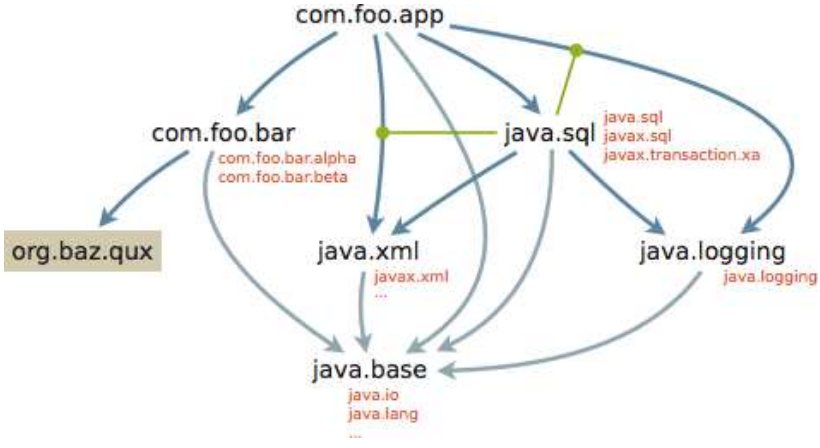
3.3 Automatic modules

Bottom-up migration is straightforward, but it is not always possible. Even if the maintainer of org-baz-qux.jar has not yet converted it into a proper module—or perhaps never will—we might still want to modularize our com-foo-app.jar and com-foo-bar.jar components.

We already know that code in com-foo-bar.jar refers to types in org-baz-qux.jar. If we convert com-foo-bar.jar into the named module com.foo.bar but leave org-baz-qux.jar on the class path, however, then that code will no longer work: Types in org-baz-qux.jar will continue to be defined in the unnamed module but com.foo.bar, which is a named module, cannot declare a dependence upon the unnamed module.

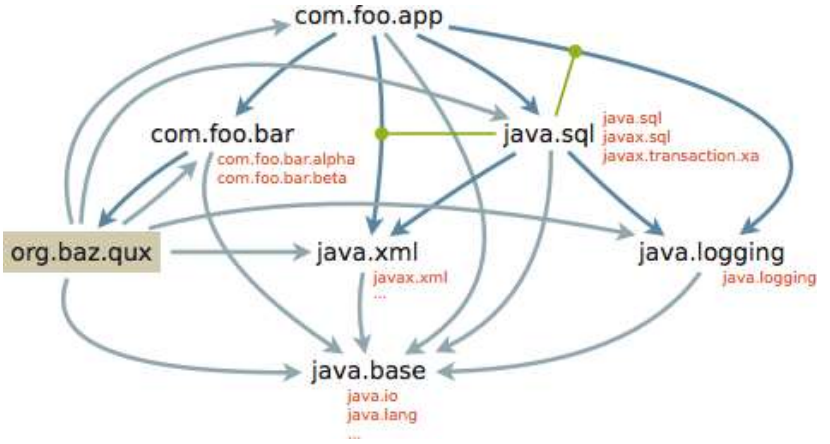
We must, then, somehow arrange for org-baz-qux.jar to appear as a named module so that com.foo.bar can depend upon it. We could fork the source code of org.baz.qux and modularize it ourselves, but if the maintainer is unwilling to merge that change into the upstream repository then we would have to maintain the fork for as long as we might need it.

