Java Class Loading

m hoehme <java@mhoehme.de>, 2007

Table of Contents

Introduction 1
Understanding class loading 1
A VersionClassLoader 7
References 17

Introduction

Class loading is one of the more advanced topics of the Java language. Normally a Java programmer does not need to think about how classes are made available to the Java Virtual Machine (JVM). However, there are situations when you might want to have better control over class loading, e.g. for making available different versions of a software module.

Before we are working out the details of a class loader that loads classes from a jar file, I want to delve into the mechanics of class loading.

Understanding class loading

The runtime system loads classes when they are needed. The command-line argument

```
-verbose:class
```

displays class loading information.

Running the following HelloWorld class with this argument loads a total of 249 classes:

Listing 1

```
public class HelloWorld {
   public static void main (String[] args) {
      System.out.println ("Hello, World!");
   }
}
```

Here is the shortened list of classes required for running the HelloWorld class:

Classes loaded for the HelloWorld.class

```
%> java -verbose:class HelloWorld
[Opened C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Opened C:\Programme\Java\j2re1.4.2_01\lib\sunrsasign.jar]
[Opened C:\Programme\Java\j2re1.4.2_01\lib\jsse.jar]
[Opened C:\Programme\Java\j2re1.4.2_01\lib\jce.jar]
[Opened C:\Programme\Java\j2re1.4.2_01\lib\charsets.jar]
[Loaded java.lang.Object from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.io.Serializable from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.Comparable from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.CharSequence from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.String from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.Class from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.Cloneable from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.ClassLoader from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.System from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.Throwable from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.Error from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.ThreadDeath from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.Exception from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.RuntimeException from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.security.ProtectionDomain from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.security.AccessControlContext from
C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
```

```
[Loaded java.lang.ClassNotFoundException from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
...
[Loaded java.io.FilePermission from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.io.FilePermission$1 from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.io.FilePermissionCollection from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.security.AllPermission from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.security.UnresolvedPermission from
C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.security.BasicPermissionCollection from
C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.security.Principal from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.security.cert.Certificate from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded HelloWorld]
Hello, World!
[Loaded java.lang.Shutdown from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
[Loaded java.lang.Shutdown$Lock from C:\Programme\Java\j2re1.4.2_01\lib\rt.jar]
```

The anatomy of a class loader

Class loading is performed by subclasses of `java.lang.ClassLoader`. As an abstract class, `java.lang.ClassLoader` cannot be instantiated directly. Because there are no abstract methods, we can use it as an anonymous subclass:

Listing 2

```
public class Test1 {
   public static void main (String[] args) throws Throwable {
      System.out.println("Class loader: "+Test1.class.getClassLoader());
      Class c = Class.forName ("HelloWorld", true, new ClassLoader() {});
      System.out.println (c+", class loader: "+c.getClassLoader());
   }
}
```

Every class has a reference to the class loader that loaded it. We obtain a reference to the class loader by calling getClassLoader() on the class under investigation:

Output of Test1

```
%> java Test1
Class loader: sun.misc.Launcher$AppClassLoader@e80a59
HelloWorld@194df86, class loader: sun.misc.Launcher$AppClassLoader@e80a59
```

We have instantiated a new class loader – but `HelloWorld.class` is loaded by the same class loader as `Test1.class`.

A closer look to the class loader's `loadClass()` method reveals what happens when we load a class:

java.lang.ClassLoader

```
   protected synchronized Class loadClass(String name, boolean resolve)
      throws ClassNotFoundException
   {
      // First, check if the class has already been loaded
      Class c = findLoadedClass(name);
      if (c == null) {
          try {
             if (parent != null) {
                c = parent.loadClass(name, false);
             } else {
                c = findBootstrapClass0(name);
             }
          } catch (ClassNotFoundException e) {
             // If still not found, then invoke findClass in order
             // to find the class.
             c = findClass(name);
          }
      }
```

```
        if (resolve) {
            resolveClass(c);
        }
        return c;
    }
```

Loading a class involves three steps:

1. The class loader looks at its internal map of loaded classes whether this class has already been loaded.
2. When the class is not with the map, the loading request is delegated to the class loader's parent class loader. If the parent is null, the request is delegated to the bootstrap class loader. The bootstrap class loader, however, loads only trusted system classes.
3. When the loading request has not yielded a result, the class loader itself tries to find the class by calling `findClass (name)`.

So, `HelloWorld.class` was in fact loaded by the parent class loader, and not by our anonymous subclass.

