Java Reflection - Dynamic Class Loading and Reloading

By Jakob Jenkov

It is possible to load and reload classes at runtime in Java, though it is not as straightforward as one mi hoped. This text will explain when and how you can load and reload classes in Java.

You can argue whether Java's dynamic class loading features are really part of Java Reflection, or a pa Java platform. Anyways, the article has been put in the Java Reflection trail in lack of a better place to p

The ClassLoader

All classes in a Java application are loaded using some subclass of `java.lang.ClassLoader`. Lo classes dynamically must therefore also be done using a `java.lang.ClassLoader` subclass.

d, all classes it references are loaded too. This class loading pattern happens re loaded. This may not be all classes in the application. Unreferenced classes are eferenced.

TOC Previous Next

The ClassLoader Hierarchy

Class loaders in Java are organized into a hierarchy. When you create a new standard Java `ClassLo` must provide it with a parent `ClassLoader`. If a `ClassLoader` is asked to load a class, it will ask its loader to load it. If the parent class loader can't find the class, the child class loader then tries to load it i

Class Loading

The steps a given class loader uses when loading classes are:

1. Check if the class was already loaded.
2. If not loaded, ask parent class loader to load the class.
3. If parent class loader cannot load class, attempt to load it in this class loader.

When you implement a class loader that is capable of reloading classes you will need to deviate a bit fr sequence. The classes to reload should not be requested loaded by the parent class loader. More on th

Dynamic Class Loading

Loading a class dynamically is easy. All you need to do is to obtain a `ClassLoader` and call its `load` method. Here is an example:

```
public class MainClass {

  public static void main(String[] args){

    ClassLoader classLoader = MainClass.class.getClassLoader();

    try {
```

```
        Class aClass = classLoader.loadClass("com.jenkov.MyClass");
        System.out.println("aClass.getName() = " + aClass.getName());
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }

}
```

Dynamic Class Reloading

Dynamic class reloading is a bit more challenging. Java's builtin Class loaders always checks if a class loaded before loading it. Reloading the class is therefore not possible using Java's builtin class loaders class you will have to implement your own `ClassLoader` subclass.

Even with a custom subclass of `ClassLoader` you have a challenge. Every loaded class needs to be is done using the `ClassLoader.resolve()` method. This method is final, and thus cannot be overr `ClassLoader` subclass. The `resolve()` method will not allow any given `ClassLoader` instance t same class twice. Therefore, everytime you want to reload a class you must use a new instance of your `ClassLoader` subclass. This is not impossible, but necessary to know when designing for class reloa

Designing your Code for Class Reloading

As stated earlier you cannot reload a class using a `ClassLoader` that has already loaded that class Therefore you will have to reload the class using a different `ClassLoader` instance. But this poses so challenges.

Every class loaded in a Java application is identified by its fully qualified name (package name + class the `ClassLoader` instance that loaded it. That means, that a class `MyObject` loaded by class loade same class as the `MyObject` class loaded with class loader B. Look at this code:

```
MyObject object = (MyObject)
    myClassReloadingFactory.newInstance("com.jenkov.MyObject");
```

Notice how the `MyObject` class is referenced in the code, as the type of the `object` variable. This ca `MyObject` class to be loaded by the same class loader that loaded the class this code is residing in.

If the `myClassReloadingFactory` object factory reloads the `MyObject` class using a different cla than the class the above code resides in, you cannot cast the instance of the reloaded `MyObject` clas `MyObject` type of the `object` variable. Since the two `MyObject` classes were loaded with different loaders, the are regarded as different classes, even if they have the same fully qualified class name. Tr an object of the one class to a reference of the other will result in a `ClassCastException`.

It is possible to work around this limitation but you will have to change your code in either of two ways:

1. Use an interface as the variable type, and just reload the implementing class.
2. Use a superclass as the variable type, and just reload a subclass.

Here are two coresponding code examples:

```
MyObjectInterface object = (MyObjectInterface)
    myClassReloadingFactory.newInstance("com.jenkov.MyObject");
```

```
MyObjectSuperclass object = (MyObjectSuperclass)
    myClassReloadingFactory.newInstance("com.jenkov.MyObject");
```

Either of these two methods will work if the type of the variable, the interface or superclass, is not reloa implementing class or subclass is reloaded.

To make this work you will of course need to implement your class loader to let the interface or supercla loaded by its parent. When your class loader is asked to load the `MyObject` class, it will also be aske `MyObjectInterface` class, or the `MyObjectSuperclass` class, since these are referenced from `MyObject` class. Your class loader must delegate the loading of those classes to the same class load loaded the class containing the interface or superclass typed variables.

ClassLoader Load / Reload Example

The text above has contained a lot of talk. Let's look at a simple example. Below is an example of a sim `ClassLoader` subclass. Notice how it delegates class loading to its parent except for the one class it to be able to reload. If the loading of this class is delegated to the parent class loader, it cannot be reloa Remember, a class can only be loaded once by the same `ClassLoader` instance.

As said earlier, this is just an example that serves to show you the basics of a `ClassLoader`'s behavi production ready template for your own class loaders. Your own class loaders should probably not be li single class, but a collection of classes that you know you will need to reload. In addition, you should pro hardcode the class paths either.

```
public class MyClassLoader extends ClassLoader{
```

```
    public MyClassLoader(ClassLoader parent) {
        super(parent);
    }

    public Class loadClass(String name) throws ClassNotFoundException {
        if(!"reflection.MyObject".equals(name))
                return super.loadClass(name);

        try {
            String url = "file:C:/data/projects/tutorials/web/WEB-INF/" +
                         "classes/reflection/MyObject.class";
            URL myUrl = new URL(url);
            URLConnection connection = myUrl.openConnection();
            InputStream input = connection.getInputStream();
            ByteArrayOutputStream buffer = new ByteArrayOutputStream();
            int data = input.read();

            while(data != -1){
                buffer.write(data);
                data = input.read();
            }

            input.close();

            byte[] classData = buffer.toByteArray();

            return defineClass("reflection.MyObject",
                    classData, 0, classData.length);

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }

}
```

Below is an example use of the `MyClassLoader`.

```
public static void main(String[] args) throws
    ClassNotFoundException,
    IllegalAccessException,
    InstantiationException {

    ClassLoader parentClassLoader = MyClassLoader.class.getClassLoader();
    MyClassLoader classLoader = new MyClassLoader(parentClassLoader);
    Class myObjectClass = classLoader.loadClass("reflection.MyObject");

    AnInterface2       object1 =
            (AnInterface2) myObjectClass.newInstance();

    MyObjectSuperClass object2 =
            (MyObjectSuperClass) myObjectClass.newInstance();

    //create new class loader so classes can be reloaded.
    classLoader = new MyClassLoader(parentClassLoader);
    myObjectClass = classLoader.loadClass("reflection.MyObject");

    object1 = (AnInterface2)       myObjectClass.newInstance();
    object2 = (MyObjectSuperClass) myObjectClass.newInstance();

}
```

Here is the `reflection.MyObject` class that is loaded using the class loader. Notice how it both e
superclass and implements an interface. This is just for the sake of the example. In your own code you v
have to one of the two - extend or implement.

```
public class MyObject extends MyObjectSuperClass implements AnInterface2{
    //... body of class ... override superclass methods
    //    or implement interface methods
}
```