The Working Developer's Guide to Java Bytecode

by Ted Neward

May 2007

Discussion

Introduction

In the last year or two, interest in "alternative" languages on top of the Java Virtual Machine has grown exponentially, to the point where three languages were blessed with official acceptance in the Java Community Process: JRuby (a port of the Ruby programming language), Groovy (a Java-like language with more dynamic features), and Bean-Shell (a more scripting-oriented shell- like language). In addition, the Java6 (code-named "Mustang") release includes an API layer designed to encapsulate the details of various scripting engines away in the same manner JDBC does for database access. Plus, we have all the changes that took place in the Java5 version of the language itself. In short, the Java platform programming language landscape has changed pretty significantly. One thing that hasn't changed, however, is the basic fact that all of these languages, despite all their interesting features or capabilities, eventually end up in the lingua franca of the Java Virtual Machine, the JVM bytecode set. In this article, we're going to examine the JVM bytecode set, disassemble some code to see how it works, and play with some tools that allow us to manipulate bytecode directly.

It may seem counterintuitive to learn the JVM bytecode format; after all, it's not like anyone (not even me) is going to suggest that you write code in bytecode directly, right? That's what high-level languages are for, to allow us to think in more general terms and let tools (like compilers) worry about the precise details. But it's always a good thing to know what the compiler produces: for example, you've probably heard the debates surrounding the use of StringBuffer instead of doing straight String concatenation, or the idea that the compiler will generate code on your behalf in certain situations (such as creating a default constructor for a class if you haven't specified a constructor), but when's the last time you verified those assumptions? Or wondered how inner classes work to ensure that the inner class has access to the private variables of the outer class, without breaking encapsulation for any potential accessor? By looking at JVM bytecode (which, by the way, is probably the simplest "assembly language" you'll ever see in your lifetime), you can verify crucial assumptions and/or debunk persistent myths. Not to mention give you a tool for ensuring that classes running in production are the ones that are supposed to be there (because, of course, deployments into production always go the way they're supposed to, right?).

All of this completely ignores the advantages of using bytecode directly in other scenarios. For example, the java.lang.instrumentation package provides APIs for doing load-time instrumentation of classes (what others might call load-time aspect-oriented weaving of code), but to do it effectively, you need to know the bytecode set so you can know where and what to inject. And sometimes you actually need to reverseengineer obfuscated vendor code in order to fix a bug at the 11th hour before you can ship. And so on, and so on, and so on. So hop on the Java bytecode bandwagon with me for a bit and let's run through the core parts of the JVM bytecode set that you'll see. Hopefully, in the process, you'll discover some things you may not have known about Java and the JVM that will be of use in more practical scenarios.

Disassembling Java 101

While it's always possible to begin our discussion of Java bytecode by simply asserting some facts ("Java bytecode is stored in a tree format"), given that most of the audience is already familiar with Java, it's easier to start by going the opposite direction: we'll take working Java-compiled code, and disassemble it. That way, we've got a working framework (that is, your understanding of the Java language) from which to hang various points about the Java bytecode format. We begin with every Java programmer's first program, the ubiquitous traditional "Hello World" app.

