Java programming dynamics, Part 1: Java classes and class loading

A look at classes and what goes on as they're loaded by a JVM

Dennis Sosnoski

April 29, 2003

Take a look at what goes on behind the scenes of executing your Java application in this new series on the dynamic aspects of Java programming. Enterprise Java expert Dennis Sosnoski gives the scoop on the Java binary class format and what happens to classes inside the JVM. Along the way, he covers class loading issues ranging from the number of classes required for running a simple Java application to the class loader conflicts that can cause problems in J2EE and similar complex architectures.

View more content in this series

This article kicks off a new series covering a family of topics that I call *Java programming dynamics*. These topics range from the basic structure of the Java binary class file format, through run-time metadata access using reflection, all the way to modifying and constructing new classes at run time. The common thread running through all this material is the idea that programming the Java platform is much more dynamic than working with languages that compile straight to native code. If you understand these dynamic aspects, you can do things with Java programming that can't be matched in any other mainstream programming language.

In this article, I cover some of the basic concepts that underlie these dynamic features of the Java platform. These concepts revolve around the binary format used to represent Java classes, including what happens when these classes are loaded into the JVM. Not only does this material provide a foundation for the rest of the articles in the series, it also demonstrates some very practical concerns for developers working on the Java platform.

A class in binary

Developers working in the Java language normally don't need to be concerned about the details of what happens to their source code when it's run through a compiler. In this series, I'm covering a lot of behind-the-scenes details involved in going from source code to executing program, though, so I'll start with a look at the binary classes generated by a compiler.

The binary class format is actually defined by the JVM specification. Normally these class representations are generated from Java language source code by a compiler, and they're usually stored in files with a `.class` extension. Neither of these features is essential, though. Other programming languages have been developed that use the Java binary class format, and for some purposes new class representations are constructed and immediately loaded within an executing JVM. As far as a JVM is concerned, the important part is not the source or how it's stored, but the format itself.

So what does this class format actually look like? Listing 1 gives the source code for a (very) short class, along with a partial hexadecimal display of the class file output by the compiler:

Listing 1. Source and (partial) binary for Hello.java

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

0000: cafe babe 0000 002e 001a 0a00 0600 0c09  ................
0010: 000d 000e 0800 0f0a 0010 0011 0700 1207  ................
0020: 0013 0100 063c 696e 6974 3e01 0003 2829  .....<init>...()
0030: 5601 0004 436f 6465 0100 046d 6169 6e01  V...Code...main.
0040: 0016 285b 4c6a 6176 612f 6c61 6e67 2f53  ..([Ljava/lang/S
0050: 7472 696e 673b 2956 0c00 0700 0807 0014  tring;)V........
0060: 0c00 1500 1601 000d 4865 6c6c 6f2c 2057  ........Hello, W
0070: 6f72 6c64 2107 0017 0c00 1800 1901 0005  orld!...........
0080: 4865 6c6c 6f01 0010 6a61 7661 2f6c 616e  Hello...java/lan
0090: 672f 4f62 6a65 6374 0100 106a 6176 612f  g/Object...java/
00a0: 6c61 6e67 2f53 7973 7465 6d01 0003 6f75  lang/System...ou
...
```

Inside the binary

The very first thing in the binary class representation shown in Listing 1 is the "cafe babe" signature that identifies the Java binary class format (and incidentally serves as a lasting -- but largely unrecognized -- tribute to the hard working *baristas* who kept up the spirits of the developers building the Java platform). This signature is just an easy way of verifying that a block of data really *does* claim to be an instance of the Java class format. Every Java binary class, even one that isn't present on the file system, needs to start with these four bytes.

Don't miss the rest of this series

Part 2, "Introducing reflection" (June 2003)
Part 3, "Applied reflection" (July 2003)
Part 4, "Class transformation with Javassist" (September 2003)
Part 5, "Transforming classes on-the-fly" (February 2004)
Part 6, "Aspect-oriented changes with Javassist" (March 2004)
Part 7, "Bytecode engineering with BCEL" (April 2004)
Part 8, "Replacing reflection with code generation" (June 2004)

The rest of the data is less entertaining. Following the signature are a pair of class format version numbers (in this case, for minor version 0 and major version 46 -- 0x2e in hexadecimal -- as generated by the 1.4.1 javac), then a count of entries in the constant pool. The entry count (in this case 26, or 0x001a) is followed by the actual constant pool data. This is where all the constants used by the class definition are stored. It includes class and method names, signatures, and strings (which you can recognize in the text interpretation to the right of the hexadecimal dump), along with various binary values.

Items in the constant pool are variable length, with the first byte of each item identifying the type of item and how it should be decoded. I won't go into the details of all that here -- there are many references available if you're interested, starting with the actual JVM specification. The key point is just that the constant pool contains all the references to other classes and methods used by this class, along with the actual definitions for this class and its methods. The constant pool can easily make up half or more of the binary class size, though the average proportion is probably less.

Following the constant pool are several items that reference constant pool entries for the class itself, its super class, and interfaces. These items are followed by information about the fields and methods, which are themselves represented as complex structures. The executable code for methods is present in the form of *code attributes* contained within the method definitions. This code is in the form of instructions for the JVM, generally called *bytecode*, which is one of the topics for the next section.

Ask the expert: Dennis Sosnoski on JVM and bytecode issues

For comments or questions about the material covered in this article series, as well as anything else that pertains to Java bytecode, the Java binary class format, or general JVM issues, visit the JVM and Bytecode discussion forum, moderated by Dennis Sosnoski.

Attributes are used for several defined purposes in the Java class format, including the already-mentioned bytecode, constant values for fields, exception handling, and debugging information. These purposes aren't the only possible uses for attributes, though. From the beginning, the JVM specification has required JVMs to ignore attributes of unknown types. This requirement gives flexibility for extending the use of attributes to serve other purposes in the future, such as providing meta-information needed by frameworks that work with user classes -- an approach that the Java-derived C# language has used extensively. Unfortunately, no hooks have yet been provided for making use of this flexibility at the user level.

Bytecode and stacks

The bytecode that makes up the executable portion of the class file is actually machine code for a special kind of computer -- the JVM. This is called a *virtual* machine because it's been designed for implementation in software, rather than hardware. Every JVM used to run Java platform applications is built around an implementation of this machine.

This virtual machine is actually fairly simple. It uses a stack architecture, meaning instruction operands are loaded to an internal stack before they're used. The instruction set includes all the normal arithmetic and logical operations, along with conditional and unconditional branches, load/store, call/return, stack manipulation, and several special types of instructions. Some of the instructions include immediate operand values that are directly encoded into the instruction. Others directly reference values from the constant pool.

Even though the virtual machine is simple, the implementations aren't necessarily so. Early (first generation) JVMs were basically interpreters for the virtual machine bytecode. These actually *were* relatively simple, but suffered from severe performance problems -- interpreting code is always going to take longer than executing native code. To reduce these performance problems, second generation JVMs added *just-in-time* (JIT) translation. The JIT technique compiles Java bytecode to native code before executing it for the first time, giving much better performance for repeated executions. Current generation JVMs go even further, using adaptive techniques to monitor program execution and selectively optimize heavily used code.

Loading the classes

Languages such as C and C++ that compile to native code generally require a linking step after source code is compiled. This linking process merges code from separately compiled source files, along with shared library code, to form an executable program. The Java language is different. With the Java language, the classes generated by the compiler generally remain just as they are until they're loading into a JVM. Even building a JAR file from the class files doesn't change this -- the JAR is just a container for the class files.

Rather than a separate step, linking classes is part of the job performed by the JVM when it loads them into memory. This adds some overhead as classes are initially loaded, but also provides a high level of flexibility for Java applications. For example, applications can be written to use interfaces with the actual implementations left unspecified until run time. This *late binding* approach to assembling an application is used extensively in the Java platform, with servlets being one common example.

The rules for loading classes are spelled out in detail in the JVM specification. The basic principle is that classes are only loaded when needed (or at least appear to be loaded this way -- the JVM has some flexibility in the actual loading, but must maintain a fixed sequence of class initialization). Each class that gets loaded may have other classes that it depends on, so the loading process is recursive. The classes in Listing 2 show how this recursive loading works. The `Demo` class includes a simple `main` method that creates an instance of `Greeter` and calls the `greet` method. The `Greeter` constructor creates an instance of `Message`, which it then uses in the `greet` method call.

Listing 2. Source code for class loading demonstration

```
public class Demo
{
    public static void main(String[] args) {
        System.out.println("**beginning execution**");
        Greeter greeter = new Greeter();
        System.out.println("**created Greeter**");
        greeter.greet();
    }
}

public class Greeter
```

```
{
    private static Message s_message = new Message("Hello, World!");

    public void greet() {
        s_message.print(System.out);
    }
}

public class Message
{
    private String m_text;

    public Message(String text) {
        m_text = text;
    }

    public void print(java.io.PrintStream ps) {
        ps.println(m_text);
    }
}
```

Setting the parameter `-verbose:class` on the `java` command line prints a trace of the class loading process. Listing 3 shows partial output from running the Listing 2 program with this parameter:

Listing 3. Partial -verbose:class output