We can, instead, treat org-baz-qux.jar as an *automatic module* by placing it, unmodified, on the module path rather than the class path. This will define an observable module whose name, org.baz.qux, is derived from that of the JAR file so that other, non-automatic modules can depend upon it in the usual way:



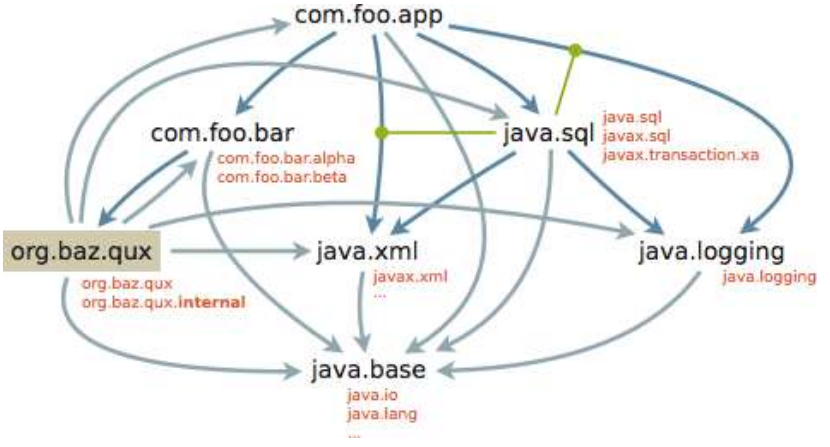
An automatic module is a named module that is defined implicitly, since it does not have a module declaration. An ordinary named module, by contrast, is defined explicitly, with a module declaration; we will henceforth refer to those as *explicit* modules.

There is no practical way to tell, in advance, which other modules an automatic module might depend upon. After a module graph is resolved, therefore, an automatic module is made to read every other named module, whether automatic or explicit:

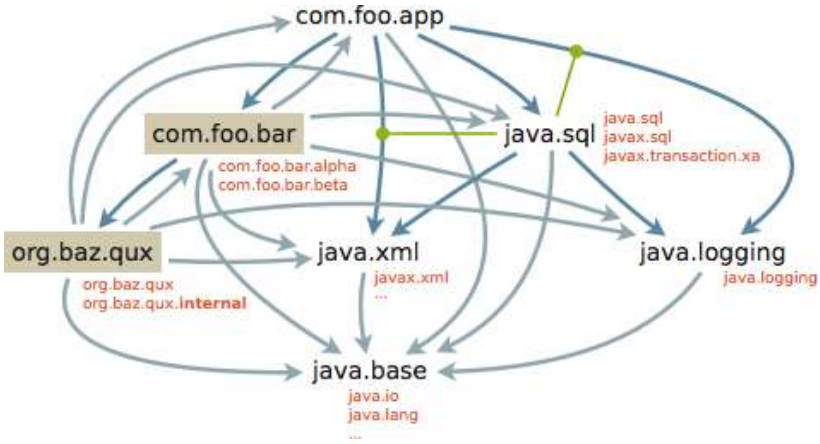


(These new readability edges do create cycles in the module graph, which makes it somewhat more difficult to reason about, but we view these as a tolerable and, usually, temporary consequence of enabling more-flexible migration.)

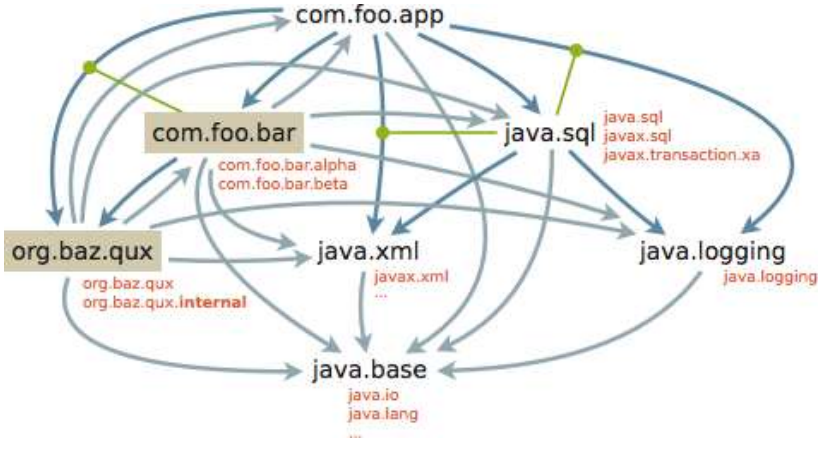
There is, similarly, no practical way to tell which of the packages in an automatic module are intended for use by other modules, or by classes still on the class path. Every package in an automatic module is, therefore, considered to be exported even if it might actually be intended only for internal use:



There is, finally, no practical way to tell whether one of the exported packages in an automatic module contains a type whose signature refers to a type defined in some other automatic module. If, *e.g.*, we modularize com.foo.app first, and treat both com.foo.bar and org.baz.qux as automatic modules, then we have the graph



It is impossible to know, without reading all of the class files in both of the corresponding JAR files, whether a public type in com.foo.bar declares a public method whose return type is defined in org.baz.qux. An automatic module therefore grants implied readability to all other automatic modules:



Now code in com.foo.app can access types in org.baz.qux, although we know that it does not actually do so.

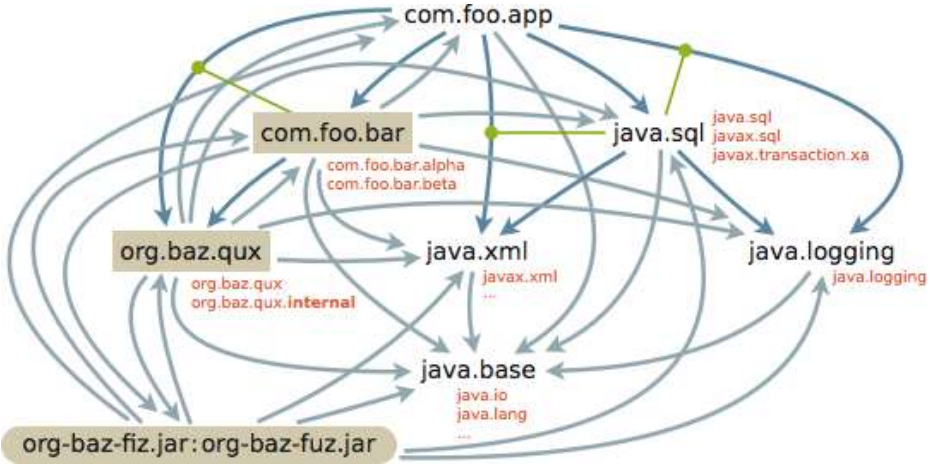
Automatic modules offer a middle ground between the chaos of the class path and the discipline of explicit modules. They allow an existing application composed of JAR files to be migrated to modules from the top down, as shown above, or in a combination of top-down and bottom-up approaches. We can, in general, start with an arbitrary set of JAR-file components on the class path, use a tool such as jdeps to analyze their interdependencies, convert the components whose source code we control into explicit modules, and place those along with the remaining JAR files, as-is, on the module path. The JAR files for components whose source code we do not control will be treated as automatic modules until such time as they, too, are converted into explicit modules.

3.4 Bridges to the class path

Many existing JAR files can be used as automatic modules, but some cannot. If two or more JAR files on the class path contain types in the same package then at most one of them can be used as an automatic module, since the module system still guarantees that every named module reads at most one named module defining a given package and

that named modules defining identically-named packages do not interfere with each other. In such situations it often turns out that only one of the JAR files is actually needed. If the others are duplicates or near-duplicates, somehow placed on the class path by mistake, then one can be used as an automatic module and the others can be discarded. If, however, multiple JAR files on the class path intentionally contain types in the same package then on the class path they must remain.

To enable migration even when some JAR files cannot be used as automatic modules we enable automatic modules to act as bridges between code in explicit modules and code still on the class path: In addition to reading every other named module, an automatic module is also made to read the unnamed module. If our application's original class path had, *e.g.*, also contained the JAR files org-baz-fiz.jar and org-baz-fuz.jar, then we would have the graph



The unnamed module exports all of its packages, as mentioned earlier, so code in the automatic modules will be able to access any public type loaded from the class path.