Let us modify the Test program. By passing null into the ClassLoader constructor, we nullify the parent class loader and thus force our anonymous subclass to load the `HelloWorld.class`. Since `HelloWorld.class` is not a trusted system class – such as `java.lang.Object` – the request cannot be successfully answered by the bootstrap class loader:

Listing 3

```
public class Test2 {
    public static void main (String[] args) throws Throwable {
        System.out.println("Class loader: "+Test2.class.getClassLoader());
        Class c = Class.forName ("HelloWorld", true, new ClassLoader(null) {});
        System.out.println ("Class HelloWorld, class loader: "+c.getClassLoader());
    }
}
```

The result is a `ClassNotFoundException`:

Output of Test2

```
%> java Test2
sun.misc.Launcher$AppClassLoader@e80a59
java.lang.ClassNotFoundException: HelloWorld
        at java.lang.ClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at de.mhoehme.classloading.Test.main(Test.java:19)
Exception in thread "main"
```

The `ClassNotFoundException` is thrown by `findClass()`. Checking the source code of `java.lang.ClassLoader` shows that it is a dummy implementation, intended to be overwritten by a subclass:

java.lang.ClassLoader

```
    protected Class findClass(String name) throws ClassNotFoundException {
        throw new ClassNotFoundException(name);
    }
```

We have already met `sun.misc.Launcher$AppClassLauncher`. The source code of this class it not available. So let's check the implementation of its parent class, `java.net.URLClassLoader`:

java.net.URLClassLoader

```
    protected Class findClass(final String name)
        throws ClassNotFoundException
    {
        try {
```

```
            return (Class)AccessController.doPrivileged(new PrivilegedExceptionAction() {
               public Object run() throws ClassNotFoundException {
                  String path = name.replace('.', '/').concat(".class");
                  Resource res = ucp.getResource(path, false);
                  if (res != null) {
                     try {
                        return defineClass(name, res);
                     } catch (IOException e) {
                        throw new ClassNotFoundException(name, e);
                     }
                  } else {
                    throw new ClassNotFoundException(name);
                  }
               }
            }, acc);
         } catch (java.security.PrivilegedActionException pae) {
            throw (ClassNotFoundException) pae.getException();
         }
      }
```

When we let aside the security checkings, the process of finding a class can be summarized as follows:

1. Getting the byte code of a class (wrapped by the `Resource` object)
2. Making it an executable Java class by calling `defineClass` on the byte code.

When we write our own class loader, the `findClass` method could be implemented in this way:

Listing 4

```
public Class findClass (String name) throws ClassNotFoundException {
   byte[] buf = getBytes(name);
   return defineClass (name, buf, 0, buf.length);
}
```

Armed with these insights, we can write our own `ClassLoader` subclass that attempts to find a class by itself before delegating the request to its parent:

Listing 5: MyClassLoader.java

```
import java.io.*;

public class MyClassLoader extends ClassLoader {
   protected synchronized Class loadClass (String name, boolean resolve)
      throws ClassNotFoundException
   {
       // #1 - check loaded classes
      Class cl = findLoadedClass(name);
      // #2 - enforce findClass call
      if (cl == null) {
         cl = findClass (name);
      }
      // #3 - else ask the parent
      if (cl == null && getParent() != null) {
         cl = getParent().loadClass (name);
      }
      if (cl == null) {
         throw new ClassNotFoundException (name);
      }
      if (resolve) {
         resolveClass (cl);
      }
      return cl;
   }

   protected Class findClass (String name) throws ClassCastException {
      byte[] b = getBytes (name);
      if (b != null) {
```

```
         return defineClass(name, b, 0, b.length);
      }
      return null;
   }

   private byte[] getBytes (String name) {
      String classname = name.replace('.', '/').concat(".class");
      byte[] b = null;
      try {
         File f = new File (classname);
         if (f.exists()) {
            b = new byte[(int) f.length()];
            BufferedInputStream in = new BufferedInputStream (
                     new FileInputStream (f));
            in.read(b);
            in.close();
         }
      } catch (IOException e) {
         // will return null
      }
      return b;
   }
}
```

This class loader will be tested by `Test4.java`:

Listing 6: Testing MyClassLoader

```
public class Test3 {
   public static void main(String[] args) throws Throwable {
      System.out.println (Test3.class.getClassLoader());
      Class c1 = Class.forName ("HelloWorld", true, new MyClassLoader());
      System.out.println (c1+", class loader: "+c1.getClassLoader());
      Class c2 = Class.forName ("HelloWorld", true, new MyClassLoader());
      System.out.println (c2+", class loader: "+c2.getClassLoader());
      System.out.println ("c1 == c2: "+ (c1 == c2));
   }
}
```

Output of Listing 6

```
%> java Test3
sun.misc.Launcher$AppClassLoader@e80a59
class HelloWorld, class loader: MyClassLoader@eee36c
class HelloWorld, class loader: MyClassLoader@bd0108
c1 == c2: false
```

The output shows that we have loaded the HelloWorld class with two different class loader instances.

Class loader namespacing

Class loaders also have namespace capabilites. In the previous example, we have enforced that the class is loaded from disk by the class loader instance passed to `Class.forName()`. We have loaded HelloWorld.class with two different instances of our class loader.

