Review: Java Packages and Classloaders

David

The Land of Bluejays
The Greatest City in America, MD21218

In this review, two issues related to Java modularity, packages and classloaders, are reviewed. Packages are Java's answer to module systems and play a central role for Java modularity. Classloaders are not a typical modularity issue, but lazy class loading and classloader delegation have some effects on module system design.

1 O Module, Where Art Thou?

Java's module system relies on *packages*. Packages are uniquely identified by *fully qualified names*, such as `java.lang`. Classes are the only type of exported features of Java packages. Any class defined in package `java.lang` needs to have its defining `.class` file (or `.java` file) put to a directory `%/java/lang`, where `%` is given by `CLASSPATH`. In addition, any class in this package needs to declare **package** `java.lang` on top of its source code definition.

In this section, our focus is to show how `CLASSPATH` works, and the difference between the logical view of packages and its physical representations as directories. Toward the end of this section, we argue, even though couter-intuitive, Java module system is not hierarchical.

1.1 Motivation for `CLASSPATH`

Java's `CLASSPATH` is designed for a very practical reason: where to find modules? This is a question to answer for any practical module systems. Think about a ML module system. If a programmer writes `M.x`, then the first question is: Where is `M` in a physical file system?

Before explaining how `CLASSPATH` works, let me first clarify one point: Why is the question to ask "where to find modules?", not "where to find classes?". Technically speaking, the `CLASSPATH` in Java plays a role to find both classes and packages. The interesting thing is, all classes that are directly locatable by a `CLASSPATH` entry can be imagined to belong to a package called **default**. In this sense, a more uniform way is to say `CLASSPATH` is used to find packages. For example, if we have `CLASSPATH = abc: def`, and the file system hierarchy looks like this:

- `abc`
  - `ClassA.java` with no package identifier
  - `pack1`
    - `ClassB.java` with package identifier **package** `pack1`

- `def`
  - `ClassC.java` with no package identifier
  - `pack1`
    - `ClassB.java` with package identifier **package** `pack1`
  - `pack2`
    - `ClassE.java` with package identifier **package** `pack2`

It can thus be said that `CLASSPATH` entry `abc` helps locate two modules: **default** (with one feature `ClassA`) and `pack1` (with one feature `ClassB`), and entry `def` helps locate three modules: **default** (with one feature `ClassC`), `pack1` (with one feature `ClassB`) and `pack2` (with one feature `ClassE`).

1.2 Logical View of Java Modules

The `CLASSPATH` as a whole can be interpreted as a merging of namespaces: each `CLASSPATH` entry defines a namespace with multiple modules defined, and the entire `CLASSPATH` merges all the namespaces, with all modules included. If two modules are of the same name, they are merged into one module, with features of the two modules merged. For instance, the aforementioned `CLASSPATH` with the file system hierarchy described as above is equivalent to a lookup table with the following logical modules:

- **default** module: with features `ClassA` and `ClassC`.
- `pack1` module: with two features by the same name `ClassB`
- `pack2` module: with one feature `ClassE`.

Some interesting issues related to this are:

- Since modules by the same name will have their internal features merged, it is easy to conclude that one logical package in Java can actually be defined in multiple physical directories, as long as they have the same identifier. See `pack1` package above.
- Similarly, all classes directly appearing in any of the `CLASSPATH` entries, even if they might belong to different physical directories, belong to the same logical package **default**. See `ClassA` and `ClassC` above.
- In each module, feature (class) name uniqueness is not required. Java searches the namespace represented by every `CLASSPATH` entry, and the first one defining the matched class name is picked. This is equivalent to many mixin module systems where name clash of export features is not a problem during mixin composition; the left most one is automatically selected in these cases.

1.3 Static `CLASSPATH` vs. Runtime `CLASSPATH`

Misconception might be formed to believe `CLASSPATH` is only used for static class type searching, and is only used for static compilation. This is not true because Java classes are loaded lazily. At runtime, Java still needs a way to figure out

where to locate the classes. It is thus important the runtime virtual machine can also answer "where are the modules?" correctly.