```
[Opened /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/sunrsasign.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/jsse.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/jce.jar]
[Opened /usr/java/j2sdk1.4.1/jre/lib/charsets.jar]
[Loaded java.lang.Object from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.io.Serializable from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.lang.Comparable from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.lang.CharSequence from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.lang.String from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
...
[Loaded java.security.Principal from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.security.cert.Certificate
  from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded Demo]
**beginning execution**
[Loaded Greeter]
[Loaded Message]
**created Greeter**
Hello, World!
[Loaded java.util.HashMap$KeySet
  from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
[Loaded java.util.HashMap$KeyIterator
  from /usr/java/j2sdk1.4.1/jre/lib/rt.jar]
```

This is only a partial listing of the most important parts -- the full trace consists of 294 lines, most of which I deleted for this listing. The initial set of class loads (279, in this case) are all triggered by the attempt to load the `Demo` class. These are the core classes that are used by every Java program, no matter how small. Even eliminating all the code from the `Demo main` method doesn't affect this initial sequence of loads. The number and names of classes involved will differ from one version of the class libraries to another, though.

The portion of the listing after the `Demo` class is loaded is more interesting. The sequence here shows that the `Greeter` class is only loaded when an instance of the class is about to be created.

However, the `Greeter` class uses a static instance of the `Message` class, so before an instance of the former can be created, the latter class also needs to be loaded.

A lot happens inside the JVM when a class is loaded and initialized, including decoding the binary class format, checking compatibility with other classes, verifying the sequence of bytecode operations, and finally constructing a `java.lang.Class` instance to represent the new class. This `Class` object becomes the basis for all instances of the new class created by the JVM. It's also the identifier for the loaded class itself -- you can have multiple copies of the same binary class loaded in a JVM, each with its own `Class` instance. Even though these copies all share the same class name, they will be separate classes to the JVM.

Off the beaten (class) path

Class loading in a JVM is controlled by *class loaders*. There's a *bootstrap* class loader built into the JVM that's responsible for loading the basic Java class library classes. This particular class loader has some special features. For one thing, it only loads classes found on the boot class path. Because these are trusted system classes, the bootstrap loader skips much of the validation that gets done for normal (untrusted) classes.

Bootstrap isn't the only class loader. For starters, a JVM defines an *extension* class loader for loading classes from standard Java extension APIs, and a *system* class loader for loading classes from the general class path (including your application classes). Applications can also define their own class loaders for special purposes (such as run-time reloading of classes). Such added class loaders are derived from the `java.lang.ClassLoader` class (possibly indirectly), which provides the core support for building an internal class representation (a `java.lang.Class` instance) from an array of bytes. Each constructed class is in some sense "owned" by the class loader that loaded it. Class loaders normally keep a map of the classes they've loaded, to be able to find one by name if it's requested again.

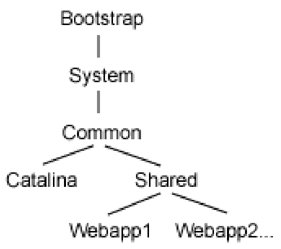
Each class loader also keeps a reference to a parent class loader, defining a tree of class loaders with the bootstrap loader at the root. When an instance of a particular class (identified by name) is needed, whichever class loader initially handles the request normally checks with its parent class loader first before trying to load the class directly. This applies recursively if there are multiple layers of class loaders, so it means that a class will normally be *visible* not only within the class loader that loaded it, but also to all descendant class loaders. It also means that if a class can be loaded by more than one class loader in a chain, the one furthest up the tree will be the one that actually loads it.

There are many circumstances where multiple application classloaders are used by Java programs. One example is within the J2EE framework. Each J2EE application loaded by the framework needs to have a separate class loader to prevent classes in one application from interfering with other applications. The framework code itself will also use one or more other class loaders, again to prevent interference to or from applications. The complete set of class loaders make up a tree-structured hierarchy with different types of classes loaded at each level.

Trees of loaders

