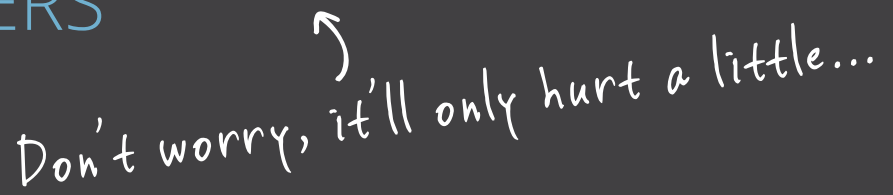
DO YOU REALLY GET CLASSLOADERS?

A [FRIGHTFUL] JOURNEY INTO THE HEART OF JAVA CLASSLOADERS



REBELLABS

PART I

AN OVERVIEW OF JAVA CLASSLOADERS, DELEGATION AND COMMON PROBLEMS

In this part, we provide an overview of classloaders, explain how delegation works and examine how to solve common problems that Java developers encounter with classloaders on a regular basis.

INTRODUCTION
WHY YOU SHOULD KNOW, AND FEAR, CLASSLOADERS

Classloaders are at the core of the Java language. Java EE containers, OSGi, various web frameworks and other tools use classloaders heavily. Yet, something goes wrong with classloading, would you know how to solve it?

Join us for a tour of the Java classloading mechanism, both from the JVM and developer point-of-view. We will look at typical problems related to classloading and how to solve them. NoClassDefFoundError, LinkageError and many others are symptoms of specific things going wrong that you can usually find and fix. For each problem, we'll go through an example with a corresponding solution. We'll also take a look at how and why classloaders leak and how can that be remedied.

And for dessert, we review how to reload a Java class using a dynamic classloader. To get there we'll see how objects, classes and classloaders are tied to each other and the process required to make changes. We begin with a bird's eye view of the problem, explain the reloading process, and then proceed to a specific example to illustrate typical problems and solutions.

Enter java.lang.ClassLoader

Let's dive into the beautiful world of classloader mechanics.

It's important to realize that each classloader is itself an object--an instance of a class that extends java.lang.ClassLoader. Every class is loaded by one of those instances and developers are free to subclass java.lang.ClassLoader to extend the manner in which the JVM loads classes.

There might be a little confusion: if a classloader has a class and every class is loaded by a classloader, then what comes first? We need an understanding of the mechanics of a classloader (by proxy of examining its API contract) and the JVM classloader hierarchy.

First, here is the API, with some less relevant parts omitted:

```
package java.lang;

public abstract class ClassLoader {

  public Class loadClass(String name);
  protected Class defineClass(byte[] b);

  public URL getResource(String name);
  public Enumeration getResources(String name);

  public ClassLoader getParent()
}
```

By far, the most important method of java.lang.ClassLoader is the **loadClass** method, which takes the fully qualified name of the class to be loaded and returns an object of class Class.

The **defineClass** method is used to materialize a class for the JVM. The byte array parameter of `defineClass` is the actual class byte code loaded from disk or any other location.

getResource and **getResources** return URLs to actually existing resources when given a name or a path to an expected resource. They are an important part of the classloader contract and have to handle delegation the same way as `loadClass` - delegating to the parent first and then trying to find the resource locally. We can even view `loadClass` as being roughly equivalent to `defineClass(getResource(name).getBytes()).`

The `getParent` method returns the parent classloader. We'll have a more detailed look at what that means in the next section.

The lazy nature of Java has an effect on how do classloaders work - everything should be done at the last possible moment. A class will be loaded only when it is referenced somehow - by calling a constructor, a static method or field.

Now let's get our hands dirty with some real code. Consider the following example: class A instantiates class B.

```
public class A {
  public void doSomething() {
 B b = new B();
 b.doSomethingElse();
  }
}
```

The statement `B b = new B()` is semantically equivalent to `B b = A.class.getClassLoader().loadClass("B").newInstance()`

As we see, every object in Java is associated with its class `(A.class)` and every class is associated with classloader `(A.class.getClassLoader())` that was used to load the class.

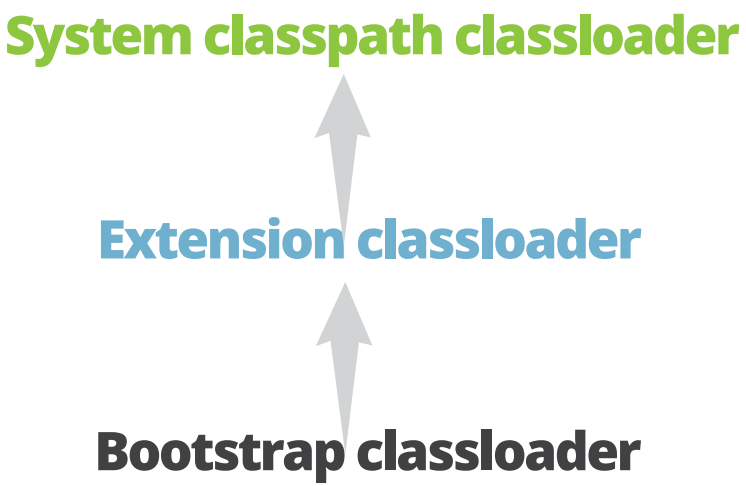
When we instantiate a `ClassLoader`, we can specify a parent classloader as a constructor argument. If the parent classloader isn't specified explicitly, the virtual machine's system classloader will be assigned as a default parent. And with this note, let's examine the classloader hierarchy of a JVM more closely.

Classloader Hierarchy

Whenever a new JVM is started the **bootstrap classloader** is responsible to load key Java classes (from `java.lang` package) and other runtime classes to the memory first. The bootstrap classloader is a parent of all other classloaders. Consequently, it is the only one without a parent.

Next comes the extension classloader. It has the bootstrap classloader as parent and is responsible for loading classes from all `.jar` files kept in the `java.ext.dirs` path--these are available regardless of the JVM's classpath.

The third and most important classloader from a developer's perspective is the **system classpath classloader**, which is an immediate child of the extension classloader. It loads classes from directories and jar files specified by the `CLASSPATH` environment variable, `java.class.path` system property or `-classpath` command line option.