JDK in fact keeps two `CLASSPATH`s, one for static compilation, and the other for virtual machine runtime lookup. The two `CLASSPATH`s are not necessarily the same. At compile time, one can use:

```
javac -classpath mypath1 HelloWorld.java
```

to specify where to find the class types for compiling `HelloWorld` class, and at runtime, one can use:

```
java -classpath mypath2 HelloWorld
```

to specify where to find the classes for lazy loading during the execution of `HelloWorld` class.

However, using different `CLASSPATH`s to compile and run the program might lead to problems: the program might typecheck under one definition of the class types, but at runtime, the real classes provided might be different. It is the task of the bytecode verfication to ensure the two are indeed consistent. Normally, when `CLASSPATH` is set as an environment variable of the operating system, it is used for both situations.

1.4 System Class Paths vs. Application `CLASSPATH`

This has undergone some changes as JDK evolves. According to the most up-to-date design, application writers only need to specify in `CLASSPATH` the entries that will help search non-system classes. System classes are located by a different system class path. Users can modify it (but rarely need to do so) by using the `Xbootclasspath` option when running `javac` or `java`.

1.5 Are Java Modules Hierarchical?

The answer is, strangely perhaps, no. When we discuss whether Java has a module system supporting submodules, we need to take the logical view, not the physical view. It might be noted that the physical implementation of Java packages, *i.e.*, the one on file systems, follows the hierarchical structure. Since directories can contain directories, it seems to be natural to assume packages can also contain packages. This unfortunately is untrue.

In Java, suppose we have a `CLASSPATH` entry, say, `usr`, and the file system has the following structure:

- `usr`
  - `packa`
    - `ClassM.java` with package identifier **package** `packa`
    - `packb`
      - `ClassN.java` with package identifier **package** `packa.packb`

It might be enticing to believe this `CLASSPATH` entry helps locate one package `packa`, and `packb` is a subpackage of `packa`. However, the logical view of this `CLASSPATH` entry is instead two independent modules:

- module with identifier `packa`, with one feature `ClassM`.
- module with identifier `packa.packb`, with one feature `ClassN`.

This also explains a seemingly strange issue about JDK. Some Java users might think: `CLASSPATH` is for programs to find classes, so the two forms should be equivlaent:

- `java -classpath usr packa.packb.ClassN`
- `java -classpath usr/packa packb.ClassN`

The truth is the second one will throw out a `CLassNotFoundException`. Why? Because this second line means to locate a package called `packb` under directory `usr/packa`, which obviously does not exist, since Java packages are not hierarchical. `CLASSPATH`s can only be pointed to root directories where packages are defined.

1.6 Java Modules Have No Runtime Representation

Java packages are stateless code pieces. They do not have runtime representations. At runtime, they together form a stateless repository for Java virtual machine to look up.

2 Static Time: `javac` Compilation

The compilation unit in Java is a class, not a package. This is different from major module systems, where compilers always set modules as the unit of compilation. This is reflected by the way `javac` tool works. Its mandatory argument is a file name storing the source code of a class, not a whole package. Of course, Java does not have separate compilation property, as compiling a class needs type information of classes used by this class, which can be thought of as a cross-module name reference and are stored in other packages.

While compiling a class, whether a class name symbolic reference will lead to the successful retrieval of its class type type is determined by two issues:

- Whether the compiler can find the class definition.
- If found, whether the qualifier of the target class/method/field allow access. The inaccessible parts (methods, fields, or the whole class) can be equivalently thought of as filtered out when the class type is retrieved.

2.1 Class Type Searching

When `javac` compiles a class, it might contain some class name symbolic references to other classes. Although the symbolic references could either be in the fully qualified form (a package name followed with a class name), or simply a class name form, they are eventually always equivlent to the fully qualified form. This is because, the simple class name form, say `ClassA`, means one of the following things:

- If there is an **import** directive at the top of the class definition, such as **import** `java.lang.*`, the `ClassA` could mean `java.lang.ClassA`; or
- the `ClassA` could mean **default**`.ClassA`.

