

Dynamic Class Loading

Philippe Collet

Partially based on notes from Michel Buffa

Master 1 IFI International
2013-2014

<http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1314>

Agenda

- Principle of a ClassLoader
- Writing a *custom* ClassLoader
- Architecture to load plugins
- How to reload plugins

Definition of a ClassLoader

- The JVM contains a ClassLoader
- ClassLoaders allows for loading classes from the filesystem, but also from multiple locations (DB, network, etc.)
- Role: convert a class name in an array of bytes representing a class

```
Class c = loadClass(String className, boolean  
                    resolveIt);
```

On ClassLoaders

- All JVMs have a « system » ClassLoader
- This default CL implements a method `loadClass()` that searches in the CLASSPATH for `.class`, `.jar` or `.zip` files
- One can create new CL by deriving from class `ClassLoader` and by redefining `loadClass()` and the methods it uses (`findClassFromClass`, `ResolveClass...`)

Time of class loading

- It depends!
- In general when:
 - `Foo f = new Foo();`
 - Static references such as:
 - `System.out,`
 - `Foo.class,`
 - `forName("Foo");`

Writing your own ClassLoader ?

- Interest?
 - Load classes from the web
 - Load classes from a DB
 - Load classes « differently »
- **Don't work with applets!**
- From JDK1.2 on, an URLClassLoader is provided

Steps for writing a ClassLoader

- Sub-class `ClassLoader` and implement the method `loadClass`
 1. Check the class name, determining whether it has already been loaded
 2. Check whether this is a system class
 3. Attempt to load it
 4. Define the class for the VM
 5. Resolve it by loading its dependencies
 6. Return the class
- From JDK1.3 on: sub-class `SecureClassLoader` to respect recommendations relative to the new Java security policy

Example of *custom* ClassLoader

```

public synchronized Class loadClass(String className, boolean resolveIt) throws
    ClassNotFoundException {
    Class result;
    Byte []classData;

    // 1) search in the cache whether the class has already been loaded
    result = (Class) classes.get(className);
    if(result != null)
        return result;

    // 2) find out whether this is a system class. Very important because one
    // could change the security manager!
    try {
        result = super.findSystemClass(className);
        return result;
    } catch(ClassNotFoundException e) {
        System.out.println("Pas une classe système !");
    }

    // 3) load the class from OUR REPOSITORY (e.g., web ou DB)
    classData = getImplFromDataBase(className);
    if(classData == null) { throw new ClassNotFoundException() }
  }

```

Example of *custom* ClassLoader (cont'd)

```

...
// 4) Define the class (by parsing). Actually the method will check the
// validity of Bytecode and make other verifications
// The result is stored in a specific data structures within the JVM
result = defineClass(classData, 0, classData.length);

// 5) Resolve class, i.e. apply the same process to superclasses and dependent
// classes.
if(resolveIt)
    resolveClass(result);

// Before returning the final result, put the class into the cache
classes.put(className, result);

// 6) return result
Return result;
}

```

Example of a code that reads a class

```
...  
byte [] result;  
try {  
    FIS fis = new FIS("store\\ " + className + ".impl");  
    result = new byte[fis.available()];  
    fis.read(result);  
} catch(Exception e) {  
    return null;  
}
```

Example of a code that reads a class from the web

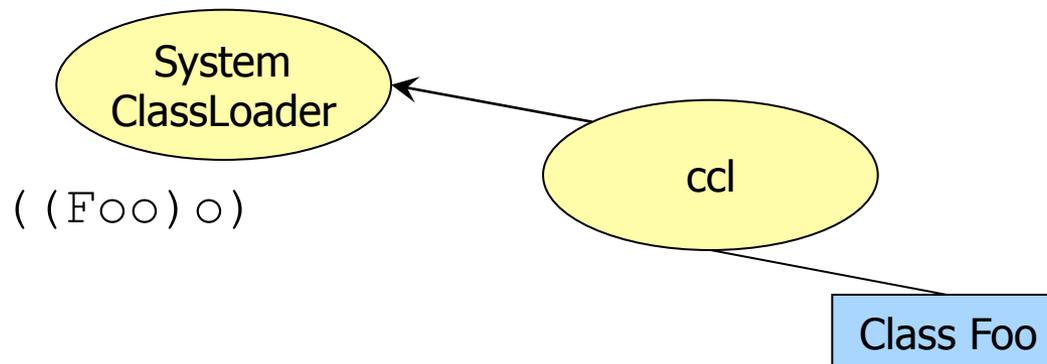
```
...  
URL url = new URL(urlClassName);  
URLConnection uc = new URLConnection(url);  
int length = uc.getContentLength();  
IS is = uc.getInputStream();  
byte []data = new byte[length];  
is.read(data);  
is.close();  
return data;  
...  

