JAVA CLASS LOADING IN COMEBACK!

MICHAEL RUDOLF

VERSION 0.4, 2009–04–13

Class loading in Java is not a trivial issue, and creating custom class loaders is a delicate endeavour, where subtle mistakes can have an enormous impact on functionality. In our ComeBack![1] project we use the ASM[2] library to generate a custom class loader. This document will describe the reason for that and the inner workings of this class loader.

[1] `http://comeback.sourceforge.net/`

[2] `http://asm.objectweb.org/`

Version History

Version	Date	Changes
0.1	2007–12–08	Initial version
0.2	2008–06–23	Changed formatting
0.3	2008–07–09	Improved figures
0.4	2009–04–13	Described class sharing

Introduction

Usually, when creating a small-scale application in Java, developers do not get into contact with class loading intricacies. However, even with a simple program there is a complete infrastructure underneath, making sure that it runs at all. Whenever a Java application is about to be started, the Java Virtual Machine prepares a class loader hierarchy similar to the one shown in Fig. 1. At the top there is the so-called *bootstrap* class loader, reigning over all classes in the standard library, which are assembled in the file `rt.jar` in the `lib` subdirectory of the Java installation directory. Beneath it is the *extension* class loader, which provides access to all classes in Java platform extensions. Finally, the *system* class loader will load classes from the classpath as specified when starting the Java application.

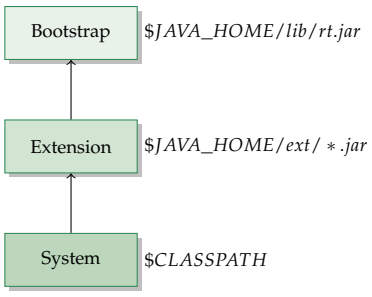

Figure 1: Default class loader structure and their class locations

Every class loader, except the bootstrap class loader, has a parent class loader associated with it. This relationship is shown by solid black arrows in the figures. Whenever a class is requested from a class loader, it first checks whether it already has been loaded by calling the method `findLoadedClass`. This method performs a lookup in a local cache, which contains all classes defined by the class loader. If the requested class is not in the cache, the class loader asks its parent class loader to load it. Only if the parent class loader is not able to load the class, the class loader itself tries to load it. This mechanism ensures that classes are loaded by the class loader nearest to the root of the tree and that class loaders only load those classes they are responsible for. If this *parent-first* approach is not used, the same class could be loaded by multiple class loaders resulting in different objects representing it. This is explained in the section 5.3,

Creation and Loading, of the Java Language Specification:[3]

> "At run time, a class or interface is determined not by its name alone, but by a pair: its fully qualified name and its defining class loader. Each such class or interface belongs to a single runtime package. The runtime package of a class or interface is determined by the package name and defining class loader of the class or interface."

[3] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, April 1999.

As a consequence, casts and assignments would fail.

Our approach at class loading depends on how the application is started, or, to be more precise, by which party. The following two sections describe the scenarios of application startup and our corresponding class loading strategy.

Plugin-Initiated Application Startup

When there is only one plugin for the framework and that plugin contains the code to launch the application, one usually does not call it "plugin" — although that is correct — but *extension*. This is because the word "plugin" conveys the image of potentially many different pieces of code being plugged into the framework, the latter also being able to fulfill its purpose without any plugins at all. This first scenario, however, describes a situation, in which the framework is no complete application by itself and can only be used by means of an extension (hereinafter referred to as "plugin") providing all the missing pieces. A simple example shall illustrate that: an application that consists of a single Swing window is usually not thought of as a plugin to the Swing framework, although this is exactly what it is. The plugin part contains the application's main method and instantiates Swing classes to construct the window.

In this scenario the plugin provides the value of the `$CLASSPATH` process environment variable, which is used for creating the system class loader. A framework update has to change the contents of that class path, so that it does not point to the framework anymore, but to the adapter layer created by our ComeBack! tool instead. However, we do not change the application startup mechanism, we just replace files on disk. As a result, the plugin will load adapter classes after the upgrade whenever it loaded framework classes before.