This equivalent view that every class name symbolic reference in a class definition is actually of the fully qualified form can set a parallel to major module systems. For instance, it is very similar to the dot notation used in ML modules where nested modules are supported.

A successful type search may find a class file, a source file, or both. Here is how `javac` handles each situation:

- Search finds a class file but no source file: `javac` uses the class file.
- Search finds a source file but no class file: `javac` compiles the source file and uses the resulting class file.
- Search finds both a source file and a class file: `javac` determines whether the class file is out of date. If the class file is out of date, `javac` recompiles the source file and uses the updated class file. Otherwise, `javac` just uses the class file.`javac` considers a class file out of date only if it is older than the source file.

Please note that when using a qualified class name, the package name part should always be complete. For instance, it is wrong to have **import** `java.`$*$ as the starting directive, and `lang.String` as the class name symbolic reference in the middle of the program. The **import** part must denote a package name.

2.2 Java Access Control with Modifiers

It is naturally stratified into three levels: packages, classes, and methods/fields.

- Whether a package can be accessed depends on the host system. It is a directory anyway in JDK implementation.
- Whether a class in a package can be accessed has two cases: 1) If the class is qualified with `public`, then it can be accessed by anyone, including cross-package references. 2) If the class is not, then it can only be accessed by other classes defined in the same package.
- Whether a method or a field in a class can be accessed has three cases: 1) If it is qualified with `public`, then it can be accessed by anyone, including cross-package references of course. 2) if it is qualified with `protected`, then it can only be accessed by other classes in the same package, or by classes which is a subclass of the current one. 3) if qualified with `private`, then it can only be accessed by the class itself.

2.3 Static Class Type Equality

Class equality at static time is determined by fully qualifed class name, the simple string-form class name following fully qualified package names. If two classes are of the same package name and the same class name, the two class types are equal. Note that this should not include `CLASSPATH`.

In `.class` files, *i.e.* the bytecode representation, every class name symbolic reference is also in the fully qualified form. **import** directive does not have a bytecode representation.

3 At Runtime: Lazy Class Loading with Classloaders

3.1 Overview

At runtime, classes are loaded lazily, *i.e.*, if there is a class name symbolic reference, say, `ClassA`, inside a class definition of, say `ClassB`, the code of `ClassA` is not loaded until the expression containing the symbolic reference `ClassA` is evaluated at runtime. The loading task is started by a classloader which also loads `ClassB`, and is called the *initiating classloader* of `ClassA`. This classloader can have arbitary program logic defined inside, and in many situations, it might call for another classloader to do the loading. This process is called *classloader delegation*, and the one eventually loading `ClassA` is called the *defining classloader* of `ClassA`.

At runtime, each class must have one and only one defining classloader, and the defining classloader itself as a class must also be loaded by some other classloaders. It is not hard to see this forms a tree structure, where non-leaf nodes are defining classloaders, and leaves are classes. Note that here the tree in concern does not represent delegation relationship, but the relationship between classes and their defining classloaders and between defining classloaders. Next we explain what the root node of the tree should be.

3.2 Bootstraping

Here the central question is: How is a typical Java application bootstraped? When a user types in `java HelloWorld` in JDK:

- A *primordial classloader* (or called a *system classloader*, a *boostraping classloader*, a *null classloader*) is triggered. In JDK implementation, it is not a real Java class; instead, it is a piece of code written in native languages such as C/C++. The primordial classloader as the root of the delegation tree takes care of the lazy loading of system classes (System classes are not loaded when the primordial classloader is loaded; instead, they are loaded only when user programs containing them are evaluated). System classes, according to the design of the primordial classloader, include those in the following directories/JAR files:
  - `%/lib/rt.jar`

    - %/lib/i18n.jar
    - %/lib/sunrsasign.jar
    - %/lib/jsse.jar
    - %/lib/jce.jar
    - %/lib/charsets.jar
    - %/classes
- The primordial classloader loads in a class called `ExtClassLoader`. This is a JDK internal class, not for the use of Java programmers. It is a subclass of `java.net.URLClassLoader`. This classloader takes care of the lazy loading of extension classes residing in the directory `%/lib/ext`. It includes classes such as for DNS name space management, or LDAP management.
- An instance of the `ExtClassLoader` loads in a class called `AppClassLoader`. This is also a JDK internal class, not for the use of Java programmers. It is also a subclass of `java.net.URLClassLoader`. This is the classloader taking care of the lazy loading of application classes.
- An instance of the `AppClassLoader` then loads in the class `HelloWorld`.