An automatic module that makes use of types from the class path must not expose those types to the explicit modules that depend upon it, since explicit modules cannot declare dependences upon the unnamed module. If code in the explicit module com.foo.app refers to a public type in com.foo.bar, *e.g.*, and the signature of that type refers to a type in one of the JAR files still on the class path, then the code in com.foo.app will not be able to access that type since com.foo.app cannot depend upon the unnamed module. This can be remedied by treating com.foo.app as an automatic module temporarily, so that its code can access types from the class path, until such time as the relevant JAR file on the class path can be treated as an automatic module or converted into an explicit module.

4 SERVICES

The loose coupling of program components via service interfaces and service providers is a powerful tool in the construction of large software systems. Java has long supported services via the java.util.ServiceLoader class, which locates service providers at run time by searching the class path. For service providers defined in modules we must consider how to locate those modules amongst the set of observable modules, resolve

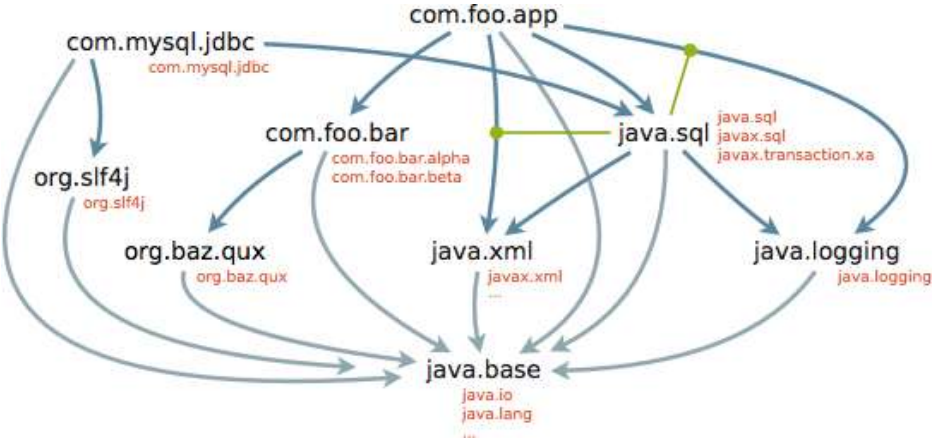
their dependences, and make the providers available to the code that uses the corresponding services.

Suppose, *e.g.*, that our com.foo.app module uses a MySQL database, and that a MySQL JDBC driver is provided in an observable module which has the declaration

```
module com.mysql.jdbc {
    requires java.sql;
    requires org.slf4j;
    exports com.mysql.jdbc;
}
```

where org.slf4j is a logging library used by the driver and com.mysql.jdbc is the package that contains the implementation of the java.sql.Driver service interface. (It is not actually necessary to export the driver package, but we do so here for clarity.)

In order for the java.sql module to make use of this driver, the ServiceLoader class must be able to instantiate the driver class via reflection; for that to happen, the module system must add the driver module to the module graph and resolve its dependences, thus:



To achieve this the module system must be able to identify any uses of services by previously-resolved modules and, then, locate and resolve providers from within the set of observable modules.

The module system could identify uses of services by scanning the class files in module artifacts for invocations of the ServiceLoader::load methods, but that would be both slow and unreliable. That a module uses a particular service is a fundamental aspect of that module's definition, so for both efficiency and clarity we express that in the module's declaration with a uses clause:

```
module java.sql {
    requires public java.logging;
    requires public java.xml;
    exports java.sql;
    exports javax.sql;
    exports javax.transaction.xa;
    uses java.sql.Driver;
}
```

The module system could identify service providers by scanning module artifacts for META-INF/services resource entries, as the ServiceLoader

class does today. That a module provides an implementation of a particular service is equally fundamental, however, so we express that in the module's declaration with a provides clause:

```
module com.mysql.jdbc {
    requires java.sql;
    requires org.slf4j;
    exports com.mysql.jdbc;
    provides java.sql.Driver with com.mysql.jdbc.Driver;
}
```

Now it is very easy to see, simply by reading these modules' declarations, that one of them uses a service that is provided by the other.