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

Our guide to Java bytecode comes in two forms. The first is a JDK tool that even long-time Java developers have never seen before, javap. javap is a bytecode disassembler, meaning it takes compiled .class files and dumps the file's structure to the console, including the bytecode making up the methods contained in that file. Use it against a class on the CLASSPATH like so:

```
$ javap -verbose -c -private HelloWorld
Compiled from "HelloWorld.java"
public class HelloWorld extends java.lang.Object
        SourceFile: "HelloWorld.java"
        minor version: 0
        major version: 50
        Constant pool:
const #1 = Method #6.#15; // java/lang/Object."<init>":()V
const #2 = Field #16.#17; // java/lang/System.out:Ljava/io/PrintStream;
const #3 = String #18; // Hello, world!
const #4 = Method #19.#20; // java/io/PrintStream.println:(Ljava/lang/String;)V
const #5 = class #21; // HelloWorld
const #6 = class #22; // java/lang/Object
const #7 = Asciz <init>;
const #8 = Asciz ()V;
const #9 = Asciz Code;
const #10 = Asciz LineNumberTable;
const #11 = Asciz main;
const #12 = Asciz ([Ljava/lang/String;)V;
const #13 = Asciz SourceFile;
const #14 = Asciz HelloWorld.java;
const #15 = NameAndType #7:#8;// "<init>":()V
const #16 = class #23; // java/lang/System
const #17 = NameAndType #24:#25;// out:Ljava/io/PrintStream;
const #18 = Asciz Hello, world!;
const #19 = class #26; // java/io/PrintStream
const #20 = NameAndType #27:#28;// println:(Ljava/lang/String;)V
const #21 = Asciz HelloWorld;
const #22 = Asciz java/lang/Object;
const #23 = Asciz java/lang/System;
const #24 = Asciz out;
const #25 = Asciz Ljava/io/PrintStream;;
const #26 = Asciz java/io/PrintStream;
const #27 = Asciz println;
const #28 = Asciz (Ljava/lang/String;)V;

{
public HelloWorld();
        Code:
                Stack=1, Locals=1, Args_size=1
                0: aload_0
                1: invokespecial #1; //Method java/lang/Object."<init>":()V
                4: return

        LineNumberTable:
                line 1: 0
public static void main(java.lang.String[]);
        Code:
                Stack=2, Locals=1, Args_size=1
                0: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                3: ldc #3; //String Hello, world!
                5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                8: return
        LineNumberTable:
                line 5: 0
                line 6: 8
}
```

[The use of the term "class file" is a bit incorrect here, since technically the Java Virtual Machine doesn't care where this stream of binary bits came from, but we refer to the binary format of Java classes like this as "class files" for historical reasons, since the original JDK 1.0 release demanded that they be in independent ".class" files in the directory structure(s) pointed to by the CLASSPATH environment variable. Thus, "class file" should be interpreted to mean "the binary format defined by the Java Virtual Machine
Specification, Second Edition".]

The various flags tell javap to provide more information than its default, scanty, output; "-c" performs method bytecode display, "-private" displays all members regardless of accessibility, and "-verbose" shows us the constant pool of the class, among other things. Examining the disassembled contents of HelloWorld is interesting; in particular, we can verify a few assumptions about the Java compiler right off. For example, it's been well-known for years that if a class doesn't explicitly extend a base class, it automatically extends java.lang.Object, which javap verifies in the first line of output of HelloWorld. Secondly, it's also been well-known that a class must have a constructor, and that if a constructor isn't explicitly written, the compiler inserts a default, no-argument constructor, and again, javap verifies this. (Constructors, at the JVM level, are regular methods that are known by the well-known name "<init>".)

The most obvious factor in the javap output (when "-verbose" is passed in, anyway) is the constant pool output. Every class file has a constant pool in which all constants–particularly strings, be they constant strings in code, class names, method names, or even field names–are stored in a central location in the class, and referenced via indeces into this pool. Normally, this particular detail will be handled by tools– this is why javap displays the constant's value in a comment at the end of the line, for example–but it's still important to realize this constant pool is in place, as it eases understanding the disassembled code. So, for example, we can see that the operation in line 5 of the "main" method invokes a method whose name is stored in slot #4 in the constant pool (which in turn is a method constant consisting of class name (slot #19) and method name (slot #20), which ultimately resolves to "java.io.PrintStream.println(String[])"). For full details of all the different types of constants, as well as the formal description of the .class file format, see the Java Virtual Machine Specification. [The JVM Specification can be purchased from Amazon, or is available in both PDF and HTML formats from the Java website, at "http://java.sun.com/docs/books/vmspec".] (In point of fact, the JVM Specification is going to be the canonical reference for this whole article, so simply assume that as a blanket statement.) The meat of what we want to examine is in the method blocks, and to begin, we're going to examine the autogenerated HelloWorld constructor:

```
public HelloWorld();
        Code:
                Stack=1, Locals=1, Args_size=1
                0: aload_0
                1: invokespecial #1; //Method java/lang/Object."<init>":()V
                4: return
        LineNumberTable:
                line 1: 0
```