Inside the `HelloWorld` class, if a symbolic reference to `java.lang.String` is present, the task of lazily loading this class at runtime will first be given to `AppClassLoader`, but since `java.lang.String` is a system class, the program logic of `AppClassLoader` delegates the task to its parent, `ExtClassLoader`, which will in turn delegate the loading to the primordial classloader. If class `HelloWorld` contains an extension class, the defining classloader will be the `ExtClassLoader`. If `HelloWorld` contains a symbolic reference to another user defined class, the defining classloader is `AppClassLoader`.

3.3 Custom Classloader Use

The primordial classloader, the `ExtClassLoader`, the `AppClassLoader` are all pre-defined classloaders in JDK. JDK users can also define their own custom classloaders, and use them to load other classes. If `HelloWorld` needs to load some class via some custom classloader, it can first create an instance of this classloader, and invoke the `loadClass` method of the instance, feeding it with the name of the class it intends to load. The loaded class will thus have the custom classloader as its defining classloader. A typical example appears as follows:

```
ClassLoader loader = new MyClassLoader();
Class c = loader.loadClass(‘‘MyClass’’);
```

Associating a class with a custom classloader is a task of programmers.

3.4 Custom Classloader Writing

We now discuss the other side of the issue related to custom classloders: how they are developed. Any user-defined classloader should directly or indirectly be a subclass of `java.lang.ClassLoader`. In Java programming prior to JDK 1.2, the customization job is mostly reflected in overriding the `loadClass` method of

`java.lang.ClassLoader`. Programmers can define their own loading protocols, any crazy things basically. Of course, this needs to be restricted by Security Managers, but the classloader as a mechanism itself does not subject to any syntactical or semantic restriction. The `loadClass` method has the signature:

```
protected synchronized Class loadClass(String clsn, boolean resolve)
    throws ClassNotFoundException
```

where the first argument indicates the name of the class to be loaded, and the second flag indicates whether symbolic references inside the class should be resolved at the time class named `clsn` is loaded; if yes, it is equivalent to eager class loading. A typical `loadClass` method looks like this:

- Call `findLoadedClass` to check if the class has already been loaded. If yes, the method immediately returns the loaded class.
- If the current class loader has a specified delegation parent, call the `loadClass` method of the parent to load the class. Otherwise (which means the class loader is the primordial classloader), call `findSystemClass` method to see whether the class can be found among system classes. If in either of the two aforementioned ways the class is loaded, the method returns.
- Otherwise, the current classloader defines its own way to find the class. After the class is read in as a byte stream, use `defineClass` method to construct a `Class` object out of the stream, and use `resolveClass` to resolve all symbolic references if eager name resolution is expected.

The problem with this JDK1.0 and JDK1.1 approach is, classloader delegation becomes a behavior that is totally subject to the program logic of the `loadClass`. Normally it should be true that parent classloaders are called first to load the class before the current classloader decides to make up a class itself, but this can not be ensured according to this design. A careless classloader writer could even forget to call parent classloaders entirely, and lead to many system classes unable to find.

To solve this, in JDK1.2 and later versions, programmers are not recommended to override `loadClass` anymore (this is still possible, but it is more for backward compatibility). Instead, `loadClass` method is now changed to the following form, and it is fixed so that programmers normally should not override it:

- Call `findLoadedClass` to check if the class has already been loaded. If yes, the method immediately returns the loaded class.
- If the current class loader has a specified delegation parent, call the `loadClass` method of the parent to load the class. Otherwise (which means the class loader is the primordial classloader), call `findSystemClass` method to see whether the class can be found among system classes. If in either of the two aforementioned ways the class is loaded, the method returns.
- Otherwise, call `findClass` method.