Declaring service-provision and service-use relationships in module declarations has advantages beyond improved efficiency and clarity. Service declarations of both kinds can be interpreted at compile time to ensure that the service interface (*e.g.*, java.sql.Driver) is accessible to both the providers and the users of a service. Service-provider declarations can be further interpreted to ensure that providers (*e.g.*, com.mysql.jdbc.Driver) actually do implement their declared service interfaces. Service-use declarations can, finally, be interpreted by ahead-of-time compilation and linking tools to ensure that observable providers are appropriately compiled and linked prior to run time.

For migration purposes, if a JAR file that defines an automatic module contains META-INF/services resource entries then each such entry is treated as if it were a corresponding provides clause in a hypothetical declaration of that module. An automatic module is considered to use every available service.

5 ADVANCED TOPICS

The remainder of this document addresses advanced topics which, while important, may not be of interest to most developers.

5.1 Reflection

To make the module graph available via reflection at run time we define a Module class in the java.lang.reflect package and some related types in a new package, java.lang.module. An instance of the Module class represents a single module at run time. Every type is in a module, so every Class object has an associated Module object, which is returned by the new Class::getModule method.

The essential operations on a Module object are:

```
package java.lang.reflect;

public final class Module {
    public String getName();
    public ModuleDescriptor getDescriptor();
    public ClassLoader getClassLoader();
    public boolean canRead(Module target);
    public boolean isExported(String packageName);
}
```

where ModuleDescriptor is a class in the java.lang.module package, instances of which represent module descriptors; the getClassLoader method returns the module's class loader; the canRead method tells whether the module can read the target module; and the isExported method tells whether the module exports the given package.

The java.lang.reflect package is not the only reflection facility in the platform. Similar additions will be made to the compile-time javax.lang.model package in order to support annotation processors and documentation tools.

5.2 Reflective readability

A *framework* is a facility that uses reflection to load, inspect, and instantiate other classes at run time. Examples of frameworks in the Java SE Platform itself are service loaders, resource bundles, dynamic proxies, and serialization, and of course there are many popular external framework libraries for purposes as diverse as database persistence, dependency injection, and testing.

Given a class discovered at run time, a framework must be able to access one of its constructors in order to instantiate it. As things stand, however, that will usually not be the case.

The platform's streaming XML parser, *e.g.*, loads and instantiates the implementation of the XMLInputFactory service named by the system property javax.xml.stream.XMLInputFactory, if defined, in preference to any provider discoverable via the ServiceLoader class. Ignoring exception handling and security checks the code reads, roughly:

```
String providerName
    = System.getProperty("javax.xml.stream.XMLInputFactory");
if (providerName != null) {
    Class providerClass = Class.forName(providerName, false,
                                        Thread.getContextClassLoader());
    Object ob = providerClass.newInstance();
    return (XMLInputFactory)ob;
}
// Otherwise use ServiceLoader
...
```

In a modular setting the invocation of Class::forName will continue to work so long as the package containing the provider class is known to the context class loader. The invocation of the provider class's constructor via the reflective newInstance method, however, will not work: The provider might be loaded from the class path, in which case it will be in the unnamed module, or it might be in some named module, but in either case the framework itself is in the java.xml module. That module only depends upon, and therefore reads, the base module, and so a provider class in any other module will be not be accessible to the framework.

To make the provider class accessible to the framework we need to make the provider's module readable by the framework's module. We could mandate that every framework explicitly add the necessary readability edge to the module graph at run time, as in an earlier version of this document, but experience showed that approach to be cumbersome and a barrier to migration.

