Java bytecode:

Understanding bytecode makes you a better programmer

Peter Haggar
July 01, 2001

This article gives you an understanding of Java bytecode that will enable you to be a better programmer. Like a C or C++ compiler translates source code into assembler code, Java compilers translate Java source code into bytecode. Java programmers should take the time to understand what the bytecode is, how it works, and most importantly, what bytecode is being generated by the Java compiler. In some cases, the bytecode generated is not what you expect.

The information about bytecode, as well as the bytecode presented here, is based on the Java 2 SDK Standard Edition v1.2.1 javac compiler. The bytecode generated by other compilers may vary slightly.

Why understand bytecode?

Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C or C++ programs. The most knowlegable C and C++ programmers know the assembler instruction set of the processor for which they are compiling. This knowledge is crucial when debugging and doing performance and memory usage tuning. Knowing the assembler instructions that are generated by the compiler for the source code you write, helps you know how you might code differently to achieve memory or performance goals. In addition, when tracking down a problem, it is often useful to use a debugger to disassemble the source code and step through the assembler code that is executing.

An often overlooked aspect of Java is the bytecode that is generated by the javac compiler. Understanding bytecode and what bytecode is likely to be generated by a Java compiler helps the Java programmer in the same way that knowledge of assembler helps the C or C++ programmer.

The bytecode is your program. Regardless of a JIT or Hotspot runtime, the bytecode is an important part of the size and execution speed of your code. Consider that the more bytecode you have, the bigger the .class file is and the more code that has to be compiled by a JIT or Hotspot runtime. The remainder of this article gives you an in depth look at Java bytecode.
Generating bytecode

javac Employee.java
javap -c Employee > Employee.bc

Compiled from Employee.java
class Employee extends java.lang.Object {
    public Employee(java.lang.String,int);
    public java.lang.String employeeName();
    public int employeeNumber();
}

Method Employee(java.lang.String,int)
0 aload_0
1 invokespecial #3 <Method java.lang.Object()>
4 aload_0
5 aload_1
6 putfield #5 <Field java.lang.String name>
9 aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void storeData(java.lang.String, int)>
20 return

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn

Method int employeeNumber()
0 aload_0
1 getfield #4 <Field int idNumber>
4 ireturn

Method void storeData(java.lang.String, int)
0 return

This class is very simple. It contains two instance variables, a constructor and three methods. The first five lines of the bytecode file list the file name that is used to generate this code, the class definition, its inheritance (by default, all classes inherit from java.lang.Object), and its constructors and methods. Next, the bytecode for each of the constructors is listed. Then, each method is listed in alphabetical order with its associated bytecode.

You might notice on closer inspection of the bytecode that certain opcodes are prefixed with an `a' or an `i'. For example, in the Employee class constructor you see aload_0 and iload_2. The prefix is representative of the type that the opcode is working with. The prefix `a' means that the opcode is manipulating an object reference. The prefix `i' means the opcode is manipulating an integer. Other opcodes use `b' for byte, `c' for char, `d' for double, etc. This prefix gives you immediate knowledge about what type of data is being manipulated.

Note: Individual codes are generally referred to as opcode. Multiple opcode instructions are generally referred to as bytecode.
The details

To understand the details of the bytecode, we need to discuss how a Java Virtual Machine (JVM) works regarding the execution of the bytecode. A JVM is a stack-based machine. Each thread has a JVM stack which stores frames. A frame is created each time a method is invoked, and consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method. Conceptually, it might look like this:

Figure 1. A frame

![Figure 1. A frame](image)

The array of local variables, also called the local variable table, contains the parameters of the method and is also used to hold the values of the local variables. The parameters are stored first, beginning at index 0. If the frame is for a constructor or an instance method, the reference is stored at location 0. Then location 1 contains the first formal parameter, location 2 the second, and so on. For a static method, the first formal method parameter is stored in location 0, the second in location 1, and so on.

The size of the array of local variables is determined at compile time and is dependent on the number and size of local variables and formal method parameters. The operand stack is a LIFO stack used to push and pop values. Its size is also determined at compile time. Certain opcode instructions push values onto the operand stack; others take operands from the stack, manipulate them, and push the result. The operand stack is also used to receive return values from methods.

```java
public String employeeName()
{
    return name;
}
```

Method java.lang.String employeeName()
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn

The bytecode for this method consists of three opcode instructions. The first opcode, aload_0, pushes the value from index 0 of the local variable table onto the operand stack. Earlier, it was mentioned that the local variable table is used to pass parameters to methods. The this reference
is always stored at location 0 of the local variable table for constructors and instance methods. The this reference must be pushed because the method is accessing the instance data, name, of the class.

The next opcode instruction, getfield, is used to fetch a field from an object. When this opcode is executed, the top value from the stack, this, is popped. Then the #5 is used to build an index into the runtime constant pool of the class where the reference to name is stored. When this reference is fetched, it is pushed onto the operand stack.

The last instruction, areturn, returns a reference from a method. More specifically, the execution of areturn causes the top value on the operand stack, the reference to name, to be popped and pushed onto the operand stack of the calling method.