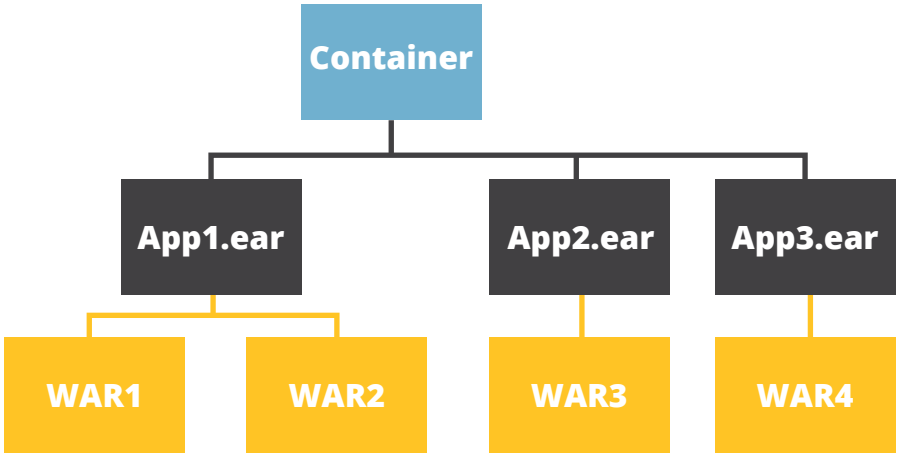
Note that the classloader hierarchy is not an inheritance hierarchy, but a delegation hierarchy. Most classloaders delegate finding classes and resources to their parent before searching their own classpath. If the parent classloader cannot find a class or resource, only then does the classloader attempt to find them locally. In effect, a classloader is responsible for loading only the classes not available to the parent; classes loaded by a classloader higher in the hierarchy cannot refer to classes available lower in the hierarchy.

The classloader delegation behavior is motivated by the need to avoid loading the same class several times. Back in 1995, the chief application of the Java platform was thought to be web applets living in the client (browser). As connections at that time were slow, it was important to load the application lazily over the network. Classloaders enabled this behavior by abstracting away how each class was loaded. All the bells and whistles came later.

However, history has shown that Java is much more useful on the server side, and in Java EE, the order of the lookups is often reversed: a classloader may try to find classes locally before going to the parent.

Java EE delegation model

Here's a typical view of an application container's classloaders hierarchy: there is a classloader of the container itself, every EAR module has its own classloader and every WAR has its own classloader.



The Java Servlet specification recommends that a web module's classloader look in the local classloader before delegating to its parent--the parent classloader is only asked for resources and classes not found in the module.

In some application containers this recommendation is followed, but in others the web module's classloader is configured to follow the same delegation model as other classloaders--so it is advisable to consult the documentation of the application container that you use.

The reason for reversing the ordering between local and delegated lookups is that application containers ship with lots of libraries with their own release cycles that might not fit application developers. The classical example is the log4j library--often one version of it is shipped with the container and different versions bundled with applications.

> **JEVGENI** KABANOV:
> The reversed behavior of web module classloader has caused more problems with classloaders than anything else... ever.

Now, let's take a look at several common classloading problems that we might encounter and provide possible solutions.

When something goes wrong

The Java EE delegation model can cause some interesting problems with classloading. `NoClassDefFoundError, LinkageError, ClassNotFoundException, NoSuchMethodError, ClassCastException`, etc. are very common exceptions encountered while developing Java EE applications. We can make all sorts of assumptions about the root causes of these problems, but it's important to verify them so that we don't feel like "dummies" later.

NoClassDefFoundError

`NoClassDefFoundError` is one of the most common problems that you can face when developing Java EE Java applications. The complexity of the root cause analysis and resolution process mainly depend of the size of your Java EE middleware environment; especially given the high number of classloaders present across the various Java EE applications.

As the Javadoc entry says, `NoClassDefFoundError` is thrown if the Java Virtual Machine or a `ClassLoader` instance tries to load in the definition of a class and no definition of the class could be found. Which means, the searched-for class definition existed when the currently executing class was compiled, but the definition can not be found at runtime.

This is the reason you cannot always rely on your IDE telling you that everything is OK, the code compiles and should work just fine. Instead, this is a runtime problem and an IDE cannot help here.

> **JEVGENI** KABANOV:
> All classloading happens at runtime, which makes the IDE results irrelevant.

Let's look at the following example.

```
public class HelloServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.print(new Util().sayHello());
}
```

The servlet `HelloServlet` instantiates an instance of `Util` class which provides the message to be printed. Sadly, when the request is executed we might see something as follows:


```
java.lang.NoClassdefFoundError: Util
       HelloServlet:doGet(HelloServlet.java:17)
       javax.servlet.http.HttpServlet.service(HttpServlet.
java:617)
       javax.servlet.http.HttpServlet.service(HttpServlet.
java:717)
```

How can we resolve the problem? Well, the most obvious action that you might do is to check if the missing `Util` class has actually been included into the package.

One of the tricks that we can use here is to make the container classloader to confess where is it loading the resources from. To do that we could try to cast the `HelloServlet's` classloader to **URLClassLoader** and ask for its classpath.

```
public class HelloServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.print(Arrays.toString(
      ((URLClassLoader)HelloServlet.class.getClassLoader()).getURLs()));
}
```

The result might easily be something like this:

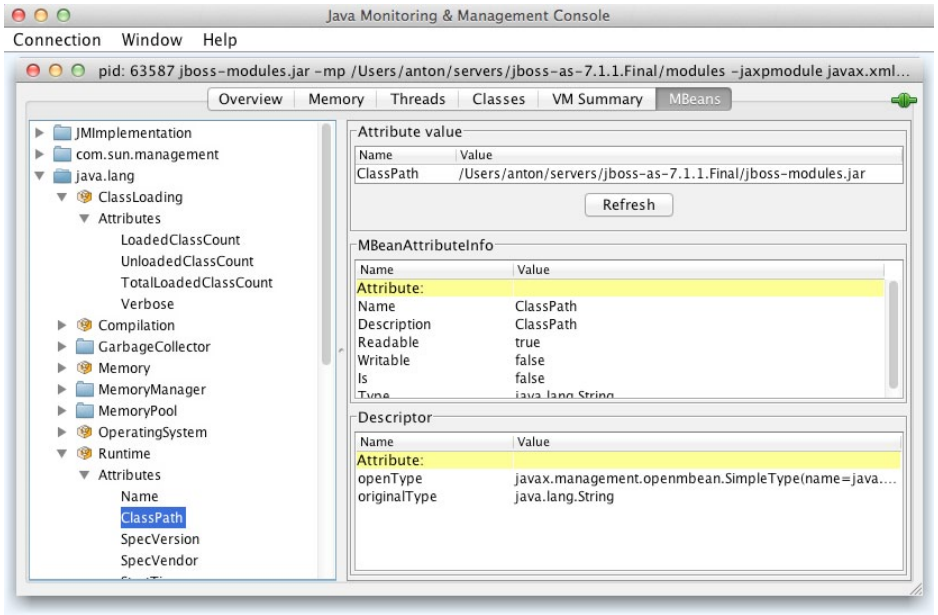
```
file:/Users/myuser/eclipse/workspace/.metadata/.plugins/
org.eclipse.wst.server.core/tmp0/demo/WEB-INF/classes,
file:/Users/myuser/eclipse/workspace/.metadata/.plugins/
org.eclipse.wst.server.core/tmp0/demo/WEB-INF/lib/demo-
lib.jar
```

The path to the resources `(file:/Users/myuser/eclipse/workspace/.metadata/)` actually reveals that the container was started from Eclipse and this is where the IDE unpacks the archive for deployment.

Now we could check if the missing `Util` is actually included into `demo-lib.jar` or if it exists in `WEB-INF/classes` directory of the expanded archive.

So for our particular example it might be the case that `Util` class was supposed to be packaged to the `demo-lib.jar` but we haven't retriggered the build process and the class wasn't included to the previously existing package, hence the error.

The `URLClassLoader` trick might not work in all application servers. Another way we could try to figure out where the resources are loaded from is to use jconsole utility to attach to the container JVM process in order to inspect the class path. For instance, the screenshot (below) demonstrates the `jconsole` window attached to JBoss application server process and we can see the `ClassPath` attribute value from the Runtime properties.



NoSuchMethodError

In another scenario with the same example we might encounter the following exception:

```
java.lang.NoSuchMethodError: Util.sayHello()Ljava/lang/String;
        HelloServlet:doGet(HelloServlet.java:17)
        javax.servlet.http.HttpServlet.service(HttpServlet.
        java:617)
        javax.servlet.http.HttpServlet.service(HttpServlet.
        java:717)
```

NoSuchMethodError represents a different problem. In this case, the class that we're referring to exists, but an incorrect version of it is loaded, hence the required method is not found. To resolve the issue, we must first of all understand where the class was loaded from. The easiest way to do so is to add `'-verbose:class'` command line parameter to the JVM, but if you can change the code quickly (e.g. with JRebel) then you can use the fact that getResource searches the same classpath as `loadClass`.

```
public class HelloServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    PrintWriter out = response.getWriter();
  out.print(HelloServlet.class.getClassLoader().getResource(
      Util.class.getName.replace('.', '/') + ".class"));

}
```

Assume, the result of the request execution for the example above is as follows:

```
file:/Users/myuser/eclipse/workspace/.metadata/.plugins/org.
eclipse.wst.server.core/tmp0/demo/WEB-INF/lib/demo-lib.jar!/
Util.class
```

Now we need to verify our assumption about the incorrect version of the class. We can use **javap** utility to decompile the class - then we can see if the required method actually exists or not.

```
$ javap -private Util
Compiled from "Util.java"
public class Util extends java.lang.Object {
   public Util();
}
```

As you can see, there isn't sayHello method in the decompiled version of the `Util` class. Probably, we had the initial version of the `Util` class packaged in the `demo-lib.jar` but we didn't rebuild the package after adding the new `sayHello` method.

Variants of the wrong class problem

`NoClassDefFoundError` and `NoSuchMethodError` are very typical when dealing with Java EE applications and it is the required skill for Java developers to be able to understand the nature of those errors in order to effectively solve the problems.

There are plenty of variants of these problems: `AbstractMethodError, ClassCastException, IllegalAccessError` - basically all of them

encounter when we think that application uses one version of a class but it actually uses some other version, or the class is loaded in some different way than is required.

ClassCastException

Let's demonstrate a case for `ClassCastException`. We'll modify the initial example to use a factory in order to provide the implementation of a class that provides the greeting message. Sounds contrived but this is quite a common pattern.

```
public class HelloServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
    PrintWriter out = response.getWriter();
  out.print(((Util)Factory.getUtil()).sayHello());
}

class Factory {
   public static Object getUtil() {
      return new Util();
   }
}
```

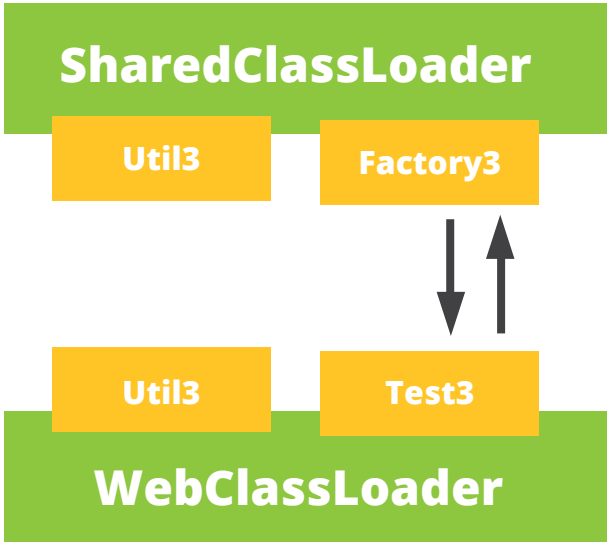
The possible result of the request is the unexpected `ClassCastException`:

```
java.lang.ClassCastException: Util cannot be cast to Util
       HelloServlet:doGet(HelloServlet.java:18)
       javax.servlet.http.HttpServlet.service(HttpServlet.
       java:617)
       javax.servlet.http.HttpServlet.service(HttpServlet.
       java:717)
```