Java sees classes loaded by different class loaders as completely different classes – even if they have the same name and the same package. That is why the comparison of `c1` and `c2` in `Test3.java` returns false.

By investigating the loadClass() method, we have seen that Java class loaders always try to delegated the loading request to their parents first. This strategy ensures that as many classes as possible are loaded by a higher-level class loader.

However, there are situations when this default strategy is not appropriate. J2EE containers, for example, use their own class loaders. Not only applications, but also each JSP instance has a class

loader instance of its own. The custom class loaders avoid different applications running in the same container to interfere with each other.[1]

The class loader hierarchy

Java has a hierarchy of class loaders. This can be queried by getting the parent class loader:

```
public class TestParents {
   public static void main (String[] args) {
      ClassLoader cl = TestParents.class.getClassLoader();
      System.out.println (cl);
      while (cl != null) {
         cl = cl.getParent();
         System.out.println (cl);
      }
   }
}
```

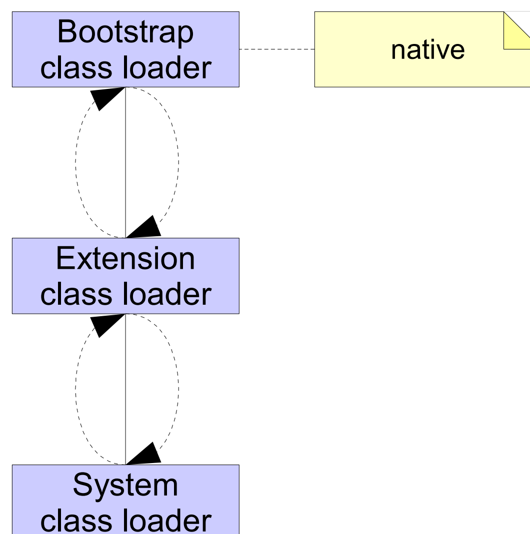
`TestParents.java` displays the hierarchy of three class loaders, with the top-most class loader – the bootstrap class loader – not being accessible:

```
%> java TestParents
sun.misc.Launcher$AppClassLoader@e80a59
sun.misc.Launcher$ExtClassLoader@1ff5ea7
null
```

The bootstrap class loader loads the trusted system classes.

The extension class loader is responsible for loading the classes contained in the Java extension libraries.

Finally the application class loader (or system class loader) loads the application classes.



Java class loaders use a delegation system: Before an application's system class loader attempts to load a class, it delegates the request to its parent class loader. When the extension class loader cannot find the requested class, it delegates the request to the bootstrap class loader. When the bootstrap classloader, too, cannot return the class, the system class loader executes its findClass() method.

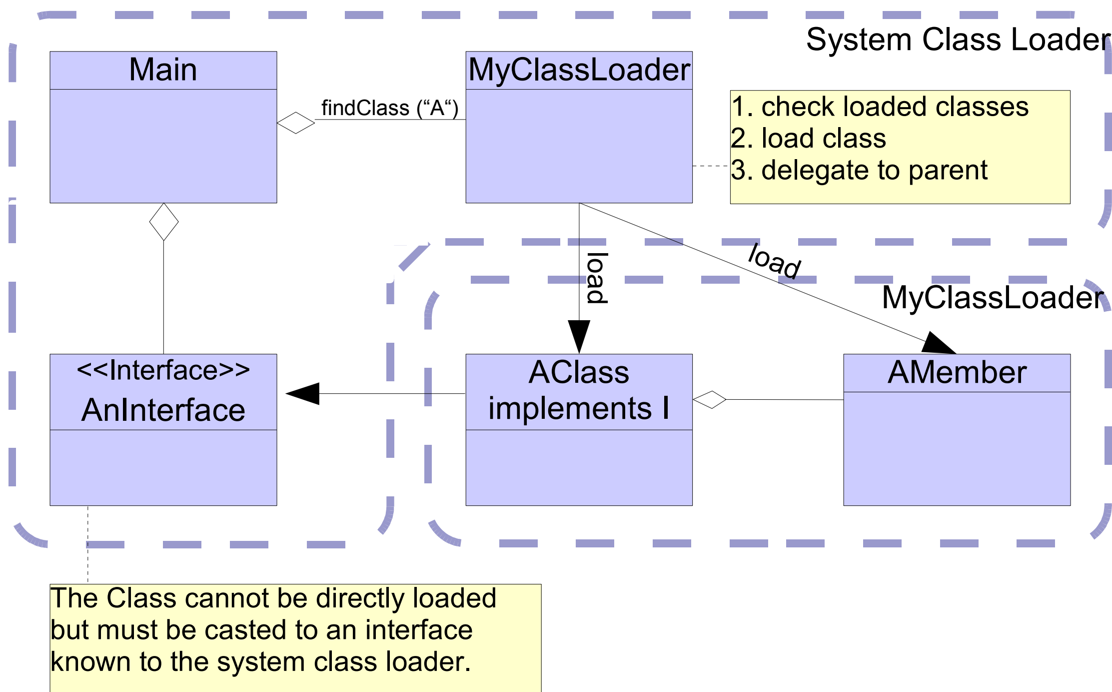
Class loader scope

A class is „owned“ by a class loader. When a class references another class, the class's class loader

1 See Tomcat documentation for details on the class loading concept.

is asked to load the class of the member. The class loader follows the path we have investigated above: If it does not own the class, it asks its parent, and so on.