As an example of a class loader hierarchy in action, Figure 1 shows the class loader hierarchy defined by the Tomcat servlet engine. Here the Common class loader loads from JAR files in a particular directory of the Tomcat installation that's intended for code shared between the server and all Web applications. The Catalina loader is for Tomcat's own classes, and the Shared loader for classes shared between Web applications. Finally, each Web application gets its own loader for its private classes.

Figure 1. Tomcat class loaders



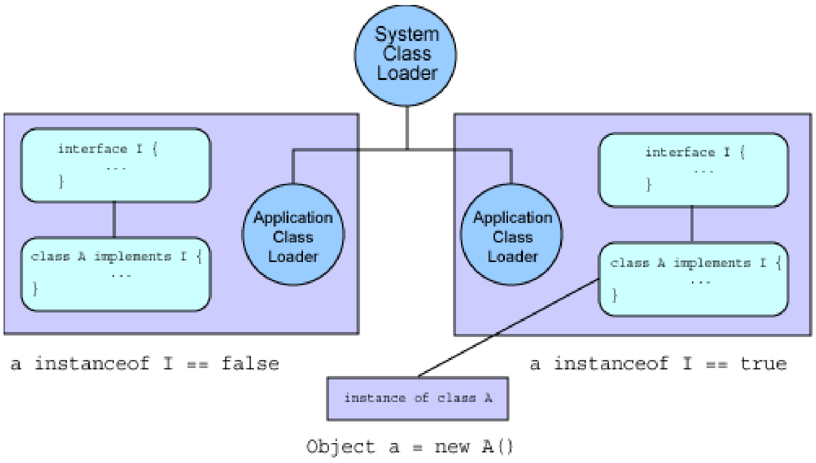
In this type of environment, keeping track of the proper loader to use for requesting a new class can be messy. Because of this, the `setContextClassLoader` and `getContextClassLoader` methods were added to the `java.lang.Thread` class in the Java 2 platform. These methods let the framework set the class loader to be used for each application while running code from that application.

The flexibility of being able to load independent sets of classes is an important feature of the Java platform. Useful as this feature is, though, it can create confusion in some cases. One confusing aspect is the continuing issue of dealing with JVM classpaths. In the Tomcat hierarchy of class loaders shown in Figure 1, for instance, classes loaded by the Common class loader will never be able to directly access (by name) classes loaded by a Web application. The only way to tie these together is through the use of interfaces visible to both sets of classes. In this case, that includes the `javax.servlet.Servlet` implemented by Java servlets.

Problems can arise when code is moved between class loaders for any reason. For instance, when J2SE 1.4 moved the JAXP API for XML processing into the standard distribution, it created problems for many environments where applications had previously relied on loading their own chosen implementations of the XML APIs. With J2SE 1.3, this can be done just by including the appropriate JAR file in the user class path. In J2SE 1.4, the standard versions of these APIs are now in the extensions class path, so these will normally override any implementations present in the user class path.

Other types of confusion are also possible when using multiple class loaders. Figure 2 shows an example of a *class identity crisis* that results when an interface and associated implementation are each loaded by two separate class loaders. Even though the names and binary implementations of the interfaces and classes are the same, an instance of the class from one loader cannot be recognized as implementing the interface from the other loader. This confusion could be resolved in Figure 2 by moving the interface class `I` into the System class loader's space. There would still be two separate instances of class `A`, but both would implement the same interface `I`.

Figure 2. Class identity crisis



Conclusions

The Java class definition and JVM specification together define an extremely powerful framework for run-time assembly of code. Through the use of class loaders, Java applications are able to work with multiple versions of classes that would otherwise cause conflicts. The flexibility of class loaders even allows dynamic reloading of modified code while an application continues to execute.

The cost of the Java platform's flexibility in this area is somewhat higher overhead when starting an application. Hundreds of separate classes need to be loaded by the JVM before it can start executing even the simplest application code. This startup cost generally makes the Java platform better suited to long-running, server-type applications than for frequently used small programs. Server applications also benefit the most from the flexibility of run-time assembly of code, so it's no surprise that the Java platform has become increasingly favored for this type of development.

In Part 2 of this series, I'll cover an introduction to using another aspect of the Java platform's dynamic underpinnings: the Reflection API. Reflection gives your executing code access to internal class information. This can be a great tool for building flexible code that can be hooked together at run time without the need for any source code links between classes. But as with most tools, you need to know when and how to use it to best advantage. Check back to find out the tricks and trade-offs of effective reflection in Part 2 of *Java programming dynamics*.

Related topics

- Go straight to the source in *The Java Virtual Machine Specification* for details of the binary class format, loading of classes, and actual Java bytecode.
- Learn all the details of building your own special class loaders with the tutorial, "Understanding the Java ClassLoader" by Greg Travis (*developerWorks*, April 2001).
- Get details on the Jikes Research Virtual Machine (RVM), which is implemented in the Java language and is self-hosted (that is, its Java code runs on itself without requiring a second virtual machine).
- Keep up with progress on making attributes available to Java developers with Java Specification Request (JSR) 175 for A Metadata Facility for the Java Programming Language.
- Find out the details of the Apache Tomcat Java language Web server project of the Apache Software Foundation, including the details of Tomcat class loader usage.