In the JVM, all bytecode operates on a basic principle, that of stack manipulation: each opcode may consume one or more operands off the execution stack, [Each of these "slots" are 32 bits wide, meaning that long and double values pushed on to the stack consume two slots each, a decision that is widely regarded as the biggest mistake in the JVM's implementation today.] and may in turn push an operand on to the execution stack as a result. In addition, each method has a collection of locals, where local variables and arguments live. So, for example, the "aload_0" instruction takes the first argument to the method (the "this" pointer implicit in every instance method call) and pushes it to the execution stack. The "invokespecial" instruction, as its name implies, invokes an instance method, but ignores traditional dynamic binding. (This particular opcode is used in all "super" calls from the language, because we specifically want to invoke the base class version of an overridden method.) Because the Object constructor takes one argument (the "this" pointer on which to invoke the method), it consumes one slot off the execution stack (which is the argument we just pushed, the "this" pointer of our own instance, remember), and because it returns no values (as given by the "V" in the end of the signature above), nothing is pushed on the stack when the method returns. At this point, there's nothing more for the HelloWorld constructor to do, so it simply returns via the "return" opcode.

Still with us? Next, let's tackle one other method, HelloWorld's "main" method, reproduced below:

```
public static void main(java.lang.String[]);
        Code:
                Stack=2, Locals=1, Args_size=1
                0: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                3: ldc #3; //String Hello, world!
                5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                8: return
        LineNumberTable:
                line 5: 0
                line 6: 8
```

Since this method is static, one particular difference will be the fact that the first argument won't be a "this" pointer, but other than that it'll look similar to what we saw with the HelloWorld constructor. The first opcode, "getstatic", fetches a static field and pushes its value onto the stack, which in this case is the "System.out" reference, described by constant pool entry #2 and displayed in the javap comment after the opcode. Next, we load the constant string "Hello, world!", stored in constant pool entry #3. With both of those references on the stack, we "invokevirtual" the PrintStream.println(String[]) method; since it expects one argument (plus the initial "this" reference on which to invoke the method), both of those items we just pushed onto the stack are consumed, and since println() returns nothing (again, as described by the "V" at the end of the signature), nothing is left on the stack when we're finished. A simple "return" opcode terminates the method, and we're done.

We'll get a bit more complicated as we go on, but for the most part, understanding Java bytecode is an exercise in understanding how each opcode manipulates the execution stack. If you've ever done any assembly in your history, regardless of CPU, Java's bytecode format will seem a highly simpler version.

Java Bytecode Taxonomy

Fundamentally, the JVM bytecode set is broken into several distinct categories. Rather than go over each bytecode opcode individually, we'll instead discuss the categories, highlight a few more common opcodes in particular, and refer you to the JVM Specification for details on the rest.

Stack manipulation. The easiest set of opcodes to understand are those that simply manipulate the stack directly:

- pop, pop2: Pop the top value of the stack. (pop2 is used if the value is 64 bits wide instead of the usual 32.)
- dup, dup2: Duplicate the top value of the stack, effectively forming a pop/push/push combination. (Again, dup2 is the 64-bit version of dup.)
- const_null pushes the null reference onto the stack.
- bipush pushes a (byte) constant value from -127 to 128 onto the stack.
- sipush pushes a (short) constant value from -32k to +32k onto the stack.
- ldc pushes a constant value from the constant pool onto the stack.
- Xload, where X is one of a, d, f, l or i, pushes a "local" (argument or variable) of the indicated type onto the stack. [Where a means reference, b means boolean, c means char, d means double, f means float, i means integer, l means long and s means short; this encoding scheme shows up repeatedly in the opcode names.]
- Xstore, where X is one of a,d,f,l or i, pops the top value off the stack and puts it into a "local" slot.
- Xconst_Y are a series of optimizations in the opcode set, designed to push a constant value Y of type X onto the stack; for example iconst_0 pushes the integer constant 0 onto the stack, effectively a hard-coded variation of bipush (value).

Branching and control flow. No opcode set would be useful without some kind of constructs to allow the code to execute in some kind of nonlinear form. Accordingly, the JVM opcode set has all the basics you would expect:

- nop, which (not surprisingly) does nothing.
- if(condition), which tests a condition and, if true, jumps to the indicated opcode offset, where (condition) is one of "null", "notnull", "eq", "ne", "gt", "lt", "_icmpeq", "_icmpne", and so on.
- goto. Yes, even if Java doesn't support goto, the JVM does.
- return and Xreturn, where X is one of a, d, f, l, or i, returns from the current to the caller, returning the top of the stack as a value of type X.
- lookupswitch provides an implementation for switch/case tables.