In MyClassLoader.java we have broken up this class loading strategy and enforced loading the class by the specified class loader. As a consequence, member classes are also loaded by this class loader:



The result is a type problem: The only way to ensure that AClass can be recognized by Main is to cast it to an interface known to the system class loader.

The class format

When we think of classes, we normally expect files with the extension .class. However, for the process of finding a class, this is not important. The essential point is whether the obtained sequence of byte conforms to the specification of the class format or not. A class loader may read the byte code from disk as well as compile it on the fly.

Summary

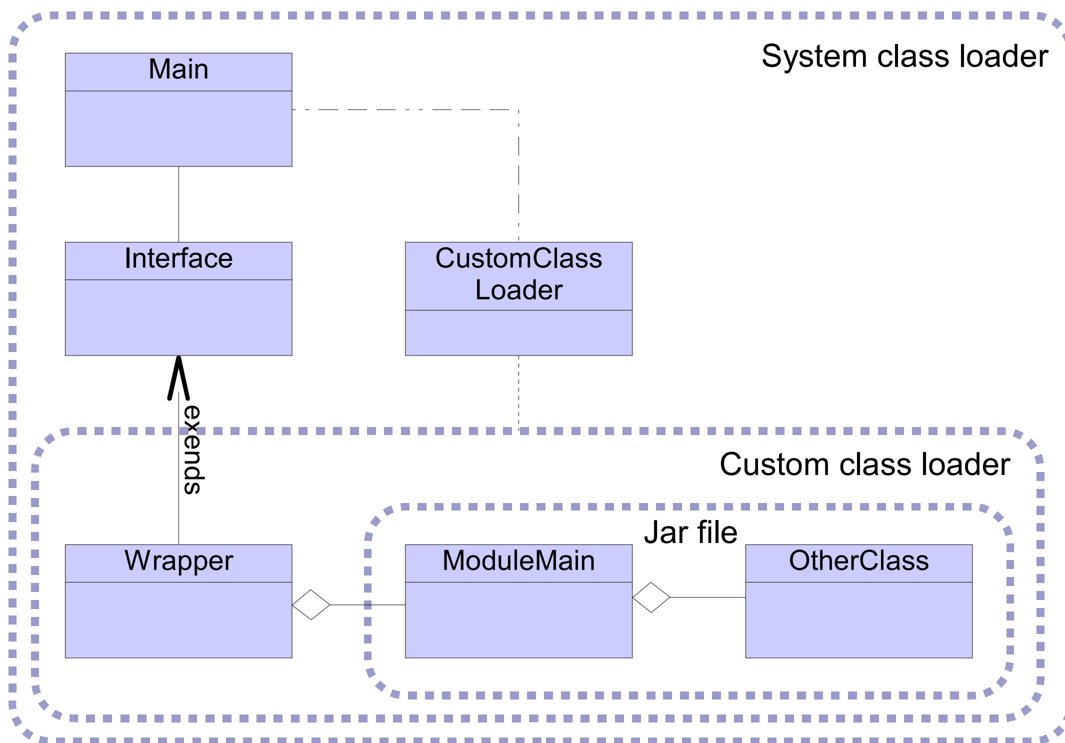
We have looked at the process of loading and of finding a class. Loading a class involves checking the map of loaded classes and asking the parent to load the class. If these attempts have not loaded the class, the class loader instances finds the class on its own by obtaining the byte code and by making it executable to the JVM.

A VersionClassLoader

Suppose we have a jar file containing a number of class files. This software module exists in several versions. Each version has the same main class. In order to be able to load different versions on demand, we need a custom class loader.

As we have seen, class loaders also act as a namespace. Since the main class is not accessible to us, we put a wrapper class inbetween. This wrapper class implements an interface to which it can be casted by the calling program. The wrapper class has one member, which is the module's main class.

The wrapper class does nothing else but forwarding all calls to its member.



The following code provides a rather general custom class loader to accomplish the task. Depending on the circumstances of its employment, it can be embellished. For instance, this VersionClassLoader does not perform any security checking.