We therefore, instead, revise the reflection API simply to assume that any code that reflects upon some type is in a module that can read the module that defines that type. This enables the above example, and other code like it, to work without change. This approach does not weaken strong encapsulation: A public type must still be in an exported package in order to be accessed from outside its defining module, whether from compiled code or via reflection.

5.3 Class loaders

Every type is in a module, and at run time every module has a class loader, but does a class loader load just one module? The module system, in fact, places few restrictions on the relationships between modules and class loaders. A class loader can load types from one module or from many modules, so long as the modules do not interfere with each other and all of the types in any particular module are loaded by just one loader.

This flexibility is critical to compatibility, since it allows us to retain the platform's existing hierarchy of built-in class loaders. The bootstrap and extension class loaders still exist, and are used to load types from platform modules. The application class loader also still exists, and is used to load types from artifacts found on the module path.

This flexibility will also make it easier to modularize existing applications which already construct sophisticated hierarchies or even graphs of custom class loaders, since such loaders can be upgraded to load types in modules without necessarily changing their delegation patterns.

5.4 Unnamed modules

We previously learned that if a type is not defined in a named, observable module then it is considered to be a member of the unnamed module, but with which class loader is the unnamed module associated?

Every class loader, it turns out, has its own unique unnamed module, which is returned by the new `ClassLoader::getUnnamedModule` method. If a class loader loads a type that is not defined in a named module then that type is considered to be in that loader's unnamed module, *i.e.*, the `getModule` method of the type's `Class` object will return its loader's unnamed module. The module colloquially referred to as "*the* unnamed module" is, then, simply the unnamed module of the application class loader, which loads types from the class path when they are in packages not defined by any known module.

5.5 Layers

The module system does not dictate the relationships between modules and class loaders, but in order to load a particular type it must somehow be able to find the appropriate loader. At run time, therefore, the instantiation of a module graph produces a *layer*, which maps each module in the graph to the unique class loader responsible for loading the types defined in that module. The *boot layer* is created by the Java virtual machine at startup, by resolving the application's initial module against the observable modules, as discussed earlier.

Most applications, and certainly all existing applications, will never use a layer other than the boot layer. Multiple layers can, however, be of

use in sophisticated applications with plug-in or container architectures such as application servers, IDEs, and test harnesses. Such applications can use dynamic class loading and the reflective module-system API, thus far described, to load and run hosted applications that consist of one or more modules. Two additional kinds of flexibility are, however, often required:

- A hosted application might require a different version of a module that is already present. A Java EE web application, *e.g.*, may require a different version of the JAX-WS stack, which is in the java.xml.ws module, than the one that is built-in to the run-time environment.
- A hosted application might require service providers other than the providers that have already been discovered. A hosted application might even embed its own preferred providers. A web application, *e.g.*, might include a copy of its preferred version of the Woodstox streaming XML parser, in which case the ServiceLoader class should return that provider in preference to any others.

A container application can create a new layer for a hosted application on top of an existing layer by resolving that application's initial module against a different universe of observable modules. Such a universe can contain alternate versions of upgradeable platform modules and other, non-platform modules already present in the lower layer; the resolver will give these alternate modules priority. Such a universe can also contain different service providers than those already discovered in the lower layer; the ServiceLoader class will load and return these providers before it returns providers from the lower layer.

Layers can be stacked: A new layer can be built on top of the boot layer, and then another layer can be built on top of that. As a result of the normal resolution process the modules in a given layer can read modules in that layer or in any lower layer. A layer's module graph can hence be considered to include, by reference, the module graphs of every layer below it.

5.6 Qualified exports

It is occasionally necessary to arrange for some types to be accessible amongst a set of modules yet remain inaccessible to all other modules.

