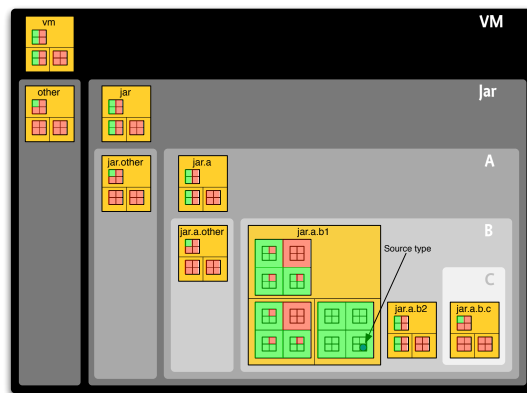
JAVA 294 (NESTED) MODULES



Draft 1.0

aQute

2009-02-10

Introduction

This is a short report on defining a strawman architecturel for nested modules and their visibility rules in Java, related to the JSR 294 effort. It was requested by the spec lead on the mailing JSR 294 list. The purpose of this document is to describe the concepts of a nested module system so that the JSR 294 expert group can create a common understanding and terminology. It is understood that after the concepts are agreed upon, this model requires translation into the current access control descriptions in the Java Language Specification. It is not an attempt to describe the necessary changes in that document.

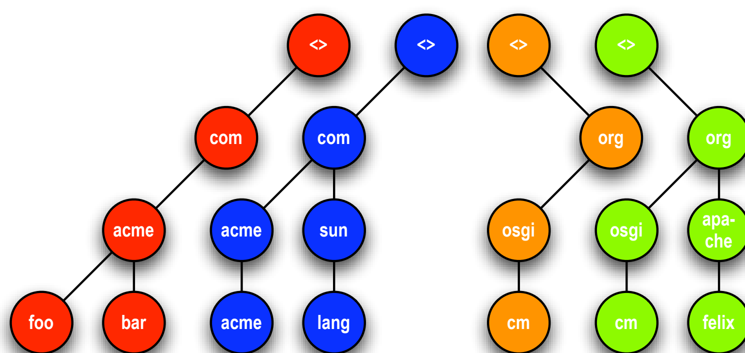
Several concepts in this report are derived from earlier discussions on the mailing lists of JSR 294, JSR 277 (specifically Alex Buckley's proposals) and OSGi groups, as well as private conversations with, among others, Richard Hall, Glyn Normington, and BJ Hargrave. This report does not contain any ideas to which the author claims authorship.

This document is a strawman and is definitely not finished or completely worked out. Expect errors and misunderstandings from the author.

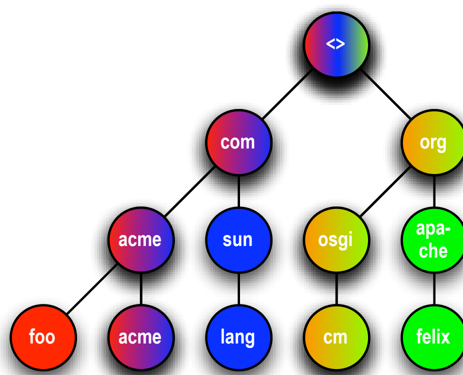
Today's Situation

Java is an environment that has *sources* and *byte codes*. The compiler compiles the sources into byte codes. The JVM executes the byte codes when loaded through a *class loader,* that finds the byte codes for a given type. The Java language has a name space for all its types. Two types with the same name are the same type. However, this name space is an optical illusion. Runtime and compile time create this illusion in significantly different ways and with different side effects.

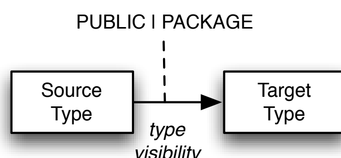
During compilation, the compiler is fed sources from one or more *containers*, for example a src directory with sources, listed on the *source path*. and JAR files and directories from the class path containing byte codes. Each container provides a hierarchical package name space, where the packages contain the types. During compilation, the compiler collapses all these hierarchical name spaces from the containers on the source and class path in a single hierarchical view to match the illusion in the Java language. The compiler uses the ordering of the class path to find a type where the first container with the type delivers the byte codes. Container boundaries are completely ignored, which results in split packages if different containers contain content for the same packages. Split packages can cause confusion: two types in the same package can come from different containers or types can be shadowed. To illustrate, the following pictures shows the different containers and their local package tree. Each container has its own color.