The employeeName method is fairly simple. Before looking at a more complex example, we need to examine the values to the left of each opcode. In the employeeName method’s bytecode, these values are 0, 1, and 4. Each method has a corresponding bytecode array. These values correspond to the index into the array where each opcode and its arguments are stored. You might wonder why the values are not sequential. Since bytecode got its name because each instruction occupies one byte, why are the indexes not 0, 1, and 2? The reason is some of the opcodes have parameters that take up space in the bytecode array. For example, the aload_0 instruction has no parameters and naturally occupies one byte in the bytecode array. Therefore, the next opcode, getfield, is in location 1. However, areturn is in location 4. This is because the getfield opcode and its parameters occupy location 1, 2, and 3. Location 1 is used for the getfield opcode, location 2 and 3 are used to hold its parameters. These parameters are used to construct an index into the runtime constant pool for the class to where the value is stored. The following diagram shows what the bytecode array looks like for the employeeName method:

**Figure 2. Bytecode array for employeeName method**

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>aload_0</td>
<td>getfield</td>
<td>00</td>
<td>05</td>
<td>areturn</td>
</tr>
</tbody>
</table>
```

Actually, the bytecode array contains bytes that represent the instructions. Looking at a .class file with a hex editor, you would see the following values in the bytecode array:

**Figure 3. Values in the bytecode array**

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2A</td>
<td>B4</td>
<td>00</td>
<td>05</td>
<td>B0</td>
</tr>
</tbody>
</table>
```

2A , B4 , and B0 correspond to aload_0, getfield, and areturn, respectively.

```java
public Employee(String strName, int num) {
    name = strName;
    idNumber = num;
    storeData(strName, num);
}
```
Method Employee(java.lang.String,int)
0 aload_0
1 invokespecial #3 <Method java.lang.Object>()
4 aload_0
5 aload_1
6 putfield #5 <Field java.lang.String name>
9 aload_0
10 iload_2
11 putfield #4 <Field int idNumber>
14 aload_0
15 aload_1
16 iload_2
17 invokespecial #6 <Method void storeData(java.lang.String, int)>
20 return

The first opcode instruction at location 0, aload_0, pushes the this reference onto the operand stack. (Remember, the first entry of the local variable table for instance methods and constructors is the this reference.)

The next opcode instruction at location 1, invokespecial, calls the constructor of this class's superclass. Because all classes that do not explicitly extend any other class implicitly inherit from java.lang.Object, the compiler provides the necessary bytecode to invoke this base class constructor. During this opcode, the top value from the operand stack, this, is popped.

The next two opcodes, at locations 4 and 5 push the first two entries from the local variable table onto the operand stack. The first value to be pushed is the this reference. The second value is the first formal parameter to the constructor, strName. These values are pushed in preparation for the putfield opcode instruction at location 6.

The putfield opcode pops the two top values off the stack and stores a reference to strName into the instance data name of the object referenced by this.

The next three opcode instructions at locations 9, 10, and 11 perform the same operation with the second formal parameter to the constructor, num, and the instance variable, idNumber.

The next three opcode instructions at locations 14, 15, and 16 prepare the stack for the storeData method call. These instructions push the this reference, strName, and num, respectively. The this reference must be pushed because an instance method is being called. If the method was declared static, the this reference would not need to be pushed. The strName and num values are pushed since they are the parameters to the storeData method. When the storeData method executes, the this reference, strName, and num, will occupy indexes 0, 1, and 2, respectively, of the local variable table contained in the frame for that method.

Size and speed issues

Performance is a critical issue for many desktop and server systems that use Java. As Java moves from these systems to smaller embedded devices, size issues become important too. Knowing what bytecode is generated for a sequence of Java instructions can help you write smaller and more efficient code. For example, consider synchronization in Java. The following two methods return the top element from a stack of integers that is implemented as an array. Both methods use synchronization and are functionally equivalent:
public synchronized int top1()
{
    return intArr[0];
}

public int top2()
{
    synchronized (this) {
        return intArr[0];
    }
}

These methods, although using synchronization differently, are functionally identical. What is not obvious, however, is that they have different performance and size characteristics. In this case, top1 is approximately 13 percent faster than top2 as well as much smaller. Examine the generated bytecode to see how these methods differ. The comments are added to the bytecode to assist in understanding what each opcode does.

Method int top1()
0 aload_0      //Push the object reference(this) at index
               //0 of the local variable table.
1 getfield #6 <Field int intArr[]>        
   //Pop the object reference(this) and push
   //the object reference for intArr accessed
   //from the constant pool.
4 iconst_0      //Push 0.
5 iaload        //Pop the top two values and push the
               //value at index 0 of intArr.
6 ireturn       //Pop top value and push it on the operand
               //stack of the invoking method. Exit.