Code in the JDK's implementations of the standard java.sql and java.xml modules, *e.g.*, makes use of types defined in the internal sun.reflect package, which is in the java.base module. In order for this code to access types in the sun.reflect package we could simply export that package from the java.base module:

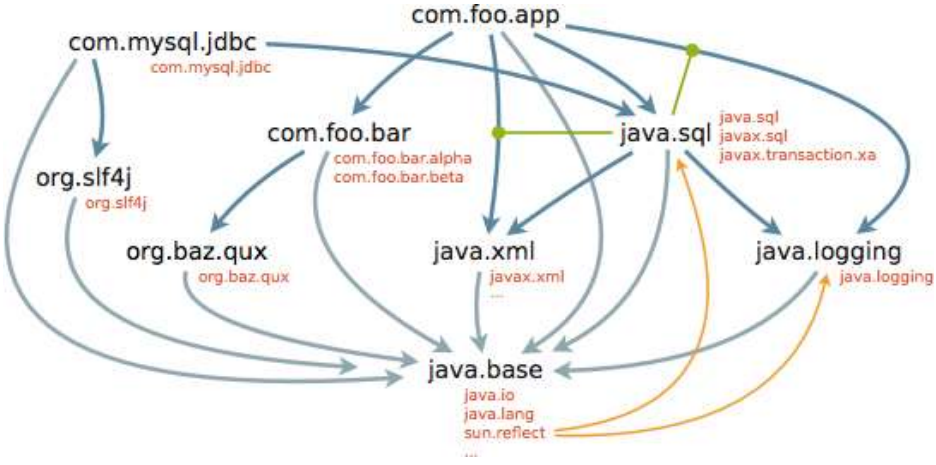
```
module java.base {
    ...
    exports sun.reflect;
}
```

This would, however, make every type in the sun.reflect package accessible to every module, since every module reads java.base, and that is undesirable because some of the classes in that package define privileged, security-sensitive methods.

We therefore extend module declarations to allow a package to be exported to one or more specifically-named modules, and to no others. The declaration of the java.base module actually exports the sun.reflect package only to a specific set of JDK modules:

```
module java.base {
    ...
    exports sun.reflect to
        java.corba,
        java.logging,
        java.sql,
        java.sql.rowset,
        jdk.scripting.nashorn;
}
```

These *qualified exports* can be visualized in a module graph by adding another type of edge, here colored gold, from packages to the specific modules to which they are exported:



The accessibility rules stated earlier are refined as follows: If two types S and T are defined in different modules, and T is public, then code in S can access T if:

1. S's module reads T's module, and
2. T's module exports T's package, *either directly to S's module or to all modules*.

We also extend the reflective Module class with a method to tell whether a package is exported to a specific module, rather than to all modules:

```
public final class Module {
    ...
    public boolean isExported(String packageName, Module target);
}
```

Qualified exports can inadvertently make internal types accessible to modules other than those intended, so they must be used with care. An adversary could, *e.g.*, name a module java.corba in order to access types in the sun.reflect package. To prevent this we can analyze a set of related modules at build time and record, in each module's descriptor, hashes of the content of the modules that are allowed to depend upon it and use its qualified exports. During resolution we verify, for any

module named in a qualified export of some other module, that the hash of its content matches the hash recorded for that module name in the second module. Qualified exports are safe to use in an untrusted environment so long as the modules that declare and use them are tied together in this way.

SUMMARY

The module system described here has many facets, but most developers will only need to use some of them on a regular basis. We expect the basic concepts of module declarations, modular JAR files, the module path, readability, accessibility, the unnamed module, automatic modules, and modular services to become reasonably familiar to most Java developers in the coming years. The more advanced features of reflective readability, layers, and qualified exports will, by contrast, be needed by relatively few.

ACKNOWLEDGEMENTS

This document includes contributions from Alan Bateman, Alex Buckley, Mandy Chung, Jonathan Gibbons, Chris Hegarty, Karen Kinnear, and Paul Sandoz.