A JavaGeeks.com
White Paper

Ted Neward
http://www.javageeks.com/~tneward
tneward@javageeks.com
Last modified on 12 June, 2000

Java statics

When is a static *not* static?

Java statics

When is a static *not* a static?

Abstract

Java programmers, like C++ programmers, expect that static instances (namely, static fields) are unique within the JVM. While true for most Java code, the real story behind the "static" keyword is more complex than that. What's worse, it can rear its head to bite Java programmers in very sensitive areas in complex environments like a servlet container or EJB server.

Problem Discussion

Ask any Java programmer what "static" means as a keyword within the Java language, and they'll toss back an answer along the lines of "statics mean that only one instance exists across all instances of the class" or that "static means you have no *this* pointer". In fact, this is a true statement, *most* of the time.

The Java Language Specification (1.0) backs this idea up entirely. Quoting directly from section 8.3.1.1 of the JLS, it reads:

> 8.3.1.1 static *Fields*
>
> If a field is declared static, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a *class variable*, is incarnated when the class is initialized (§12.4).

So, in fact, the Java Language Specification agrees with the Java programmers of the world, claiming that there exists "exactly *one* incarnation of the field". What's more, running this code:

```
// Dummy.java
// ----------
/**
 * Simple class that just prints the number of instances that have been
 * created thus far.
 */
public class Dummy
{
    public static int staticCount = 0;

    public Dummy()
    {
        staticCount++;
        System.out.println("Instance #" + staticCount + " constructed.");
    }
}
```

```
// StaticTest.java
// ---------------
import java.io.*;
import java.net.*;

/**
 * Simple test--instantiate three Dummy objects, and watch the  staticCount
 * static field get incremented, as we would expect.
 */
public class StaticTest
{
    public static void main (String args[])
        throws Exception
    {
        new Dummy();
        new Dummy();
        new Dummy();
    }
}
```

produces a result that most Java programmers will comfortably predict with exacting accuracy:

```
C:\Projects\Papers\JavaStatics\src>java  StaticTest
Instance #1 constructed.
Instance #2 constructed.
Instance #3 constructed.

C:\Projects\Papers\JavaStatics\src>
```

Three Dummy instances were created—three constructors fired, and each constructor in turn incremented the "exactly *one* incarnation of the field" called *Dummy.staticCount*.

So why is it, then, that *this* code produces such radically different results[1]?

```
// StaticTestClassLoader.java
// --------------------------
import java.io.*;
import java.net.*;

/**
 * Load the same class (with statics) into two separate ClassLoaders.
 * See what the class' reported static count is.
 */
public class StaticTestClassLoader
{
    public static void main (String args[])
        throws Exception
    {
        URL[] url = { new File("subdir").toURL() };

        URLClassLoader cl1 = new URLClassLoader(url);
        URLClassLoader cl2 = new URLClassLoader(url);
        URLClassLoader cl3 = new URLClassLoader(url);

        cl1.loadClass("Dummy").newInstance();
        cl2.loadClass("Dummy").newInstance();
        cl3.loadClass("Dummy").newInstance();
    }
}

C:\Projects\Papers\JavaStatics\src>md  subdir
C:\Projects\Papers\JavaStatics\src>move  Dummy.class subdir
```

[1] To make this test work as shown, you *must* move the Dummy.class file into a subdirectory called "subdir" that does *not* reside on the CLASSPATH. I explain this later in the paper, trust me.

```
C:\Projects\Papers\JavaStatics\src>java  StaticTestClassLoader
Instance #1 constructed.
Instance #1 constructed.
Instance #1 constructed.

C:\Projects\Papers\JavaStatics\src>
```

What happened?

Solution Discussion

To answer the question quite technically, ClassLoaders happened. Or, more accurately, *multiple* ClassLoaders happened. At first glance, this may seem to be a horrendous bug; in fact, this is exactly the behavior specified by the Java Virtual Machine Specification, although it takes a bit of reading to "get it" entirely.

The key point centers, as you might guess from reading the above "broken" code, around the Java ClassLoader mechanism. (If you are unfamiliar with ClassLoaders, you may want to brush up on the concepts in ClassLoaders [1, 2, 3] before proceeding.) The key relevant point in the Java Language Specification comes in Section 12.1.1, talking about the processes a class goes through when loaded into a ClassLoader.

To start, we examine the third paragraph of section 12.1.2:

> Preparation involves allocation of static storage and any data structures that are used internally by the virtual machine, such as method tables. If a problem is detected during preparation, then an error is thrown. Preparation is described further in §12.3.2.