It means that `HelloServlet` and `Factory` classes operate in different context. We have to figure out how these classes were loaded. Let's use `-verbose:class` and figure out how the `Util` class was loaded in regards to `HelloServlet` and `Factory` classes.

```
[Loaded Util from file:/Users/ekabanov/Applications/
apache-tomcat-6.0.20/lib/cl-shared-jar.jar]
[Loaded Util from file:/Users/ekabanov/Documents/
workspace-javazone/.metadata/.plugins/org.eclipse.wst.
server.core/tmp0/wtpwebapps/cl-demo/WEB-INF/lib/cl-demo-
jar.jar]
```

So the `Util` class is loaded from two different locations by different classloaders. One is in the web application classloader and the other in the application container classloader.



But why are they incompatible? Turns out that originally every class in Java was identified uniquely by its fully qualified name. But in 1997 a paper was published that exposed an expansive security issue cause by this--namely, it was possible for a sandboxed application (i.e. applet) to define any class including `java.lang.String` and inject its own code outside the sandbox.

The solution was to identify the class by a combination of the fully qualified name and the classloader! It means that `Util` class loaded from classloader A and `Util` class loaded from classloader B are different classes as far as the JVM is concerned and one cannot be cast to the other!

The root of this problem is the reversed behavior of the web classloader. If the web classloader would behave in the same way as the other classloaders, then the `Util` class would have been loaded once from the application container classloader and no `ClassCastException` would be thrown.

LinkageError

Things can get even worse. Let's slightly modify the `Factory` class from the previous example so that the `getUtil` method now returns `Util` type instead of an Object:

```
class Factory {
     public static Util getUtil() {
          return new Util();
     }
}
```

The result of the execution is now a **LinkageError** instead of `ClassCastException:`

```
java.lang.LinkageError: loader constraint violation: when
resolving method Factory.getUtil()LUtil; <…>
HelloServlet:doGet(HelloServlet.java:18)
javax.servlet.http.HttpServlet.service(HttpServlet.
java:617)
javax.servlet.http.HttpServlet.service(HttpServlet.
java:717)
```

The underlying issue is the same as with the `ClassCastException`--the only difference is that we do not cast the object, but instead, the loader constraint kicks in resulting in `LinkageError`.

A very important principle when dealing with classloaders is to recognize that classloading behaviour often defeats your intuitive understanding, so validating your assumptions is very important. For example, in case of `LinkageError` looking at the code or build process will hinder rather than help your progress. The key is to look where exactly classes are loaded from, how did they get there and how to prevent that from happening in the future.

A common reason for same classes present in more than one classloader is when different versions of the same library is bundled in different places --e.g. the application server and the web application. This typically happens with de facto standard libraries like log4j or hibernate. In that case the solution is either unbundling the library from the web application or taking painstaking care to avoid using the classes from the parent classloader.

IllegalAccessError

It turns out that not only is the class identified by its name and classloader, but the rule is also applicable for packages. To demonstrate that, let's change the `Factory.getUtil` method's access modifier to default:

```
class Factory {
    static Object getUtil() {
        return new Util();
    }
}
```

Both `HelloServlet` and `Factory` are assumed to reside in the same (default) package, so `getUtil` is visible in the `HelloServlet` class. Unfortunately, if we try to access it at runtime, we'll see the `IllegalAccessError` exception

```
java.lang.IllegalAccessError: tried to access method
Factory.getUtil()Ljava/lang/Object;
      HelloServlet:doGet(HelloServlet.java:18)
      javax.servlet.http.HttpServlet.service(HttpServlet.java:617)
      javax.servlet.http.HttpServlet.service(HttpServlet.java:717)
```

Although the access modifier is correct for the application to compile, the classes are loaded from different classloaders at runtime, and the application fails to operate. This is caused by the fact that same as classes, packages are also identified by their fully qualified name and the classloader for the same security reasons.

`ClassCastException`, `LinkageError` and `IllegalAccessError` manifest themselves a bit differently according to the implementation but the root cause is the same - classes being loaded by different classloaders.

YOUR CLASSLOADER CHEATSHEET

Print out this cheatsheet and see quickly how to fix common issues with classloaders. It might save your sanity, job, or both one day!

No class found

Variants

- ClassNotFoundException
- NoClassDefFoundError

Helpful

- IDE class lookup (Ctrl+Shift+T in Eclipse)
- find *.jar -exec jar -tf '{}'\; | grep MyClass
- URLClassLoader.getUrls()
- Container specific logs

Wrong class found

Variants

- IncompatibleClassChangeError
  - AbstractMethodError
  - NoSuch(Method|Field)Error
- ClassCastException, IllegalAccessError

Helpful

- -verbose:class
- ClassLoader.getResource()
- javap -private MyClass

More than one class found

Variants

- LinkageError (class loading constraints violated)
- ClassCastException, IllegalAccessError

Helpful

- -verbose:class
- ClassLoader.getResource()