In the end, the view that the compiler will see, by collapsing the containers' namespace, is:



The Java access model is two phased. The first phase is *type visibility* and the second phase is *member visibility.* Visibility is defined between two types. The *source* type is the type that makes the reference to a *target* type. Java has PUBLIC and PACKAGE access for type visibility. PACKAGE access requires the source and target type to belong to the same package. In runtime, there is an additional constraint that the types are from the same class loader as well.

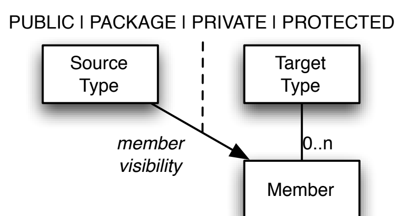


Type S has visibility of type T when:

```
T access = PUBLIC     :     true
T access = PACKAGE    :     T package = S package
                      &&    T loader = S loader
```

A source type can have access to a target type's member only if it has access to the target type first. Access is granted depending on the target type's member access modifier and field of use of the access (inheritance or not). Member access modifiers are PUBLIC, PACKAGE, PRIVATE and PROTECTED. PRIVATE is restricted to where the source type is the same as the target type. Inner classes are from the compilers perspective in lexical scope but in runtime translated to standard classes with synthetic accessors that provide the lexical scope without introducing new modifiers. Therefore, inner classes provide a set of additional rules and constraints that are mapped to these basic rules by the compiler.[1]

PROTECTED is special because it only requires the field of use to be inheritance and that the target type is visible.



Type S has visibility of member m in type T for field of use I (inheritance or not) when:

[1] This is a cop out for now, I need to look at this in more detail but I do not think is overly relevant for the core proposal.

```
S has visibility of T &&
        m access = PUBLIC    :      true
        m access = PACKAGE   :      T package = S package
                            &&     T loader = S
        m access = PRIVATE   :      T = S
        m access = PROTECTED:      I
```

Interfaces require that their implementers make all the interface methods public.

The flexibility of class loaders allows developers to control the *class space* of a type during runtime. A class space is a name space; it allows each type name to be used once. Class loaders can therefore be used to provide additional features and flexibility over the specified semantics in the compiler. The mechanism for class loaders is quite simple, a class loader is requested to load a class by name. If the class has not yet been loaded, the class loader finds the class' byte codes in a container and calls defineClass. This method turns the byte codes into a Class object. Class loaders can delegate the load of a class to another class loader. The default strategy is parent-first, then if not found search the container. Classes loaded by a class loader will request their own class loaders to find referred classes. he class loader can define any delegation model it likes to create the class space.

This powerful capability has led to *module systems*. A module system controls the class loading during runtime and thereby provides extra semantics and features. For example, OSGi, and in the future Jigsaw, are examples of a module system that shape a type's class space to provide additional features with custom class loaders.

Problem Description

JSR 294 tries to address the problem that not all types are required to be visible to all other types when they are in a different package. In almost any application, there are large parts of the code that do not require access from outside the application; it is often pure implementation code. Allowing other applications to use this implementation code could create unwanted side effects during runtime, it could create unexpected maintenance problems, and it would require more effort to write due to the external access. Also, when the scope of accessibility of a type is known it enables certain performance optimizations.

Java today provides modularity with packages: it can restrict access to types in the same package. However, this only provides two choices: public or in the same package. There is a need in the industry to provide a more finer grained model where a number of packages have access to each other's implementation details without exposing these details to the rest of the world.

Additionally, there is a need to mark the boundary of the containers. Currently, an application can not restrict a type or its members to its enclosing container because the container is not an entity in the language nor is there an access modifier that provides this restriction. That said, though the container should be a boundary, there is also a need to allow multiple containers to provide aspects of the same module. Large systems that can not be delivered in a single JAR could still require private access.

A special case is formed with interfaces that require that the methods that implement the interface are made public. In a modular system, it should be possible to define interfaces that can only be used inside a module.

Requirements

The following requirements have driven this proposal.

- Provide an access model that limits access to within the same container

- Provide an access model where a number of packages can collaborate without exposing implementation details to outsiders.

- Provide a model where module systems (Project Jigsaw, OSGi) can control the modularity aspects and add their own semantics and features.

- It must be possible to package a compiled code base in a new module structure without altering the byte codes.