VersionClassLoader.java

```
package de.mhoehme.versionclassloader;

import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;
import java.util.jar.JarFile;
import java.util.zip.ZipEntry;


/**
 * <P>
 * Custom class loader to load a specific version of a library.
 * The class loader requires two parameters:
 * <OL>
 * <LI>The classpath for the versioned classes, e.g. a jar file
 *     such as <CODE>myjar-0.1.jar</CODE>.
 *     Classes that could not be found in this classpath are
 *     loaded by the parent classloader.</LI>
 * <LI>A <CODE>WrapperIdentifier</CODE> that tells the class loader
 *     which classes to load from the VM's classpath</LI>
 * </OL>
 * </P>
 *
 * <P>
 * When the system property <CODE>de.mhoehme.versionclassloader.verbose</CODE>
 * is set to <CODE>true</CODE>, additional output on the loading location
 * is provided at System.out.
```

```
 * </P>
 *
 * <H3>A typical scenario</H3>
 * <P>
 * There are several versions of a class. These versions do not implement an
 * interface, and you cannot change the source code. The following HelloWorld
 * code serves as an example to demonstrate how this VersionClassLoader can be
 * used to load different versions of the same components.
 * </P>
 *
 * <PRE>
 *
 * class Hello {
 *
 *    public void hello() {
 *       System.out.println("Hello, World! (Version <I>n</I>)");
 *    }
 * }
 *
 * </PRE>
 *
 *
 * <H4>Step 1: Create an interface to be loaded by the system class loader</H4>
 * <P>
 * This interface has to be loaded by the application's class loader (a.k.a.
 * system class loader). It <I>must</I> not be loaded by the
<CODE>VersionClassLoader</CODE>!
 * </P>
 *
 * <PRE>
 *
 * package test;
 *
 * public interface IHello {
 *    public void sayHello ();
 * }
 *
 * </PRE>
 *
 * <H4>Step 2: Create a wrapper to be loaded by the custom class loader</H4>
 * <P>
 * Create a wrapper class that wraps the classes to be loaded by the custom
 * class loader. The wrapper's member class - hello - will be loaded by the same
 * class loader as the wrapper. For class casting purposes, the wrapper must
 * implement an interface that is loaded by the parent class loader. Typically,
 * the wrapper class is not part of the jar file containing the software module
 * to be loaded by this VersionClassLoader. By specifying the wrapper class as a
 * constructor argument, this wrapper will be loaded by this class loader. The
 * calling program will cast this wrapper class to the interface (<CODE>IHello
 * </CODE>), which is known to the system class loader.
 * </P>
 * <P>
 * The wrapper class (or adapter class) does not need to have the same interface
 * as its wrapped member. When wrapping a more complex module, it is easier to
 * have only one wrapper class for the whole module.
 * </P>
 * <P>
 * The wrapper class should have a no-argument constructor so that it can be
 * instantiated by <CODE>aClass.newInstance()</CODE>.
 * </P>
 *
 * <PRE>
 *
 * package test;
 *
 * class HelloWrapper implements IHello {
 *    private Hello hello;
 *                <I>// ^^^^^ the module class</I>
 *
```

```
 *    public HelloWrapper () {
 *       hello = new Hello();
 *                <I>// ^^^^^</I>
 *    }
 *
 *    public void sayHello () {
 *       hello.hello();
 *    }
 * }
 *
 * </PRE>
 *
 *
 * <H4>Step 3: Telling wrapper classes and versioned classes apart</H4>
 * <P>
 * The class loader needs a means to tell wrapper classes and versioned
 * classes apart.
 * The wrapper classes are to be loaded from the VM classpath, whereas the
 * versioned classes have their own classpathes, e.g. myjar-0.1.jar
 * or myjar-0.2.jar.
 * </P>
 * <P>
 * To this end a <CODE>WrapperIdentifier</CODE> must be provided that
 * implements the method <CODE>boolean isWrapper (String classname)</CODE>.
 * </P>
 *
 * <H4>Step 4: Loading the classes</H4>
 * <P>
 * Loading the classes consists mainly in loading the wrapper class. When the
 * wrapper class is loaded, its member classes are loaded, too. Member classes
 * are always loaded by the same classes as the classes they are members of.
 * </P>
 * <P>
 * When the wrapper and its members and the members' members have been loaded,
 * the wrapper is casted to the interface that is known to the application's
 * class loader.
 * </P>
 *
 *  * <PRE>
.
.
.
            String jar1 = "./lib/test1.jar";
            String jar2 = "./lib/test2.jar";
            String wrapper = "test.HelloWrapper";

            Class cl = Class.forName (wrapper, true,
                        new VersionClassLoader (jar1,
                            new VersionClassLoader.WrapperIdentifier () {
                              public boolean isWrapper (String classname) {
                                    return classname != null &&
                                          classname.endsWith("Wrapper.class");
                              }
                        }));
            Object o = cl.newInstance();
            hello = (IHello) o;
            System.out.println (hello.getClass().getClassLoader());

            hello.sayHello();

            cl = Class.forName (wrapper, true,
                        new VersionClassLoader (jar2,