REBELLABS

PART II
DYNAMIC CLASSLOADERS IN WEB APPS

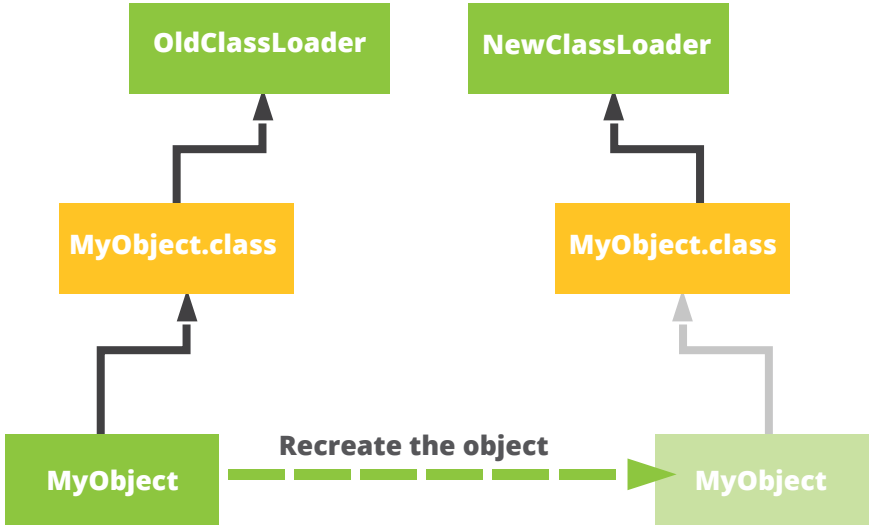
In Part II, we review how dynamic classloaders are used in web applications and module systems, what headaches they can cause and discuss how to do it better.

Dynamic classloading

The first thing to understand when talking about dynamic classloading is the relation between classes and objects. All Java code is associated with methods contained in classes. Simplified, you can think of a class as a collection of methods, that receive "this" as the first argument. The class with all its methods is loaded into memory and receives a unique identity. In the Java API this identity is represented by an instance of `java.lang.Class` that you can access using the `MyObject.class` expression.

Every object created gets a reference to this identity accessible through the `Object.getClass()` method. When a method is called on an object, the JVM consults the class reference and calls the method of that particular class. That is, when you call `mo.method()` (where mo is an instance of `MyObject`), then it's semantically equivalent to `mo.getClass().getDeclaredMethod("method").invoke(mo)` (this is not what the JVM actually does, but the result is the same).

Every `Class` object is in turn associated with its classloader (`MyObject.class.getClassLoader()`). The main role of the classloader is to define a class scope - where the class is visible and where it isn't. This scoping allows for classes with the same name to exist as long as they are loaded in different classloaders. It also allows loading a newer version of the class in a different classloader.



The main problem with code reloading in Java is that although you can load a new version of a class, it will get a completely different identity and the existing objects will keep referring the previous version of the class. So when a method is called on those objects it will execute the old version of the method.

Let's assume that we load a new version of the `MyObject` class. Let's refer to the old version as `MyObject_1` and to the new one as `MyObject_2`. Let's also assume that `MyObject.method()` returns "1" in `MyObject_1` and "2" in `MyObject_2`. Now if `mo2` is an instance of `MyObject_2`:

- `mo.getClass() != mo2.getClass()`
- `mo.getClass().getDeclaredMethod("method").invoke(mo)`
- `!= mo2.getClass().getDeclaredMethod("method").invoke(mo2)`
- `mo.getClass().getDeclaredMethod("method").invoke(mo2) throws a ClassCastException, because the Class identities of mo and mo2 do no match.`

Down & Dirty

Let's see how this would look in code. Remember, what we're trying to do here is load a newer version of a class, in a different classloader. We'll use a `Counter` class that looks like this:

```
public class Counter implements ICounter {
  private int counter;


public String message() {
    return "Version 1";
  }
  public int plusPlus() {
    return counter++;
  }
  public int counter() {
    return counter;
  }
}
```

We'll use a `main()` method that will loop infinitely and print out the information from the Counter class. We'll also need two instances of the `Counter` class: *counter1* that is created once in the beginning and *counter2* that is recreated on every roll of the loop:

```
public class Main {
  private static ICounter counter1;
  private static ICounter counter2;

  public static void main(String[] args)  {
    counter1 = CounterFactory.newInstance();

    while (true) {
      counter2 = CounterFactory.newInstance();

      System.out.println("1) " +
        counter1.message() + " = " + counter1.plusPlus());
      System.out.println("2) " +
        counter2.message() + " = " + counter2.plusPlus());
      System.out.println();

      Thread.currentThread().sleep(3000);
    }
  }
}
```

`ICounter` is an interface with all the methods from `Counter`. This is necessary because we'll be loading `Counter` in an isolated classloader, so `Main` cannot use it directly (otherwise we'd get a `ClassCastException`).

```
public interface ICounter {
  String message();



int plusPlus();
}
```

From this example, you might be surprised to see how easy it is to create a dynamic classloader. If we remove the exception handling it boils down to this:

```
public class CounterFactory {
  public static ICounter newInstance() {
    URLClassLoader tmp =
      new URLClassLoader(new URL[] {getClassPath()})
{
        public Class loadClass(String name) {
          if ("example.Counter".equals(name))
            return findClass(name);
          return super.loadClass(name);
        }
      };

    return (ICounter)
      tmp.loadClass("example.Counter").newInstance();
  }
}
```

The method `getClassPath()` for the purposes of this example could return the hardcoded classpath. However we can use the `ClassLoader.getResource()` API to automate that.
Now let's run `Main.main` and see the output after waiting for a few loop rolls:

```
1) Version 1 = 3
2) Version 1 = 0
```

As expected, while the counter in the first instance is updated, the second stays at "0". If we change the `Counter.message()` method to return "Version 2". The output will change as follows:

```
1) Version 1 = 4
2) Version 2 = 0
```

As we can see, the first instance continues incrementing the counter, but uses the old version of the class to print out the version. The second instance class was updated, however all of the state is lost.

To remedy this, let's try to reconstruct the state for the second instance. To do that we can just copy it from the previous iteration.