The *preparation* step of loading a class is the point at which static allocation is made (initialization of static fields to predefined values occurs later, but that's irrelevant to this discussion). This, then, is when statics are created for the class. Unfortunately, what's *not* stated clearly in the Specification is that these statics are created on a *per ClassLoader instance basis*.

Bracha and Liang, in their paper on ClassLoaders at the OOPSLA '98 conference [3], discuss the notion of ClassLoaders providing multiple namespaces, areas which conceptually isolate different versions of the same class from one another. This allows Java to load different versions of the same class into the same JVM, so long as they are loaded through different ClassLoaders.

This is, in fact, exactly what the StaticTestClassLoader example from above does—it creates three separate URLClassLoader instances, each of which looks for the Dummy class, and creates a new instance. When each URLClassLoader instance is asked to load the Dummy class, it first delegates to its parent ClassLoader[2], the AppClassLoader. AppClassLoader, which is responsible for loading code from the CLASSPATH, reports (after checking with its parent, and so on up the ClassLoader tree) that it cannot find the class, and the new URLClassLoader instance checks itself. It knows about the "subdir" directory in which the "Dummy.class" file resides, loads the code, and creates the static *staticCount* instance for the Dummy class. A new instance of Dummy is created, and the Dummy constructor reports its *staticCount* value, which at this point is "1".

[2] All of this is covered in excrutiating detail in [1] and [2].

Now we move to the second line of StaticTestClassLoader, which again creates a new URLClassLoader to load Dummy. Once again, URLClassLoader delegates to its parent, AppClassLoader, who again reports that it has no idea where the "Dummy" class lives. Remember, it was the first URLClassLoader that loaded the Dummy class a few seconds ago, and ClassLoaders never check with peer ClassLoaders, only parents. Thus, the second URLClassLoader loads the Dummy clas, and as part of that loading step, prepares the class all over again—which means it once again allocates space for the static *staticCount* field! A new instance of Dummy is created, the Dummy constructor increments *cl2*'s *Dummy.staticCount* field, and reports its value: "1".

I could belabor the point by walking through this exercise again for URLClassLoader *cl3*, but by now you should see the problem—statics are static only until we reach ClassLoader boundaries. Normally, were Dummy to be found along the CLASSPATH[3], it would be picked up by the AppClassLoader, of which we always have just one, and the statics would have appeared to have been static across the entire VM. As soon as we introduced multiple ClassLoaders, however, we complicated the picture, and got exactly the response you saw earlier.

At this point, most Java programmers will be experiencing one of two possible reactions:

1. "Why do I care?", or

2. "*Why* doesn't Sun fix this obviously broken implementation?"

I'll answer the first reaction in just a second (but bear with me—you *do* care). As to the second reaction, the honest response is "It's not a bug, it's a feature". By associating static instances to the ClassLoader that loaded it, it's possible to do the "on-the-fly" upgrading of code mentioned in [1] and [3]. That is, if the Dummy class were to change on disk between the *cl1.loadClass("Dummy").newInstance()* and corresponding *cl2* calls, then *cl2* would actually get the new version of Dummy, even while the old version peacefully coexisted within the JVM in *cl1*. This has powerful implications for creating Java applications that can be safely upgraded on the server without taking the server down, making your system administrators that much closer to achieving "Five-Nines" availability[4]. Without this feature, servlets couldn't be reloaded "on the fly".

Which brings us back to point #1: you *do* care. You care, because two *very* important environments, servlets and EJB, both make use of ClassLoaders in a big way.

Consider the ubiquitous servlet environment, in which the servlet container supports "servlet reloading"—that is, change the servlet class on disk and the servlet container will automatically reload the servlet, even while old versions of the servlet are still executing (presumably in other Threads). The only way a servlet container can make this work is to load the new servlet into a different ClassLoader than the original version.

This is where "you *do* care" comes into play. Most servlet classes that access a relational database typically want to minimize the number of Connections they make to the database. In order to best support this, most servlets will make use of a Singleton [4] ConnectionManager to share Connections across multiple servlets:

---

[3] Which is why you had to move Dummy.class into the "subdir" subdirectory, or else the AppClassLoader would have found it along the CLASSPATH, which typically includes the current directory.

[4] "Five-Nines", which is talked about in [1], is the idea of having servers available (that is, *not* down for repairs, maintenance or upgrade) 99.999% of the time. .001% of a year is just about five minutes—it's an incredibly ambitious goal, but definitely within the realm of possibility—telephone companies hit this metric relatively often.

```
/**
 *
 */
public class DatabaseManager
{
    /**
     * Marked private to prevent accidental  instantiation
     */
    private DatabaseManager()
    {
        // Do the usual thing here
    }

    /**
     * Gain access to the singleton  DatabaseManager instance
     */
    public DatabaseManager instance()
    {
        return staticInstance;
    }
    private static DatabaseManager staticInstance =
        new DatabaseManager();

    // . . .
}
```

Notice that this Singleton instance relies on the static method *instance*—and the associated static field *staticInstance*—to ensure that only one instance of ConnectionManager exists.