Arithmetic instructions. Again, the JVM opcode set is similar to almost all other CPU instruction sets in that it provides basic mathematic operations, such as add, substract, multiply and divide, as well as basic conversion operators to do widening and narrowing conversions:

- The data conversion opcodes take the form XtoY, where X and Y are one of b, c, d, f, l, s or i; take the top of the stack, which must be of the form delineated by X, and push it back onto the stack in the form of Y.
- The mathematics opcodes take the form XOP, where X is one of d, f, i and l, and where OP is one of add, sub, mul, div, and rem (remainder).
- The bitwise opcodes take the form iOP, where OP is one one of and, or, xor, shl (shift-left), and shr (shift-right).
- The comparison opcodes take the form of XcmpY, where X is d (meaning a double-based comparison), f (meaning a float-based comparison), or l (meaning a long-based comparison) and Y is either g or l, where the difference between the two is simply the treatment of NaN. The first value is compared against the second value, and if the first is greater than the second, 1 is pushed on the stack; if the two values are equal, 0 is pushed, and if the second is greater than the first, -1 is pushed.

Object model instructions. The JVM has a built-in awareness of objects, and as a result, has several opcodes designed specifically to work against objects: creating them, calling methods, accessing fields, and so on:

- new, newarray, anewarray: Create an object (new), an array (newarray), or an array of object references (anewarray). The resulting object or array is pushed onto the top of the stack. Note that in the case of the new opcode, constructors are not called; that is the responsibility of the code to follow.
- getfield, setfield, getstatic, setstatic Get or set the value of the object field referenced by the reference at the top of the stack, be it an instance or static field. When setting the value, the value is at the top of the stack, and the object reference is right beneath it. When working with static fields, obviously no object reference is necessary.
- invokevirtual, invokestatic, invokespecial, invokeinterface Each of these calls a method described by the constant pool entry specified as an operand, using the values pushed on the stack as parameters to the call in left- to-right order (meaning the first parameters to the call are the lowest on the stack, the "this" reference being the first, which is also to say, the lowest reference on the stack). The invokevirtual opcode indicates a call invoked in"normal" method calls on objects, invokeinterface when calling a method through an interface reference, invokestatic when calling static methods, and invokespecial is used to call methods without concern for dynamic binding, in order to invoke the particular class' version of a method regardless of derived type overrides.
- castclass, instanceof As implied, these two deal with casting the object reference at the top of the stack to the type implied in the operand; if it succeeds, either the new reference or a boolean "true" is pushed to the top of the stack, and if it fails, either an exception of type CastClassException is thrown or a boolean "false" is pushed, respsectively.

Block synchronization (that is, synchronized blocks or methods) are handled by two opcodes, monitorenter and monitorexit, each of which takes an object reference on the stack as the object whose monitor the code wishes to attempt to acquire. Note that it is the compiler's job to ensure balanced entry/exit calls, so synchronized methods or blocks generally turn into try/finally blocks to ensure that the monitorexit opcode is always called. (Failure to do so will leave the monitor owned by the thread and eventually deadlock the system.)

Exception handling. Exception handling isn't handled via a particular set of opcodes at all, but by creating a table marking blocks of instructions as guarded and establishing a series of entries indicating what to do, namely, the opcode offset to goto when a particular type of exception is thrown. For example, a fairly simple try/catch/finally block catching several types of exceptions, such as:

```
public class ExFun
{
        public static void main(String[] args)
        {
                try
                {
                        System.out.println("In try block");
                        throw new Exception();
                }
                catch (java.io.IOException ioEx)
                {
                        System.out.println("In catch IOException block");
                        System.out.println(ioEx);
                }
                catch (Exception ex)
                {
                        System.out.println("In catch Exception block");
                        System.out.println(ex);
                }
                finally
                {
                        System.out.println("In finally block");
                }
        }
}
```

