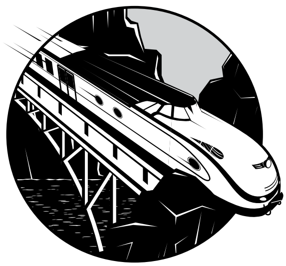
D

JAVA BYTECODE ASSEMBLY FOR THE HLL PROGRAMMER



Compilers for the Java language differ from many contemporary compilers in that they typically produce a special Java bytecode (JBC) rather than true machine language. An interpreter processes (interprets) this bytecode at runtime. In many systems, a just-in-time (JIT) compiler (operating during interpretation) translates the JBC into a sequence of native machine instructions that do the same work as the interpreter, but the JIT translation boosts the JBC's performance by eliminating the need to fetch and interpret it at runtime.

To better understand how the Java compiler operates, you need a basic familiarity with the Java bytecode instruction set. To that end, this appendix:

- Describes the basic JBC machine architecture
- Provides an overview of JBC assembly language so that you'll be able to read the bytecode output produced by the Java compiler

D.1 Assembly Syntax

Although some people have written Java bytecode assemblers, you don't really write JBC assembly language. The Java compiler does not produce JBC assembly language output (that a bytecode assembler would convert to object code). JBC is so simple, by design, that the Java compiler emits it directly without using an assembler. Some people have written JBC assembly by hand, but there's really no reason for doing so other than for the experience. Because of runtime interpretation, there are no runtime benefits to writing JBC assembly. Furthermore, the JBC architecture doesn't really provide any instructions that let you accomplish tasks in JBC assembly that couldn't be done directly in Java.

This book uses the javap Java class file disassembler to disassemble compiler output files. For the most part, JBC disassemblies consist of a list of instructions that each have zero or one operands.

D.2 Basic Java Machine Architecture

The Java virtual machine (JVM) uses a typical *p-machine* (pseudo-machine) stack architecture, similar to the UCSD Pascal p-machine from the 1970s and 1980s. The complete and current documentation for the JVM is available at the Oracle website (*http://docs.oracle.com*). Search online for "Java Virtual Machine Specification," and you'll find the latest version of this document. This appendix discusses the Java SE 10 Edition, most of which should still apply as newer versions are released.

D.2.1 Java VM Registers

The Java VM reserves a set of registers for each executing thread. Specifically, there is a Java PC (program counter) register that holds the address of the currently executing bytecode instruction,[1] and a stack pointer register that points at the current top of stack (TOS).

D.2.2 Java VM Primitive Types and Values

The Java VM supports two types of data: reference types and primitive types. *Reference types,* as their name suggests, are pointers to various objects (including array, interface, and class objects). *Primitive types* are the basic numeric types, the `boolean` type, and the `returnAddress` type, as described here:

`byte` 8-bit signed two's-complement integers in the range –128 through +127.

`short` 16-bit signed two's-complement integers in the range –32,768 through +32,767.

1. If the system is executing native code, such as that produced by the Java JIT compiler, then the value held in the Java PC register is meaningless.

`int` 32-bit signed two's-complement integers in the range –2,147,483,648 through +2,147,483,647.

`long` 64-bit signed two's-complement integers in the range –9,223,372,036,854,775,808 through +9,223,372,036,854,775,807.

`char` 16-bit unsigned integers representing Unicode code points in the Basic Multilingual Plane (BMP).

`float` 32-bit single-precision floating-point values.

`double` 64-bit double-precision floating-point values.

`boolean` 8-bit Boolean values encoding `0` as `false` and `1` as `true`.

`returnAddress` Typically a pointer holding the address of a Java VM instruction (bytecode).

D.2.3 Java VM Reference Types

A Java VM reference type is a pointer to a class object, an interface object, or an array object. Reference objects can also have the special value `null`, meaning they're not pointing at a specific value.

D.2.4 Java Memory Areas

The Java VM defines several special areas of memory that contain specific data for the threads running under the VM:

Stack Each Java VM thread has its own stack area.

Heap The heap is a large (possibly expandable) block of memory. All threads share the same heap. Dynamic allocation of objects and arrays occurs on the heap.

Method area The method area holds the executable bytecodes for all methods in the Java VM. All threads share the same method area.

Runtime constant pool The runtime constant pool is a per-class or per-interface runtime representation of the constant pool in a class. It holds numeric literal (constant/immediate) values, string constants, and other entities.

Native method stacks Native method stacks are blocks of memory associated with native code running on the underlying CPU (for example, when executing pure machine code produced by a JIT translation of the bytecode, or when linking in code written in a different language with a Java program).