- The proposed model must be compatible in concepts as well as feel with Java 6. Earlier Java could should run unmodified.
- The compiler must not become dependent on the semantics of a specific module system. That is the compiler should not have specific knowledge of OSGi or Jigsaw.
- Provide module interfaces that can be implemented with MODULE private methods.
- Allow multiple containers to contribute to the same module.

Proposed Solution

Top Level Module

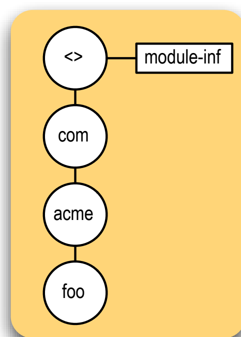
A module is defined in the name space of a *container*. A *top level module* definition consists of a module-info.java class. in the root of the container. From the compiler's perspective, top level module definitions of different containers are distinct even though they are in the same default package. If a container has a top level module definition, then types and members marked with MODULE access modifiers inside that container must not be visible to types outside this container. Types therefore do not have to encode their module membership in the source nor in their byte codes, the location of the module-info file is sufficient. for the compiler to establish the top level module membership.

In the following example, it is shown how com.acme.foo.Public iclass s part of the M module.

```
<> (container root)
  module-info.java:   module M;

  com/acme/foo
       Public.java:   package com.acme.foo;
                      public class Public {}
```

Or in a picture:



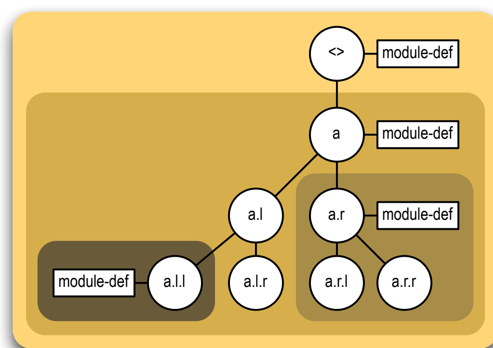
If the top level module definitions of different containers have the same name, then the compiler must collapse their content into a single module and allow split packages between these containers, as is the case today. If these top level modules have a different name, then they must be treated as distinctly different modules and split packages are not allowed.

Containers without a top level module definition belong to a singleton *root module* in the compiler, MODULE modifiers in such a container are then effectively the same as PUBLIC. Split packages between these containers is allowed.

Nested Modules

The previous module definition can easily be extended to a nested module definition by allowing module definitions to occur anywhere in the container's hierarchical package name space. This hierarchy provides the natural scoping and parenting of the nesting without requiring redundant information in source and class files. A module's scope contains its own package and any descendant packages. Its parent module is defined to be the closest module definition in its list of ancestors, starting with its parent. If none of the ancestor packages contains a module definition, then the parent is the singleton root module.

The following illustrations the scope using coloring. The outer most box is the container.



Nested modules can provide the fine grained capability of grouping packages and restricting access to within this group using the same concept and keyword as a top level module. If there is a top level module, it will by definition be the ancestor of all module definitions in the container. Modules inside other modules have MODULE access to their ancestors but ancestor modules have no module access to their descendants.[2]

The compiler must merge different modules if they have the same name and share the same ancestors. That is, if one container provides the modules Top/M, and the other container as well, then Top/M will be the same module. for both containers, even if the package name differs. However, if one container defines the modules Top/M and the other Top/Int/M, then that M is not the same module if if their package names are the same.

For consistency, module definitions must encode their package name in the source. For example, a module definition for all sub packages down from the com.acme.foo package must look like:

```
package com.acme.foo;
module Foo;
```

Rules

The rules for visibility are extended to add a module type[3]:

Type S has visibility of type T when:

```
T access = PUBLIC     :     true
T access = MODULE     :     T module = S module
                      ||    S module isAncestor( T )
T access = PACKAGE    :     T package = S package
                      &&    T loader = S loader
```

Type S has visibility of member m in type T when:

[2] In practice, there seems to be a need to have some visibility between siblings in the same parent. For example, Felix would like to package its code in a Felix module. This will have the modules Main, ModuleLayer, ServiceLayer, and LifeCycleLayer. With hierarchical access, these modules cannot privately collaborate, which is a very common requirement. Allowing access to all siblings with MODULE makes it impossible for a sibling to hide anything. An EXPORT modifier could provide these semantics. at a small cost. However it seems to imply the need to re-export to allow this visibility to crawl the ancestors, which is likely more complex. The rules for this modifier would be quite easy. If they have the same parent module, they would have access. This sounds like an interesting concept that could be further investigated because it feels like the current rules are a bit too restricted for pragmatic use cases.