                            new VersionClassLoader.WrapperIdentifier () {
                              public boolean isWrapper (String classname) {
                                    return classname != null &&
                                          classname.endsWith("Wrapper.class");
                              }
                        }));
            hello = (IHello) cl.newInstance();
```

```
            System.out.println (hello.getClass().getClassLoader());

            hello.sayHello();
.
.
.
 * </PRE>
 *
 * <P><I>Output</I></P>
 * <PRE>
Hello, World! (Version 1)
Hello, World! (Version 2)
 * </PRE>
 *
 * <H3>Articles and resources</H3>
 * <UL>
 * <LI>Java Language Specification, 2nd Edition</LI>
 * <LI>Ken Arnold, James Gosling, <I>The Java Language 3rd Edition </I></LI>
 * <LI>ibm.com/developerWorks, Understanding the Java ClassLoader</LI>
 * <LI>JDC Tech Tips: October 31, 2000 ( <A
 *
href="http://developer.java.sun.com/developer/TechTips/2000/tt1027.html">http://develop
er.java.sun.com/developer/TechTips/2000/tt1027.html
 * </A>)</LI>
 * <LI>GoF, Design Patterns, <I>Adapter</I></LI>
 * </UL>
 *
 * @author mhoehme (&gt;<A href="mailto:java@mhoehme.de">java@mhoehme</A>&lt;)
 * @version 0.2 (04/03/2007)
 */
public class VersionClassLoader extends ClassLoader {

      /**
       * An interface for a mechanism to identify wrapper clases.
       * <H3>A simple WrapperIdentifier</H3>
       * <PRE>
       *     public static class SimpleWrapperIdentifier {
       *       public boolean isWrapper (String classname) {
       *          return classname != null && classname.endsWith("Wrapper.class");
       *       }
       *   }
       * </PRE>
       */
      public static interface WrapperIdentifier {
             /**
              * Returns whether the specified class is a wrapper class or not.
              * @return <CODE>true</CODE> when the class is to be loaded from
              *         the VM class path, else <CODE>false</CODE>
              */
             boolean isWrapper (String classname);
      }

      private boolean verbose = false;

      private String[] myclasspath;

      private String[] vmclasspath;

      private WrapperIdentifier wrapperIdentifier;

      /**
       * Constructor. The wrapper classes are supposed to be loadable from the
       * Virtual Machine's class path.
       *
       *
       * @param classpath
       *            the jar files and/or directories from where to load the
       *            versioned classes
       */
```

```
    public VersionClassLoader(String classpath,
                              WrapperIdentifier wrapperIdentifier) {
        setup(classpath, wrapperIdentifier);
    }

    /**
     * Constructor with parent class loader.
     * The wrapper classes are supposed to be loadable from the
     * Virtual Machine's class path.
     *
     * @param classpath
     *            the jar files and/or directories from where to load the
     *            versioned classes
     * @param parent
     *            a parent class loader
     */
    public VersionClassLoader(String classpath,
                              WrapperIdentifier wrapperIdentifier,
                              ClassLoader parent) {
        super(parent);
        setup(classpath, wrapperIdentifier);
    }

    /**
     * Parses the classpath for the versioned classes and the VM class path.
     *
     * @param classpath
     *           the classpath for the versioned classes
     */
    private void setup(String classpath, WrapperIdentifier wrapperIdentifier) {
        this.myclasspath = getArray(classpath);
        this.vmclasspath = getArray(System.getProperty("java.class.path"));
        this.wrapperIdentifier = wrapperIdentifier;


        verbose = "true".equals (
             System.getProperty ("de.mhoehme.versionclassloader.verbose"));

        if (verbose) {
            System.out.println ("[VersionClassLoader VM classpath = "+
                    System.getProperty ("java.class.path")+"]");
            System.out.println ("[VersionClassLoader versioned classpath = "+
                    classpath+"]");
        }
    }

    /**
     * Creates an array of locations from a classpath
     * @param classpath
     *           a classpath
     * @return the locations
     */
    private String[] getArray(String classpath) {
        List list = new ArrayList();

        StringTokenizer tokenizer = new StringTokenizer(classpath,
                                         System.getProperty("path.separator"));
        while (tokenizer.hasMoreTokens()) {
            list.add(tokenizer.nextToken());
        }

        String[] s = new String[list.size()];
        return (String[]) list.toArray(s);
    }

    // ~~~[ OVERWRITTEN CLASSLOADER METHODS ]~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    /**
     * Overwrites {@link ClassLoader#loadClass (String, boolean)}. Loads a
     * class
```

```
     * <OL>
     * <LI>from the class loader's cache</LI>
     * <LI>by calling #findClass</LI>
     * <LI>delegating the request to the parent class loader</LI>
     * </OL>
     *
     * @param name