First we add a new `copy()` method to `Counter` class (and corresponding interface method):

```
public ICounter copy(ICounter other) {
      if (other != null)
            counter = other.counter();
      return this;
   }
```

Next we update the line in the `Main.main()` method that creates the second instance:

```
counter2 = CounterFactory.newInstance().copy(counter2);
```

Now waiting for a few iterations yields:

```
1) Version 1 = 3
2) Version 1 = 3
```

And changing `Counter.message()` method to return "Version 2" yields:

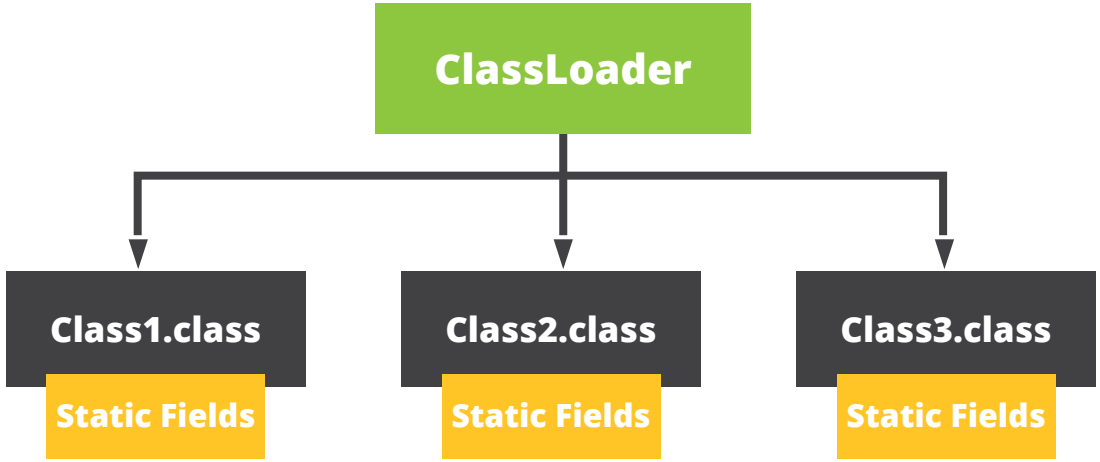
```
1) Version 1 = 4
2) Version 2 = 4
```

As you can see even though it's possible for the end user to see that the second instance is updated and all its state is preserved, it involves managing that state by hand. Unfortunately, there is no way in the Java API to just update the class of an existing object or even reliably copy its state, so we will always have to resort to complicated workarounds.

How Do Classloader Leaks Happen?

If you have programmed in Java for some time you know that memory leaks do happen. Usually it's the case of a collection somewhere with references to objects (e.g. listeners) that should have been cleared, but never were. Classloaders are a very special case of this, and unfortunately, with the current state of the Java platform, these leaks are both inevitable and costly: routinely causing `OutOfMemoryError's` in production applications after just a few redeploys.

We saw before that every object has a reference to its class, which in turn has a reference to its classloader. However we didn't mention that every classloader in turn has a reference to each of the classes it has loaded, each of which holds static fields defined in the class:



This means that

1. If a classloader is leaked it will hold on to all its classes and all their static fields. Static fields commonly hold caches, singleton objects, and various configuration and application states. Even if your application doesn't have any large static caches, it doesn't mean that the framework you use doesn't hold them for you (e.g. Log4J is a common culprit as it's often put in the server classpath). This explains why leaking a classloader can be so expensive.

2. To leak a classloader it's enough to leave a reference to any object, created from a class, loaded by that classloader. Even if that object seems completely harmless (e.g. doesn't have a single field), it will still hold on to its classloader and all the application state. A single place in the application that survives the redeploy and doesn't clean up properly is enough to sprout the leak. In a typical application there will be several such places, some of them almost impossible to fix due to the way third-party libraries are built. Leaking a classloader is therefore, quite common.

Introducing the LEAK

We will use the exact same `Main` class as before to show what a simple leak could look like:

```
public class Main {
private static ICounter counter1;
  private static ICounter counter2;

  public static void main(String[] args)  {
    counter1 = CounterFactory.newInstance().copy();

    while (true) {
      counter2 = CounterFactory.newInstance().copy();

      System.out.println("1) " +
        counter1.message() + " = " + counter1.plusPlus());
      System.out.println("2) " +
        counter2.message() + " = " + counter2.plusPlus());
      System.out.println();

      Thread.currentThread().sleep(3000);
    }
  }
}
```

The `CounterFactory` class is also exactly the same, but here's where things get leaky. Let's introduce a new class called `Leak` and a corresponding interface `ILeak`:

```
interface ILeak {
}

public class Leak implements ILeak {
  private ILeak leak;

  public Leak(ILeak leak) {
    this.leak = leak;
  }
}
```

As you can see it's not a terribly complicated class: it just forms a chain of objects, with each doing nothing more than holding a reference to the previous one. We will modify the `Counter` class to include a reference to the `Leak` object and throw in a large array to take up memory (it represents a large cache). Let's omit some methods shown earlier for brevity:

```
public class Counter implements ICounter {
  private int counter;
  private ILeak leak;

  private static final long[] cache = new long[1000000];

  /* message(), counter(), plusPlus() impls */

  public ILeak leak() {
    return new Leak(leak);
  }

  public ICounter copy(ICounter other) {
    if (other != null) {
      counter = other.counter();
      leak = other.leak();
    }
    return this;
  }
}
```

The important things to note about the Counter class are:

1. `Counter` holds a reference to `Leak`, but `Leak` has no references to `Counter`.
2. When `Counter` is copied (method `copy()` is called) a new `Leak` object is created holding a reference to the previous one.

If you try to run this code an `OutOfMemoryError` will be thrown after just a few iterations:

```
Exception in thread "main" java.lang.OutOfMemoryError:
Java heap space at example.Counter.<clinit>(Counter.
java:8)
```

With the right tools, we can look deeper and see how this happens.