... turns into the following method implementation:

```
Compiled from "ExFun.java"
public class ExFun extends java.lang.Object
        SourceFile: "ExFun.java"
        minor version: 0
        major version: 50
        Constant pool:
                (snipped for simplicity)
{
public ExFun();
        (snipped for simplicity)

public static void main(java.lang.String[]);
        Code:
                Stack=2, Locals=3, Args_size=1
                0: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                3: ldc #3; //String In try block
                5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                8: new #5; //class java/lang/Exception
                11: dup
                12: invokespecial #6; //Method java/lang/Exception."<init>":()V
                15: athrow
                16: astore_1
                17: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                20: ldc #8; //String In catch IOException block
                22: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                25: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                28: aload_1
                29: invokevirtual #9; //Method java/io/PrintStream.println:(Ljava/lang/Object;)V
                32: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                35: ldc #10; //String In finally block
                37: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                40: goto 81
                43: astore_1
                44: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                47: ldc #11; //String In catch Exception block
                49: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                52: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                55: aload_1
                56: invokevirtual #9; //Method java/io/PrintStream.println:(Ljava/lang/Object;)V
                59: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                62: ldc #10; //String In finally block
                64: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                67: goto 81
                70: astore_2
                71: getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
                74: ldc #10; //String In finally block
                76: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
                79: aload_2
                80: athrow
                81: return
        Exception table:
                from to target type
                0 16 16 Class java/io/IOException
                0 16 43 Class java/lang/Exception
                0 32 70 any
                43 59 70 any
                70 71 70 any
        LineNumberTable: (snipped)
        StackMapTable: (snipped)
}
```

The contents of the "try" block are the opcode range 0 - 16, so the "catch" blocks from the Java source are encoded in the first two lines of the Exception table (from 0/to 16/target 16/type java.io.IOException and from 0/to 16/target 43/type java.lang.Exception), and the bodies of each begin at the opcode offsets 16 and 43, respectively. Notice, too, that the last three lines of the Exception table indicate that any exception thrown anywhere else in the method–even within the exception blocks–must jump to opcode 70 for processing the body of the finally block. Notice, however, that this is not the only place the finally block appears in the bytecode–it's repeated twice more, once in each catch block. This is just one example of how the javac compiler produces bytecode output that may be surprising and counterintuitive to the Java developer. Also, it's interesting to note that goto opcodes are used to help control flow, so as to create a single point of return when exiting the method from one of the two catch blocks.

The JVM bytecode set has 212 opcodes, with 46 more reserved for future use/expansion. The full set, as mentioned previously, are fully documented in the JVM Specification, and should be kept handy when doing disassembly of Java bytecode (at least until most of the opcodes are recognizable on sight). More importantly, beginners to Java bytecode should recognize that certain Java constructs turn into particular bytecode patterns, and learn to recognize those patterns so as to recognize what the bytecode is doing, without having to work out the stack manipulations on an opcode-by-opcode basis.

Java Bytecode In Practice

Armed with a decent awareness of the JVM bytecode set, we can now examine some common and familiar Java language constructs and see how they map to bytecode, and gain some interesting insight into Java's implementation details while we're at it.

Java 5: Autoboxing

One of the new features of the Java 5 language is that of "autoboxing", where a primitive type instance can be transformed into an object instance because object semantics are required, such as when trying to pass a primitive into a Collection class, as in the following:

```
public class Autoboxing
{
        public static void main(String[] args)
        {
                int x = 5;
                java.util.ArrayList al = new java.util.ArrayList();
                al.add(x);
        }
}
```

Under pre-Java5 rules, the add call would fail, since x is not of a type that extends Object. Under Java5, the code compiles into:

```
0: iconst_5
1: istore_1
2: new #2; //class java/util/ArrayList
5: dup
6: invokespecial #3; //Method java/util/ArrayList."<init>":()V
9: astore_2
10: aload_2
11: iload_1
12: invokestatic #4; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
15: invokevirtual #5; //Method java/util/ArrayList.add:(Ljava/lang/Object;)Z
18: pop
19: return
```

Line 0 pushes the integer constant 5 onto the stack; line 1 stores the top of the stack (5) into the first locals slot (this is our "x" local). Next, we see four opcode instructions, new/dup/invokespecial/astore, which are the common pattern when an object is new'ed and stored into a local variable. (As an exercise, prove to yourself why the dup opcode is
necessary.) Next, on line 10, we push the ArrayList reference onto the stack, then push the value of the "x" local onto the stack after it. Line 12 we call the static Integer.valueOf method, which requires a single stack slot, and consumes the integer value "5", and pushes in its place the newly-constructed Integer object containing the value 5. This object then, in turn, becomes the argument to the add() call, which consumes both the Integer reference and the ArrayList reference off the stack, and pushes the return value from add() back onto the stack after the call.