     * @param resolve
     * @throws ClassNotFoundException
     */
    protected synchronized Class loadClass(String name, boolean resolve)
                throws ClassNotFoundException {
            Class cl = null;

            // #1 - check whether the class has already been loaded
            if (cl == null) {
                    cl = findLoadedClass(name);
            }

            // #2 - if not, get from jar
            if (cl == null) {
                    cl = findClass(name);
            }

            // #3 - else ask the parent
            if (cl == null && getParent() != null) {
                    cl = getParent().loadClass(name);
            }

            // #4 - this is the end of the story
            if (cl == null) {
                    throw new ClassNotFoundException(name);
            }

            if (resolve) {
                    resolveClass(cl);
            }
            return cl;
    }



    /**
     * Loads a class from the specified locations.
     * <OL>
     * <LI>load class from specified locations (jar, directory)</LI>
     * <LI>wrapper class are loaded by the parent class loader</LI>
     * <OL>
     *
     * @param name
     *            the name of the class to be loaded
     * @return the class object or <CODE>null</CODE>
     * @throws ClassNotFoundException
     */
    protected Class findClass(String name) {
            Class cl = null;
            byte[] b = null;

            String classname = name.replace('.', '/').concat(".class");

            if (wrapperIdentifier.isWrapper(classname)) {
                    b = getBytes(vmclasspath, classname);
                    if (verbose && b != null) {
                            System.out.println ("[VersionClassLoader loaded "+
                                    classname+" from VM classpath]");
                    }
            } else {
                    b = getBytes(myclasspath, classname);
                    if (verbose && b != null) {
```

```
                        System.out.println ("[VersionClassLoader loaded "+
                              classname+" from specified location]");
                  }
            }

            if (b != null) {
                  cl = defineClass(name, b, 0, b.length);
            }

            return cl;
      }

      // ~~~[ PRIVATE ]~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      /**
       * Loads the bytecode from the specified location(s).
       *
       * @param loc
       *            the locations for loading the class
       * @param classname
       *            the name of the class to load
       * @return the bytecode
       */
      private byte[] getBytes(String[] classpath, String classname) {
            byte[] b = null;
            for (int i = 0; i < classpath.length; i++) {
                  if (classpath[i].endsWith(".jar")) {
                        b = getBytesFromJar(classpath[i], classname);
                  } else {
                        b = getBytesFromDisk(classpath[i], classname);
                  }
                  if (b != null) {
                        break;
                  }
            }
            return b;
      }


      /**
       * Loads the class specified by the classname from a directory.
       *
       * @param directory
       *            the directory (without trailing slash)
       * @param classname
       *            the name of the class
       * @return the byte code
       * @throws ClassNotFoundException
       */
      private byte[] getBytesFromDisk(String directory, String classname) {
            byte[] b = null;
            File f = new File(directory + System.getProperty("file.separator")
                        + classname);
            try {
                  BufferedInputStream in = new BufferedInputStream(
                              new FileInputStream(f));

                  b = new byte[(int) f.length()];

                  in.read(b);

                  in.close();
            } catch (Exception e) {
                  // maybe in the next location
            }
            return b;
      }

      /**
```

```
     * Loads the byte code of the specified classfile from the specified jarfile
     *
     * @param jarfilename
     *            the jar file from which the byte code is to be obtained
     * @param classfile
     *            the class to be loaded
     * @return the byte code of the class file, or <CODE>null</CODE>
     * @throws ClassNotFoundException
     */
    private byte[] getBytesFromJar(String jarfilename, String classfile) {
        byte[] b = null;
        File f = new File(jarfilename);
        try {
            JarFile jarfile = new JarFile(f);

            ZipEntry entry = jarfile.getEntry(classfile);

            if (entry != null) {
                BufferedInputStream in = new BufferedInputStream(jarfile
                        .getInputStream(entry));

                b = new byte[(int) entry.getSize()];

                in.read(b);

                in.close();
            }
        } catch (Exception e) {
            // maybe in the next location
        }
        return b;
    }
}
```