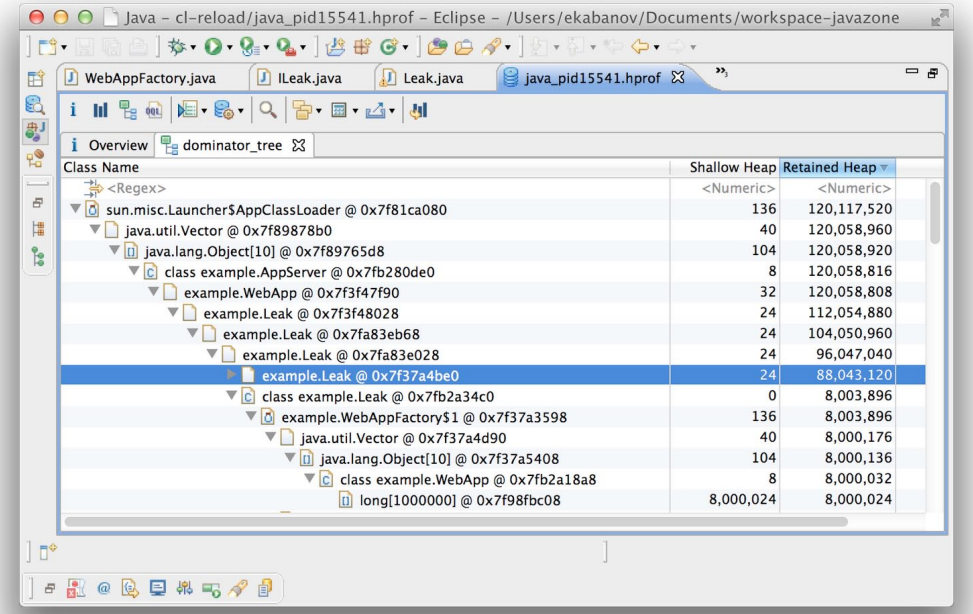
Post Mortem Leak Analysis

Since Java 5.0, we've been able to use the `jmap` command line tool included in the JDK distribution to dump the heap of a running application (or for that matter even extract the Java heap from a core dump). However, since our application is crashing we will need a feature that was introduced in Java 6.0: dumping the heap on `OutOfMemoryError`. To do that we only need to add `-XX:+HeapDumpOnOutOfMemoryError` to the JVM command line:

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid37266.hprof ...
Heap dump file created [57715044 bytes in 1.707 secs]
Exception in thread "main" java.lang.OutOfMemoryError:
Java heap space
    at example.Counter.<clinit>(Counter.java:8)
```

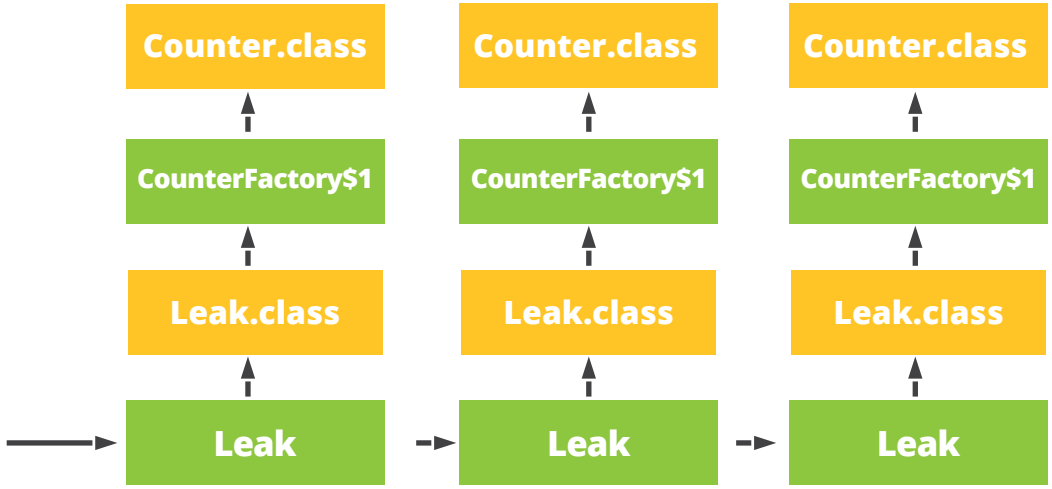
After we have the heap dump we can analyze it. There are a number of tools (including jhat, a small web-based analyzer included with the JDK), but here we will use the more sophisticated Eclipse Memory Analyzer (MAT).

After loading the heap dump into the MAT we can look at the Dominator Tree analysis. It is a very useful analysis that will usually reliably identify the biggest memory consumers in the heap and what objects hold a reference to them. In our case it seems quite obvious that the `Leak` class is the one that consumes most of the heap:



You may notice that one of the intermediate objects is `CounterFactory$1`, which refers to the anonymous subclass of `URLClassLoader` we created in the `CounterFactory` class. In fact what is happening is exactly the situation we described in the beginning of the article:

- Each Leak object is leaking. They are holding on to their classloaders
- The classloaders are holding onto the `Counter` class they have loaded:



This example is slightly contrived, the main idea to take away is that it's easy to leak a single object in Java. Each leak has the potential to leak the whole classloader if the application is redeployed or otherwise a new classloader is created. Preventing such leaks is very challenging and solving them in post-mortem does not prevent new leaks from being introduced.

Dynamic classloading in Java EE Web Applications

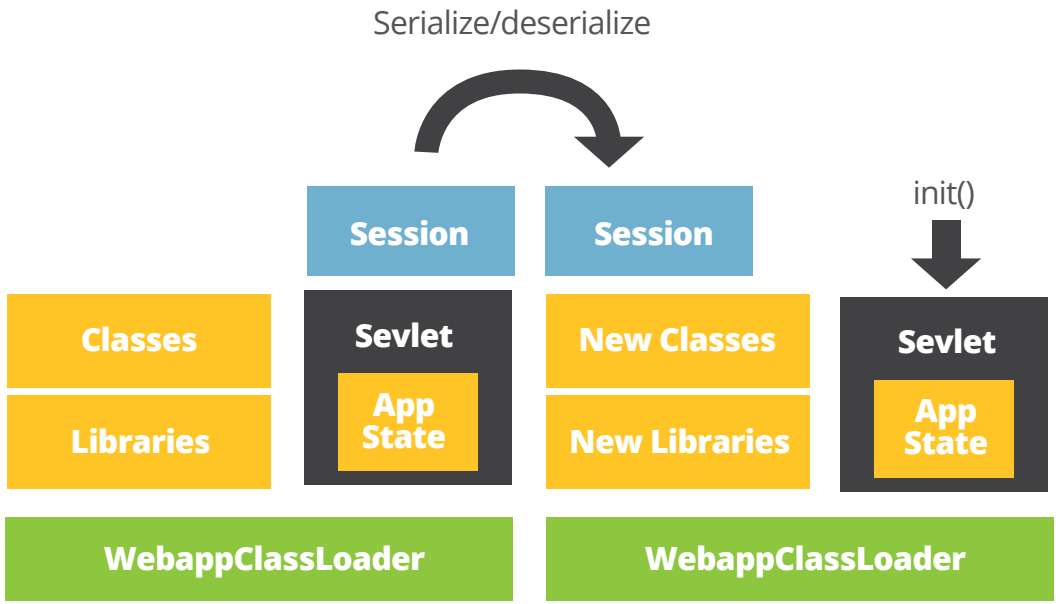
In order for a Java EE web application to run, it has to be packaged into an archive with a .WAR extension and deployed to a servlet container like Tomcat. This makes sense in production, as it gives you a simple way to assemble and deploy the application, but when developing that application you usually just want to edit the application's files and see the changes in the browser.

We already examined how dynamic classloaders can be used to reload Java classes and applications. In this part, we will take a look at how servers and frameworks use dynamic classloaders to speed up the development cycle. We'll use Apache Tomcat as the primary example (Tomcat is also directly relevant for JBoss and GlassFish as these containers embed Tomcat as the servlet container).

To make use of dynamic classloaders we must first create them. When deploying your application, the server will create one classloader for each application (and each application module in the case of an enterprise application).

In Tomcat each .WAR application is managed by an instance of the `StandardContext` class that creates an instance of `WebappClassLoader` used to load the web application classes. When a user presses "reload" in the Tomcat Manager the following will happen:

- `StandardContext.reload()` method is called
- The previous `WebappClassLoader` instance is replaced with a new one
- All reference to servlets are dropped
- New servlets are created
- `Servlet.init()` is called on them



Calling `Servlet.init()` recreates the "initialized" application state with the updated classes loaded using the new classloader instance. The main problem with this approach is that to recreate the "initialized" state we run the initialization from scratch, which usually includes loading and processing metadata/ configuration, warming up caches, running all kinds of checks and so on. In a sufficiently large application this can take many minutes, but in a small application this often takes just a few moments and is fast enough to seem instant.

This is very similar to the `Counter` example we discussed previously and is subject to the same issues. Most applications deployed in production will run out of memory after a few redeploys. This is due to the many connections between application container, libraries and Java core classes that are necessary for the application to run. If even one of those connections is not proactively cleaned up, the web application will not be garbage collected.

Modern Classloaders

Having a hierarchy of classloaders is not always enough. Modern classloaders including OSGi take the approach of having one classloader per module (typically a jar file). All classloaders are siblings, may refer to each other, and share a central repository. Each module declares the packages it imports (from siblings) and exports (to siblings). Given a package name, the common repository is able to find relevant classloaders.

Assuming that all packages must be either imported or exported, a simplified modular classloader might look like this:

```
class MClassLoader extends ClassLoader {
 // Initialized during startup from imports
 Set<String> imps;

 public Class loadClass(String name) {
   String pkg = name.substring(0, name.lastIndexOf('.'));

   if (!imps.contains(pkg))
     return null;

   return repository.loadClass(name);
 }
}

class MRepository {
 // Initialized during startup from exports
 Map<String, List<MClassLoader>> exps;

 public Class loadClass(String name) {
   String pkg = name.substring(0, name.lastIndexOf('.'));

   for (MClassLoader cl : exps.get(pkg)) {
     Class result = cl.loadLocalClass(name);
     if (result != null)
       return result;
   }

   return null;
 }
}
```

Troubleshooting issues with modern classloaders is similar as with a classloader hierarchy: `ClassLoader.getResource()` and `-verbose:class` still help. But now you need to think in terms of export/import as well as the classpath.

There are problems with the modern approach. Firstly, `import` is a one-way street - if you want to use Hibernate, you import it, but it cannot access your classes. Secondly, leaking is perhaps even more problematic: the more classloaders, the more references between them, the easier leaking becomes. Thirdly, deadlocks may happen as the JVM enforces a global lock on `loadClass()` (this is fixed with the parallel classloader in Java SE 7).

How can we fix it?

We need to realize that isolating modules or applications through classloaders is an illusion: leaks can and will happen. The natural abstraction for isolation is a process--this is understood widely outside of Java: .NET, dynamic languages and even browsers use processes for isolation.

A separate process for each application is the only approach to isolation that supports leakless updates. Processes isolate memory space and you simply can't have references between processes. Thus, if the new version of the application is a new process, it cannot contain leaks from the previous version.

The state of the art solution for Java EE production application is running on web application per application container and using rolling restarts with session draining to ensure that the new version will start free of any side effects from the previous one. ZeroTurnaround embraces this approach in our Continuous Delivery tool LiveRebel, which can automatically run no downtime rolling restarts for your web applications, besides helping you with configuration management and other aspects of application delivery.

CONCLUSION

There a lot more to classloaders that we haven't covered here, but if you were to remember just a few key things from this report, this is what they should be:

Validate classloading assumptions. Honestly! Normal intuition just doesn't work very well when classloading is concerned. So be sure to methodically check through every possibility and validate where every class is (or isn't) coming from. The cheat sheet should help you with the common problems. (on page 12)

Classloaders leak. Or to leak a classloader, it's enough to leak any object of a class loaded in that classloader. The naked truth is that classloaders do not impose any isolation guarantees and thus most applications will leak classloaders. Even modern solutions like OSGi are not immune to this issue.

Processes work. Well, they do. Processes provide strong isolation guarantees by separating memory space and most other platforms outside Java are using them for application isolation. There isn't a good reason why we shouldn't do the same.

Rebel Labs is the research & content division of ZeroTurnaround

Contact Us

labs@zeroturnaround.com

Estonia
Ülikooli 2, 5th floor
Tartu, Estonia, 51003
Phone: +372 740 4533

USA
545 Boylston St., 4th flr.
Boston, MA, USA, 02116
Phone: 1(857)277-1199

This report is brought to you by:
Jevgeni Kabanov, Anton Arhipov, Erkki Lindpere, Pavel Grigorenko, Ryan St. James and Oliver White