In other words, bytecode has revealed to us that autoboxing does pretty much what we would have been required to do in pre-Java5 environments: create a wrapper instance around the value (Integer for int, Long for long, and so on), and pass that resulting object in to the ArrayList instead.

Inner Classes

In the JDK 1.1 release, Sun introduced inner classes, a new feature of the Java language that permitted Java developers to create nested classes that had a special, private-visible relationship to the enclosing class. When introduced, however, no changes to the Java Virtual Machine granting some kind of C++-like "friend" capability were added, leading several Java developers at the time to question how this was possible: how could Java grant private access to classes, when the JVM itself enforced private accessibility, and treated these inner classes no differently than any other kind of class?

In the following example, class Inner obviously has access to the private member named "data" in class Outer:

```
class Outer
{
        private int data = 12;
        public Inner getInner()
        {
                return new Inner();
        }
        public class Inner
        {
                public int getData()
                {
                        return data;
                }
        }
}

public class NestedFun
```

```
{
        public static void main(String[] args)
        {
                Outer o = new Outer();
                Outer.Inner i = o.getInner();
                System.out.println(i.getData());
                // prints 12; how?
        }
}
```

So how, exactly, does the compiler make this little bit of syntactic magic work? Discovering this takes a bit of digging, but all is revealed through bytecode if you follow the call chain. To begin, we start with Nested-Fun.main():

```
public static void main(java.lang.String[]);
        Code:
                Stack=2, Locals=3, Args_size=1
                0: new #2; //class Outer
                3: dup
                4: invokespecial #3; //Method Outer."<init>":()V
                7: astore_1
                8: aload_1
                9: invokevirtual #4; //Method Outer.getInner:()LOuter$Inner;
                12: astore_2
                13: getstatic #5; //Field java/lang/System.out:Ljava/io/PrintStream;
                16: aload_2
                17: invokevirtual #6; //Method Outer$Inner.getData:()I
                20: invokevirtual #7; //Method java/io/PrintStream.println:(I)V
                23: return
```

Not surprisingly, this code is fairly straightforward: we see the usual new/dup/invokespecial/astore set to create the instance of Outer, the call to Outer.getInner(), the call to getData(), whose return value is immediately passed to the println() method. (Notice how the compiler chooses to fetch "System.out" prior to the call to getData(), so that the object reference is in the right place on the execution stack for the call to println().) Nothing too exciting here, so let's examine the Outer.Inner.getData() method call:

```
public class Outer$Inner extends java.lang.Object
        SourceFile: "NestedFun.java"
        InnerClass:
                public #21= #4 of #18; //Inner=class Outer$Inner of class Outer
        minor version: 0
        major version: 50
        Constant pool: (snipped)

{
final Outer this$0;

public Outer$Inner(Outer);
        Code:
                Stack=2, Locals=2, Args_size=2
                0: aload_0
                1: aload_1
                2: putfield #1; //Field this$0:LOuter;
                5: aload_0
                6: invokespecial #2; //Method java/lang/Object."<init>":()V
                9: return

public int getData();
        Code:
                Stack=1, Locals=1, Args_size=1
                0: aload_0
                1: getfield #1; //Field this$0:LOuter;
                4: invokestatic #3; //Method Outer.access$000:(LOuter;)I
                7: ireturn
}
```

(Note that I've removed parts of the dump to keep things easier to read; any future full-length listings will be similarly edited.) Several interesting things about the generated bytecode appear in this listing. First, we can see that the "outer this" reference mentioned in the Java Language Specification is explicitly added to the Inner class as a field named "this$0" and marked it final, since an inner class can't change its outerclass reference. Second, we can see that the compiler also generated a constructor for Inner that takes an Outer as a reference to fill in as the "outer this", and we can assume that the call to "new Inner()" inside the Outer.getInner() call is going to make use of this constructor. (Verifying this is left as an exercise to the reader.) Last, but ceratinly not least, we can see that something very interesting is going on inside of the Inner.getData() method, in that it's accessing a static method called "access$000" on the Outer class to get the data returned by the get- Data() call. Sure enough, if we look on Outer...

```
class Outer extends java.lang.Object{
private int data;

Outer();
        Code:
                0: aload_0
                1: invokespecial #2; //Method java/lang/Object."<init>":()V
                4: aload_0
                5: bipush 12
                7: putfield #1; //Field data:I
                10: return

public Outer$Inner getInner();
        Code:
                0: new #3; //class Outer$Inner
                3: dup
                4: aload_0
                5: invokespecial #4; //Method Outer$Inner."<init>":(LOuter;)V
                8: areturn

static int access$000(Outer);
        Code:
                0: aload_0
                1: getfield #1; //Field data:I
                4: ireturn
}
```