Hello.java

```
package test;
public class Hello {
   public void sayHello () {
      System.out.println ("@hello@");
   }
}
```

The interface to be loaded by the system class loader

```
package test;
public interface IHello {
    public void sayHello ();
}
```

The wrapper class HelloWrapper.java to be loaded by the custom class loader

```
package test;
public class HelloWrapper implements IHello {
   private Hello hello;
   public HelloWrapper () {
      hello = new Hello();
   }
   public void sayHello () {
      hello.sayHello();
   }
}
```

Test.java

```
package de.mhoehme.versionclassloader.test;

import test.IHello;
import de.mhoehme.versionclassloader.VersionClassLoader;

public class Test {

   public static void main (String[] args) throws Exception {
      System.out.println ("Classloader of Test.class = "+ Test.class.getClassLoader());

      String version1 = "./lib/test1.jar";
      String version2 = "./lib/test2.jar";
      String wrapper = "test.HelloWrapper";

      VersionClassLoader.WrapperIdentifier wrapperIdentifier =
            new VersionClassLoader.WrapperIdentifier() {
               public boolean isWrapper (String classname) {
                  return classname != null && classname.endsWith("Wrapper.class");
               }
            };

      Class cl = Class.forName (wrapper, true,
            new VersionClassLoader (version1, wrapperIdentifier));
      IHello hello = (IHello) cl.newInstance();
      System.out.println ("---> Hello class loader: "+
            hello.getClass().getClassLoader());

      hello.sayHello(); // => hello 1


      cl = Class.forName (wrapper, true,
            new VersionClassLoader (version2, wrapperIdentifier));
      hello = (IHello) cl.newInstance();
      System.out.println ("---> Hello class loader: "+
            hello.getClass().getClassLoader());

      hello.sayHello(); // => hello 2
   }
}
```

And the ant build file:

```
<project name="classloading" default="jar" basedir=".">

      <property name="src"              value="./src"/>
      <property name="classes"          value="./bin"/>
      <property name="lib"              value="lib"/>
      <property name="temp.src"         value="./temp"/>
      <property name="temp.classes"     value="./tempclasses"/>

      <!-- create two versions of the same class and jar them -->
      <!-- todo: find a better way to execute the same task;
           the property "number" cannot be overwritten -->
      <target name="testjars">
            <ant target="jar1"/>
            <ant target="jar2"/>
      </target>

      <target name="jar1">
            <property name="number" value="1"/>
            <mkdir dir="${temp.src}"/>
            <mkdir dir="${temp.classes}"/>
            <filter token="hello" value="hello ${number}"/>
            <copy todir="${temp.src}" filtering="true">
                  <fileset dir="${src}" includesfile="test/Hello.java"/>
            </copy>
            <javac srcdir="${temp.src}" destdir="${temp.classes}"/>
```

```
            <jar destfile="${lib}/test${number}.jar">
                  <fileset dir="${temp.classes}">
                        <include name="test/Hello.class"/>
                  </fileset>
            </jar>
            <delete dir="${temp.src}"/>
            <delete dir="${temp.classes}"/>
      </target>

      <target name="jar2">
            <property name="number" value="2"/>
            <mkdir dir="${temp.src}"/>
            <mkdir dir="${temp.classes}"/>
            <filter token="hello" value="hello ${number}"/>
            <copy todir="${temp.src}" filtering="true">
                  <fileset dir="${src}" includesfile="test/Hello.java"/>
            </copy>
            <javac srcdir="${temp.src}" destdir="${temp.classes}"/>
            <jar destfile="${lib}/test${number}.jar">
                  <fileset dir="${temp.classes}">
                        <include name="test/Hello.class"/>
                  </fileset>
            </jar>
            <delete dir="${temp.src}"/>
            <delete dir="${temp.classes}"/>
      </target>

      <!-- compile test program -->
      <target name="compile">
            <javac srcdir="${src}" destdir="${classes}"/>
      </target>

      <target name="jar" depends="compile">
            <jar destfile="test.jar">
                  <fileset dir="${classes}">
                        <exclude name="test/Hello.class"/>
                        <exclude name="test/HelloWrapper.class"/>
                  </fileset>
            </jar>
      </target>

</project>
```

And the output is (started in Eclipse – hence `/bin`):

```
Classloader of Test.class = sun.misc.Launcher$AppClassLoader@11b86e7
[VersionClassLoader VM classpath = D:\work\Projekte\ClassLoading\bin]
[VersionClassLoader versioned classpath = ./lib/test1.jar]
[VersionClassLoader loaded test/HelloWrapper.class from VM classpath]
HelloWrapper loader = de.mhoehme.versionclassloader.VersionClassLoader@9304b1
[VersionClassLoader loaded test/Hello.class from specified location]
---> Hello class loader: de.mhoehme.versionclassloader.VersionClassLoader@9304b1
hello 1
[VersionClassLoader VM classpath = D:\work\Projekte\ClassLoading\bin]
[VersionClassLoader versioned classpath = ./lib/test2.jar]
[VersionClassLoader loaded test/HelloWrapper.class from VM classpath]
HelloWrapper loader = de.mhoehme.versionclassloader.VersionClassLoader@1fb8ee3
[VersionClassLoader loaded test/Hello.class from specified location]
---> Hello class loader: de.mhoehme.versionclassloader.VersionClassLoader@1fb8ee3
hello 2
```

References

Java Language Specification, Second Edition

Ken Arnold, James Gosling, The Java Programming Language, Third Edition

ibm.com/developerWorks, Understanding the Java ClassLoader

JDC Tech Tips: October 31, 2000
GoF, Design Patterns (Adapter Pattern)