Note that the first two parts are unchanged. Only when it is to the last step, the `findClass` method is called. Now `findClass` becomes the new method JDK encourages programmers to override. The merit of this approach is custom classloaders can now only take care of the classes it intend to load. All delegation tasks are taken care of by the fixed program logic `java.lang.ClassLoader` itself.

3.5 Type Safety: The Compiler `javac` is Not Enough

Because of lazy loading, `.class` files may have been modified by the time lazy loading happens at runtime. Although typechecking process has already been done when source code is converted to `.class` format via `javac`, it is still necessary to do the typechecking once again on `.class` files when the classfile is loaded dynamically at runtime. This process actually has the more familiar name *bytecode verification*. Note that in theory bytecode verficiation is not only necessary when the code is downloaded from some other site; in terms of type safety, it is necessary for any dynamically loaded class.

Note that such a procedure is inherent in any lazy loading system, and is not the real problem of Java classloaders.

3.6 Type Safety: The Major Problem

What is more troublesome is at runtime, a class type is determined by a pair, the *class name* and its *defining classloader*. At static time, we have shown type equality of two classes are based on whether the fully qualified names of the two classes are equal: if they are from the same package with the same name, then they are equal. At runtime however, even if two classes have the same fully qualified names at static time, if they are loaded at runtime by different classloaders, they are still two different types.

3.7 Constraint Solving for Type Safety

Since what is the defining classloader for any given class name symobolic reference is not a property that can be determined at static time, Java relies on a runtime contraint solving system to preserve type equality. We now use pair $\langle C, L\rangle$ to represent a class named C with a defining classloader L:

- If class $\langle C1, L1\rangle$ references a field T *fn* declared in class $\langle C2, L2\rangle$, then $L1$ and $L2$ must eventually delegate the loading of T to the same classloader.
- If class $\langle C1, L1\rangle$ references a method (or constructor) $T0$ $mn(T1, \ldots, Tk)$ declared in class $\langle C2, L2\rangle$, then $L1$ and $L2$ must eventually delegate the loading of $T1$ (or $T2 \ldots, Tk$) to the same classloader.
- If class $\langle C1, L1\rangle$ overrides a method $T0$ $mn(T1, \ldots, Tk)$ declared in class $\langle C2, L2\rangle$, then $L1$ and $L2$ must eventually delegate the loading of $T1$ (or $T2 \ldots, Tk$) to the same classloader.

– If class $\langle C1, L1\rangle$ upcasts an object of class $\langle C2, L2\rangle$ to type T, then $L1$ and $L2$ must eventually delegate the loading of T to the same classloader. However, whether this constraint must be satisfied depends on how Java handles upcasting. If when it typechecks upcasting cases the same as it typechecks random casting, then a rule like this will not be needed, because even if the $L1$ and $L2$ delegate the loading of T to different classloaders, this simply means it is a random casting.

Basically whenever the typechecker by fully qualified names assumes some types are the same, at runtime they need indeed be the same.

4 Weaknesses

This section is a list of issues identified by other papers as Java modularity mechanisms' weaknesses. Technically, they can be naturally drawn from this summary, but some seems to be cosmetic, and some even depends on how one defines "weakness". The list might be very incomplete. It might grow as more complete comparisons with other modules systems are done.

– Java is lack of static checking. `javac` does do static type checking, but since the `CLASSPATH` used to statically compile a class is not necessarily compatible with the runtime `ClASSPATH` when the class is lazily loaded, all type checking tasks in theory needs to be redone.
– Modules only have exports, no imports. Parametricity is not supported, even on the value level (*i.e.* an exporting class depends on a parametric import class), not to mention on the type level.
– Classloader delegation is not a part of the module system. Dynamic linking is excluded from module system.
– Classloader delegation might form cycles.
– Packages can not contain packages.
– Packages do not support selective hiding, such as declaring a class is visible to some packages, but not others. Any classes defined as `public` in a package is automatically visible to all packages.
– Can not expose different interfaces to different clients. Not adequately support abstract datatypes.