... we can see that the compiler generated a static method that specifically opens a hole in the encapsulation wall to grant access to "data" to callers. Unfortunately, because Java has no explicit sense of granting access to specific classes, the "access$000" method has to be marked as package-private, meaning any class in the same package has access to this method. Whether this particular implementation is a hack or an elegant way to add functionality to the language after its initial definition is left up to the reader to decide. What is true, however, is that any language can make use of this hole once they know about it. So a Groovy client could, conceivably, call this method to bypass the "private" declaration on data on any Outer instance, which clearly wasn't a concern back in 1997 when this feature was introduced, but could be of concern now.

Readers are invited to explore other Java language constructs and see how they map into Java bytecode, such as the enhanced for loop, variable argument lists, enumerations, and so on. For a real challenge, take one of the new dynamic languages for the Java platform, such as Rhino (JavaScript) or JRuby, and examine the bytecode generated for particular language constructs there, such as Groovy's closures or JRuby's mixins.

Using Java Bytecode

A variety of tools are available that work on Java bytecode, and while it's well beyond the scope of this article to describe them all, certainly
it does no harm to list them and offer a one-sentence description and potential purpose of each:

- • Javassist is a bytecode-manipulation tool that allows for either static- or dynamic-time manipulation of class bytecode; when combined with a structured syntax to describe what code to manipulate and where, it can form the basis of an Aspect-Oriented Programming platform or language, and in fact has been put to such purpose

in JBoss.
- BCEL and ASM are each lower-level bytecode-manipulation libraries (the first from Apache, the second from ObjectWeb) that can either tweak existing class bytecode or even create new classes on the fly; the Sun JDK includes a version of BCEL in a sun.* package specifically for this purpose, in fact, in order to support the creation of dynamic proxies.
- Jasmin is an open-source project that provides a "Java assembler", allowing developers to explore bytecode constructs directly, thus helping to see the line between the Java language and the Java Virtual Machine itself. As an interesting exercise, the reader is encouraged to determine if it's possible for one class to access the private fields of another through bytecode. (In other words, is it the Java compiler that enforces private accessibility, or the JVM itself?) Jasmin was originally a project created by the authors of the book "Java Virtual Machine" by Jon Meyer and Troy Downing, and although the book is out-of-print, the assembler lives on.
- Bill Venners' Inside the Java Virtual Machine is probably one of the better Java Virtual Machine books written, and contains an excellent overview of Java bytecode to go alongside the Java Virtual Machine Specification, along with good descriptions of other facets of JVM internals. A bit dated at this point, but given that the JVM itself hasn't changed significantly (except in the details of the garbage collector and JIT compiler), it's still highly relevant.

Probably the most important tool to puzzling through bytecode, however, remains the javap disassembler, and readers are encouraged to play with it, to get a better feel for what language constructs turn into what bytecode patterns. Then it becomes a relatively trivial exercise to pull it out in production, examine the raw code inside of a jar file, determine where the problem is, and amaze your friends and co-workers with your incredibly deep knowledge of the Java platform. Fame and fortune are sure to follow... and if not, at least the well-deserved satisfaction that you have a much deeper idea of what's going on under the hood.

Biography

Ted Neward is an independent consultant specializing in high-scale enterprise systems, working with clients ranging in size from Fortune 500 corporations to small 20-person shops. He speaks on the conference circuit, including the No Fluff Just Stuff Symposium tour, discussing Java, .NET and XML service technologies, focusing on Java-.NET interoperability. He has written several widely-recognized books in both the Java and .NET space, including the recently-released "Effective Enterprise Java". He lives in the Pacific Northwest with his wife, two sons, two cats, and eight PCs.

PRINTER FRIENDLY VERSION

TheServerSide.COM
Your Enterprise Java Community