```

Using a *custom* ClassLoader

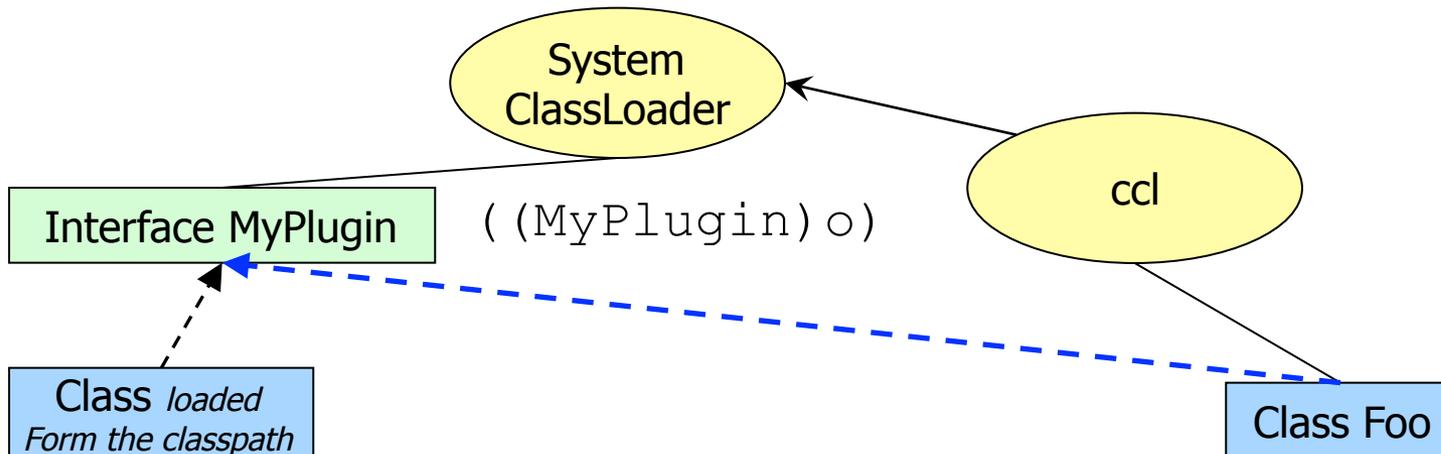
```
Class c = ccl.loadClass("Foo");  
Object o = c.newInstance();  
((Foo) o).f();
```

- Don't work, as only the new cl knows Foo!



The need for an interface

- Only the new CL knows Foo. If one wants to cast the object, a common Interface must be created and be known by the default (system) CL.
- For example, Plugin is known by the default CL (SystemClassLoader).



Writing some plugins

- Simple rule: read plugins from a directory
- Define a usage interface on plugins
- Each plugin must implement this interface
- Each class will be a plugin
- Then, one simply needs:
 1. Read the content of a directory
 2. For each class name in the plugins directory
 - `Class c = Class.forName(className);`
 - `Plugin p = (PluginDraw) c.newInstance();`
 - `p.draw(); // method present in Plugin.java`

Plugins again...

- Difficulties when
 - No default constructor
 - Plugins are in a .jar or in a DB
 - Look for ressources? Images, sounds, etc.
- `Class.forName(className)` is the equivalent of `loadClass(className)` that was studied for the custom `ClassLoader`
- Plugins are everywhere!!!
 - Photoshop, IE, Firefox, Chrome...

Plugins: ideal model

- Ideal model =
 - A directory for plugins
 - (.class files) and jar files in the directory
 - The application « discovers » files and « absorbs » plugins that are inside
- Each jar contains
 - A class that implements the interface Plugin.java
 - Other classes it needs (ie depends on)
 - Images, icons, sounds, docs, etc. that the plugin needs as well

Plugins: ideal model

- But there is a problem when too many classes are present in the plugin
 - How to test that a class is *really* a plugin without trying to load it, which is taking time
 - `Class.forName()` is long...
 - `Class.newInstance()` too...
- Eclipse, for example, contains more than 800 plugins that are loaded at startup
- Some plugins extend other plugins and thus depend on them
- The loading order is important for consistency...

How to manage plugins?

- The solution often uses a descriptor, a « map » of plugins
 - Eclipse follows a specific format to describe plugins, their dependencies (with some XML files)
- Problem #1 with plugins is the loading time
- But advantages are numerous...

Developing an application extensible by plugins

- A SDK to develop plugins must be provided
 - So that developers can compile and test plugins without the code of the main application
- What must be provided?
 - Necessary classes and interfaces
 - One or more exemple plugins, with a compiling aid (ant file for example)
 - Documentation, tutorials, etc.

Changing plugins without restarting the main application/server

- Principle of Servlet/Jsp servers:
 - Classes and jars are added
 - The server is not restarted
- Example
 - Java application servers (large majority of front-end/business tier on current linux based servers)
 - 24/7 exploitation (never stopped nor rebooted)
 - Plugins are dropped down and discovered on the fly...

Hot (Re-)Loading

- Principle : change the classloader for each plugin loading...
 - Each classloader manages its own « cache » of classes
 - If the classloader is reinstantiated, one can load/reload plugins at runtime
 - An URLClassLoader will be used for this implementation
 - One only needs to
 - re-scan every X (5 for example) seconds the plugin directory
 - re-scan on-demand (a menu for example)