Now that the updated framework is no longer part of the plugin's class path, our adapter layer has to account for that by using a dedicated mechanism for loading framework classes. This is achieved by generating a special class loader as part of the adapter layer, which is used in conjunction with reflection for accessing the framework, as shown in Figure 2. This class loader provided by the ComeBack! tool intentionally violates the principle of parent-first loading described in the introduction. After checking its local cache, it first tries to load the requested class itself, and only if that fails it turns to its parent class loader and delegates the request up the hierarchy. The workhorse

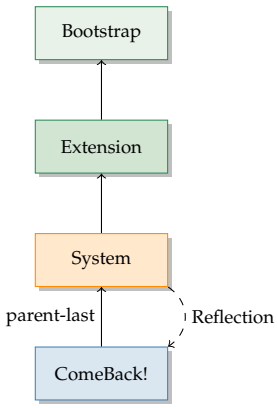

Figure 2: Plugin starts the framework

method `loadClass` is shown in the following listing; notice the exchange of lines 18 and 21 with respect to the implementation provided in the superclass `java.lang.ClassLoader`.

Although it might be as simple as in the example above, the plugin can of course also use a dedicated class loading mechanism. However, this does not affect our approach, because our generated framework class loader will still be found by the adapter layer. Figure 3 shows the class loader hierarchy for this case; the plugin class loader (the one handling the class path) is colored orange and the special framework class loader is shown in blue.

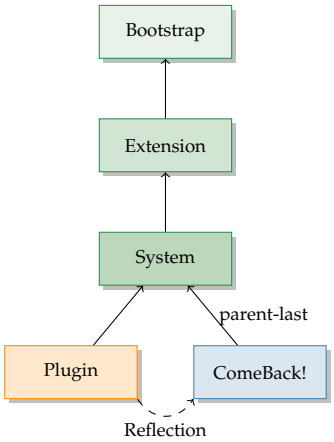

Figure 3: Plugin starts the framework using a special class loader

CLASS SHARING IS REQUIRED for all administrative (i.e., non-adapter) classes of the adapter layer, e.g., the adapter cache, the lookup facility, and the dynamic adapter generator. Therefore, the domains of the system (or custom plugin) class loader and the special ComeBack! class loader have to overlap. This is ensured by having the ComeBack! class loader check whether the name of the requested class starts with a known shared substring. If this is the case, the class loader that loaded the ComeBack! loader will be consulted first. As a consequence, all adapter layer classes needed by both the system (or custom plugin) class loader and the special ComeBack! class loader will be loaded only once and can be accessed from both class loading domains. The corresponding code is shown in the lines 5–14 in the listing below.

```
1  protected synchronized Class<?> loadClass(String name,
2          boolean resolve) throws ClassNotFoundException {
3      //check whether the class has been loaded already
4      Class<?> c = findLoadedClass(name);
5      if (c == null) {
6          for (int i = 0; i < sharedPackages.length; i++) {
7              //the class might be shared
8              if (name.startsWith(sharedPackages[i])) {
9                  c = getClass().getClassLoader().loadClass(
10                         name);
11                 break;
12             }
13         }
14     }
15     if (c == null) {
16         try {
17             //we load the class ourselves
18             c = findClass(name);
19         } catch (ClassNotFoundException ex) {
20             //we delegate to our parent class loader
21             c = getParent().loadClass(name, false);
22         }
23     }
24     if (resolve) {
25         //load all dependent classes
26         resolveClass(c);
27     }
28     return c;
29 }
```

Framework-Initiated Application Startup

Many application frameworks are designed in a way, that their basic functionality can be extended with plugins. The popular Java development environments Eclipse[4] and NetBeans[5] are prime examples of this architecture. Here, the framework constitutes a fully-functional application on its own, and it loads the plugins either as part of the startup process or later on demand. Some frameworks even support so-called *hot swapping*, where plugins can be replaced by a newer version at runtime. Naturally this requires a dedicated class loading infrastructure, which will not work together with the ComeBack! class loading approach described in the previous section. Therefore, framework developers need to account for some things, if they want to enable seamless framework updates with the help of our ComeBack! tool. In the following we describe three class loading strategies commonly encountered in this scenario and how our approach to integrating adapter layers looks like.