[3] This again ignores some of the intricacies of the inner classes and synthetic accessors

```
S has visibility of T &&
        m access = PUBLIC   :      true
        m access = MODULE   :      T module = S module
                            ||     S module isAncestor( T )
        m access = PACKAGE  :      T package = S package
                            &&     T loader = S
        m access = PRIVATE  :      T = S
        m access = PROTECTED:      I
```

The isAncestor function of module defined as:

```
isAncestor(X)
        this = X ? true : (parent != null && parent.isAncestor( T ))
```

Interfaces

Interfaces are types and can be PUBLIC, PACKAGE, or MODULE. In Java today, the methods in the interface are by definition PUBLIC. This requires that the implementation methods are also PUBLIC. JSR 294 changes this. A MODULE interface makes all methods MODULE per default. This means they can be implemented by PUBLIC or MODULE methods[4]. For example

```
module interface Foo {
        void bar();
}

module class FooImpl implements Foo {
        module bar();
}
```

Syntax

A module definition has the following syntax (should follow the JLS rules)

```
[package-def]
[annotations]
'module' <fqn> ';'
```

The 'module' keyword can be used anywhere 'public' can be used. The source code nor the byte code encode the module membership information to allow re-packaging of compiled systems without byte code alterations.

For example, a source tree using modules.

[4] Not deeply investigated

```
module-info:         module Felix;

org/apache/felix/framework
        Felix:       package org.apache.felix.framework;
                     module class Felix {... }

        BundleImpl:  package org.apache.felix.framework;
                     module class BundleImpl {... }


org/apache/felix/launcher
        module-info: package org.apache.felix.launcher;
                     module Felix.Launcher;

        Main:        package org.apache.felix.launcher;
                     public class Main { ... new Felix() ... }

        Support:     package org.apache.felix.launcher;
                     module class Support {}

org/apache/felix/launcher/unix
        module-info: package org.apache.felix.launcher.unix;
                     module FelixOnUnix

        Unix         package org.apache.felix.launcher.unix;
                     public class Unix { ... new Support() ... }
```

Runtime

The previous sections have focused on the language and compiler aspects. However, Java provides a different model in the VM then as viewed from the compiler. This happens because the Java runtime has *class loaders*. In JSR 294 classes continue to be loaded through class loaders but class loaders get the additional responsibility to assign the module to the types. This then requires a default model as well as an extension model that allow module system to provide additional semantics and constraints.

In a JSR 294 VM, each type T must be associated with a module M. The VM must provide a root module that is the ancestor of all other modules. Modules are parented to their enclosing module.

JSR 294 Class loaders will have an additional defineClass method that takes a Module parameter. The existing defineClass methods are modified to use the class loader to find the module definitions according to the closest ancestor rule. This module membership is therefore based on the class loader's resource view. If a class loader loads classes from multiple containers (for example the URL Class Loader), then it is the responsibility of that class loader to separate the container package name spaces and use the appropriate module definition from the type's container because the default class loader behavior will flatten the containers name spaces. That is, a default class loader collapses the containers in the traditional way.

A default class loader can therefore have at most one top level module definition. If such a module exists, it must be created and used as parent for all enclosing modules. The Module class provides an instance method Module defineModule(byte[], Map<String,Map> namespace) that defines a module from byte codes if it does not exist in the given namespace, otherwise it returns the module from the name space. map[5]. The default class loader will ensure that top level modules are using the same Module instance if they have the same name, this allows a module to be delivered in multiple containers and optionally different class loaders and still share module privacy. However, this only works for top level modules. All top level modules created by the default class loader must be parented at the VM singleton root module.

Custom class loaders can fully control the association from type to module. They can provide module namespace(s) as described before, or they can use the createModule(String) instance method to create modules at will, or they can provide additional intermediate parents This will also, for example, allow a module system to define a module for a JAR that has no top level module definition. Therefore, module systems are free to assign modules to types with some more elaborate scheme than the default allows for. For example, a module system can create a module that is shared between class loaders (as the default

[5] This leaves us with the problem of annotations. Two different module definitions of the same module can have different annotations. Not sure what to do about this. We can merge them, we can ignore them, or we can provide an elaborate API to access each one of them based on the container. None is overly attractive.

does) or it can use the same modules for different containers that are loaded through one class loader. Though these additional enabled semantics are not always pure, they will be required to handle compatibility with existing practices.