Method int top2()
0 aload_0      //Push the object reference(this) at index
               //0 of the local variable table.
1 astore_2     //Pop the object reference(this) and store
               //at index 2 of the local variable table.
2 aload_2      //Push the object reference(this).
3 monitorenter  //Pop the object reference(this) and
               //acquire the object's monitor.
4 aload_0      //Beginning of the synchronized block.
               //Push the object reference(this) at index
               //0 of the local variable table.
5 getfield #6 <Field int intArr[]>        
   //Pop the object reference(this) and push
   //the object reference for intArr accessed
   //from the constant pool.
8 iconst_0      //Push 0.
9 iaload        //Pop the top two values and push the
               //value at index 0 of intArr.
10 istore_1     //Pop the value and store it at index 1 of
               //the local variable table.
11 jsr 19       //Push the address of the next opcode(14)
               //and jump to location 19.
14 iload_1      //Push the value at index 1 of the local
               //variable table.
15 ireturn      //Pop top value and push it on the operand
               //stack of the invoking method. Exit.
16 aload_2     //End of the synchronized block. Push the
               //object reference(this) at index 2 of the
               //local variable table.
17 monitorexit  //Pop the object reference(this) and exit
               //the monitor.
18 athrow       //Pop the object reference(this) and throw
               //an exception.
19 astore_3     //Pop the return address(14) and store it
top2 is larger and slower than top1 because of how the synchronization and exception processing is done. Notice that top1 uses the synchronized method modifier, which does not generate extra code. By contrast, top2 uses a synchronized statement in the body of the method.

Using synchronized in the body of the method generates the bytecode for the monitorenter and monitorexit opcodes, as well as additional code to handle exceptions. If an exception is generated while executing inside of a synchronized block (a monitor), the lock is guaranteed to be released prior to exiting the synchronized block. The implementation of top1 is slightly more efficient than that of top2; this results in a very small performance gain.

When the synchronized method modifier is present, as in top1, the acquisition and subsequent release of the lock is not done with the monitorenter and monitorexit opcodes. Instead, when the JVM invokes a method, it checks for the ACC_SYNCHRONIZED property flag. If this flag is present, the thread that is executing acquires a lock, calls the method, and then releases the lock when the method returns. If an exception is thrown from a synchronized method, the lock is automatically released before the exception leaves the method.

**Note:** The ACC_SYNCHRONIZED property flag is included in a method's method_info structure if the synchronized method modifier is present.

Whether you use synchronized as a method modifier or with the synchronized block, there are size implications. Use synchronized methods only when your code requires synchronization and you understand the cost imposed by their usage. If an entire method needs to be synchronized, I prefer the method modifier over the synchronized block in order to produce smaller and slightly faster code.

This is just one example of using knowledge of bytecode to make your code smaller and faster; more information is found in my book, *Practical Java*.

**Compiler options**

The javac compiler provides a few options that you need to know. The first is the -O option. The JDK documentation claims that -O will optimize your code for execution speed. Using -O with the javac compiler with the Sun Java 2 SDK has no effect on the generated code. Previous versions of the Sun javac compiler performed some rudimentary bytecode optimizations, but these have since been removed. The SDK documentation, however, has not been updated. The only reason -O remains as an option is for compatibility with older make files. Therefore, there is currently no reason to use it.
This also means that the bytecode that is generated by the javac compiler is not any better than the code that you write. For example, if you write a loop that contains an invariant, the invariant will not be removed from the loop by the javac compiler. Programmers are used to compilers from other languages that clean up badly written code. Unfortunately, javac does not do this. More importantly, the javac compiler does not perform simple optimizations like loop unrolling, algebraic simplification, strength reduction, and others. To get these benefits and other simple optimizations, the programmer must perform them on the Java source code and not rely on the javac compiler to perform them. There are many techniques you can use to make the Java compiler generate faster and smaller bytecode. Unfortunately, until Java compilers perform them, you must implement them yourself to achieve their benefits.

The javac compiler also supplies the -g and -g:none options. The -g option tells the compiler to generate all the debugging information. The -g:none option tells the compiler to generate no debugging information. Compiling with -g:none generates the smallest possible .class file. Therefore, this option should be used when trying to generate the smallest possible .class files prior to deployment.

**Java debuggers**

One very useful feature that I have yet to see in a Java debugger is a disassembly view similar to that of a C or C++ debugger. Disassembling Java code would reveal the bytecode, much like disassembling C or C++ code reveals assembler code. In addition to this feature, another useful feature could be the ability to single step through the bytecode, executing one opcode at a time.

This level of functionality would allow the programmer to see, first hand, the bytecode generated by the Java compiler as well as step through it during debugging. The more information a programmer has about the code generated and executed, the better chance there is of avoiding problems. This type of debugger feature would also encourage programmers to look at, and understand, the bytecode that is executing for their source code.

**Summary**

This article gives you an overview and general understanding of Java bytecode. The best programmers of any language understand the intermediate form that the high-level language is translated to before execution. With Java, this intermediate representation is bytecode. Understanding it, knowing how it works, and more importantly, knowing what bytecode is generated by the Java compiler for particular source code, is critical to writing the fastest and smallest code possible.
Related topics


© Copyright IBM Corporation 2001
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)