Here's the part where you care: when the servlet container loads your servlet into its own ClassLoader (call it *scl1*, an instance of the servlet container's class ServletClassLoader), the ClassLoader notices that your servlet code references a class called DatabaseManager. Because classes (a) remember the ClassLoader that loaded them, and (b) default to using that ClassLoader to loading any dependent code, *scl1* is asked to load DatabaseManager.

By this point, the problem starts becoming obvious—because *scl1* is its own ClassLoader, the static *staticInstance* field exists only as far as *scl1* can reach. When a new version of your servlet is loaded[5], the servlet container will create a new ServletClassLoader instance (call it *scl2*), load your servlet, notice that your servlet references DatabaseManager, look to load DatabaseManager, and get fresh copies of the static *staticInstance* field. Suddenly, a Singleton isn't a Singleton anymore, and twice as many Connection objects are being created.

Solution

Unfortunately, the "solution" here isn't much of a solution—Sun isn't going to change this part of ClassLoaders anytime soon, if at all[6]. So the only real solution is knowledge: be aware of this, and take steps to lessen its impact on your code where possible.

Within the servlet environment (and EJB, since EJB servers will create their own ClassLoaders for much the same reason servlet containers do), however, you do have a more practical solution: move the DatabaseManager code someplace where a

[5] Or, in fact, when a new servlet request comes in—in order to support seamless versioning of servlets, a servlet container may run *each and every servlet request* in its own ClassLoader, although this is obviously not the most efficient way to handle the situation.

[6] Actually, as you might have guessed, I'm hoping they never do—this is an incredibly powerful feature.

ClassLoader further up the delegation chain will find it. This means one of two places: putting the DatabaseManager class somewhere on the CLASSPATH (so it gets picked up by the AppClassLoader), or else put it into the Extensions directory (so it gets picked up by the ExtClassLoader). Because the ServletClassLoader will always[7] delegate to its parent ClassLoader, by placing DatabaseManager into the AppClassLoader or ExtClassLoader namespace, its static *staticInstance* will appear to stretch across the entire JVM and thereby use only one instance of DatabaseManager.

An an exercise to prove this, go back to the StaticTestClassLoader example from a few pages back, and re-run it, this time with Dummy.class in the current directory (where it will get picked up by the AppClassLoader instead of the individual URLClassLoader instances in the *main* method):

```
C:\Projects\Papers\JavaStatics\src>move  subdir\Dummy.class .

C:\Projects\Papers\JavaStatics\src>java  StaticTestClassLoader
Instance #1 constructed.
Instance #2 constructed.
Instance #3 constructed.

C:\Projects\Papers\JavaStatics\src>
```

Exactly what you would have *expected* to see, six pages ago, because now Dummy has been loaded into the AppClassLoader namespace.

Summary

By this point, it's obvious that statics really aren't—they're static only as far as the ClassLoader boundaries stretch. Multiple ClassLoaders will defeat the "static-ness" of fields across the same version of the same class over multiple ClassLoaders, much to the surprise of most Java developers.

Unfortunately, this has some frightening implications: it means, most of all, that Singletons really aren't Singletons, either. If the Singleton is loaded into a ClassLoader that doesn't stretch across the entire VM (and, to be quite honest, the only ClassLoader that is guaranteed to stretch across the *entire* VM is the bootstrap ClassLoader, the parent to the ExtClassLoader), then the Singleton may in fact end up with multiple instances. If the Singleton in turn loads native code, you're in for a rough time, because JNI in Java2 doesn't allow a native library to be loaded more than once, even across ClassLoader boundaries.

The best solution, then, is to simply be aware of this, and try whenever possible to make sure that Singleton classes (like DatabaseManager) get loaded as far up the ClassLoader tree as possible. Fortunately, this is a problem that won't rear its ugly head very often, and is usually pretty quickly solvable ("Drop the .jar file into the jre/lib/ext directory") in most environments.

Bibliography

[1] *Server-Based Java Programming*, by Ted Neward

[7] Assuming it's written in conformance with the Java2 ClassLoader model, which is *not* always a safe assumption—see [1] for details.

[2] *Inside the Java2 Virtual Machine*, by Bill Venners

[3] “”, by Sheng Liang and Gilad Bracha, OOPSLA ’98

[4] *Design Patterns*, by Erich Gamma, Richard Helm, Richard Johnson, and John Vlissides