[4] http://www.eclipse.org/

[5] http://www.netbeans.org/

No dedicated plugin class loader

In the simplest case the framework does not use any dedicated class loader for plugins at all. Instead, all the plugins are put in the class path together with the framework (e.g., by means of a batch or shell script), so that they will be loaded by the system class loader. Unfortunately there is no way to introduce adapter layers into the class path without changing the startup mechanism (i.e., rewriting the mentioned scripts), which is not feasible. Therefore, we do not support this kind of architecture. Framework developers need to separate their framework from the plugins by at least one dedicated class loader, as discussed in the next section.

Single plugin class loader

If plugins are loaded using a dedicated class loader (e.g., fetching plugin classes from JAR files in a special "plugin" directory), our ComeBack! tool can be used to generate a single adapter layer for all plugins. However, this implies that all plugins will see the same framework version, which is reified by the adapter layer. In order for plugins to access the adapter layer, the plugin class loader created by the framework needs to be modified to first look for classes in the generated adapter layer before delegating requests to its parent (the system class loader, which is responsible for loading framework classes from the class path). If modifying the plugin class loader is no option, then the framework developers can also insert a special class loader in between the plugin class loader and the system class loader, which does not adhere to the parent-first principle similar to the one described in the section *Plugin-Initiated Application Startup*. This implementation is shown in Fig. 4, where the plugin class domain is highlighted in yellow, the adapter layer in red, and the framework

domain in blue. Notice that separating plugin and adapter classes with different class loaders, we consistently separated out the colors yellow and red from what was orange in Figs. 2 and 3.

One class loader per plugin

The highest degree of flexibility can be reached by loading each plugin in a dedicated class loader. This permits each plugin to have its own adapter layer thereby allowing for different plugins seeing different framework versions. Again, each plugin class loader needs to be retrofitted to first load adapter classes before delegating to its parent, or a special class loader must be inserted into the hierarchy. Figure 5 depicts that class loader architecture using the same color scheme as in the previous section.

Colophon

This document was typeset with the free, cross-platform LaTeX typesetting system using the `tufte-handout` package,[6] version 2.01, which simulates the layout style espoused by visual design expert Edward Rolf Tufte.

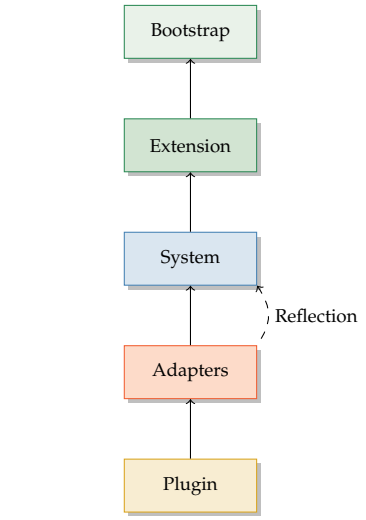

Figure 4: The frameworks starts the only plugin

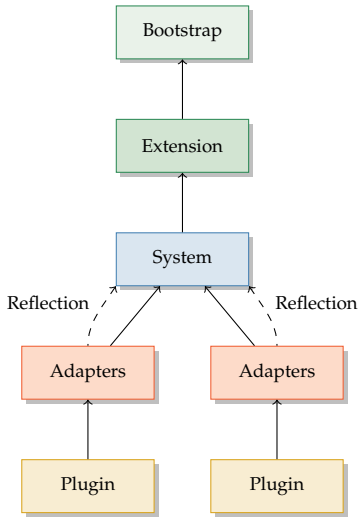

Figure 5: The frameworks starts all the plugins

[6] `http://code.google.com/p/tufte-latex/`