Frames A frame is a block of memory (allocated on a Java stack) to pass parameters, return results, and hold local variables for a method invocation.

D.3 Java VM Addressing Modes

Although the JVM is a zero-address stack machine, several instructions do include operands as part of the opcode in addition to operands appearing on the stack. This section discusses the various ways JVM instructions access memory and constant values.

D.3.1 Immediate and Constant Access

The JVM provides several instructions that push constant values onto the stack. Table D-1 lists the immediate versions of these instructions.

Table D-1: Array Load and Store Instructions

Instruction	Operand	Description
`bipush`	*`byte constant`*	Byte immediate push. Sign-extends a byte constant (*`-128..+127`*) and pushes that value onto the stack.
`sipush`	*`short constant`*	Short immediate push. Sign-extends a short constant (*`-32768..+32767`*) and pushes that value onto the stack.
`iconst_m1` `iconst_0` `iconst_1` `iconst_2` `iconst_3` `iconst_4` `iconst_5`	(none)	Pushes the specified immediate constant (encoded in the opcode) onto the stack as a 32-bit integer.
`lconst_0` `lconst_1`	(none)	Pushes the specified immediate constant (encoded in the opcode) onto the stack as a 64-bit integer.
`fconst_0` `fconst_1` `fconst_2`	(none)	Pushes the specified immediate constant (encoded in the opcode) onto the stack as a 32-bit floating-point value.
`dconst_0` `dconst_1`	(none)	Pushes the specified immediate constant (encoded in the opcode) onto the stack as a 64-bit floating-point value.
`aconst_NULL`	(none)	Pushes a `null` pointer onto the stack.
`ldc`	*`index`*	Loads a constant (numeric, string, or reference) onto the stack. The *`index`* (`0..255`) is an index into a runtime constant table.
`ldc_w`	*`index`*	Loads a 32-bit constant (numeric, string, or reference) onto the stack. The *`index`* (`0..65535`) is an index into a runtime constant table.
`ldc2_w`	*`index`*	Loads a 64-bit constant (numeric, string, or reference) onto the stack. The *`index`* (`0..65535`) is an index into a runtime constant table.

For constants outside the ranges possible with the immediate push instructions, the JVM provides the `ldc`, `ldc_w`, and `ldc2_w` instructions. These instructions copy a constant (numeric, string, or reference) from the Java VM constant pool (in memory) onto the stack.

The `ldc` instruction is a 2-byte instruction consisting of an opcode and a 1-byte unsigned index value. The *`index`* byte is an index into the runtime constant pool for the current class. This means that the `ldc` instruction provides access to, at most, 256 different constants in the constant pool for the current class. The constant value can be any 32-bit value: an integer, a float, or a reference to a string, array, or some other object.

Though it is relatively rare, a given class could have more than 256 (literal) constants. In this situation, the Java compiler will use the 3-byte `ldc_w` (wide) instruction. The `ldc_w` instruction consists of a 1-byte opcode followed by a 2-byte unsigned index. That index provides access to up to 65,536 different constants within the *current class*. Each class has its own constant pool, so if a program needs more than 65,536 different constants, you simply spread them out across multiple classes. However, keep in mind that the JVM architecture limits you to 65,536 constants within a single class. Here's a short Java program that writes a synthetic Java program (to the standard output, which you can capture using I/O redirection):

```
public class GenJava {
    /** Main method */
    public static void main(String[] args) {

        System.out.println( "public class BigVars { " );
        int i;
        for( i=65538; i < 140000; ++i )
        {
            System.out.println( "public static int v" + i + " = " + i + ";" );
        }
        System.out.println( "}" );
    }
}
```

Running this program from the command line (`java GenJava.class >BigVars.java`) produces a sample Java source file that defines more than 65,536 different static constants (along with more than 65,536 different variables; however, that's a different issue). Compiling this program with Java produces the following output:

```
[iMac-Pro:Build/Products/Debug] rhyde% javac BigVars.java
BigVars.java:2: code too large
public static int v65538 = 65538;
                  ^
BigVars.java:1: too many constants
public class BigVars {
       ^
2 errors
```

Fortunately, the likelihood that you'll ever see this error outside of a synthetic program such as *BigVars.java* is almost nil.

D.3.2 Java Static Data Access

Class static data fields appear in a special location in memory accessible through symbols in the associated class's constant pool. To access static data members of a class, Java uses the `putstatic` and `getstatic` instructions. These are 3-byte instructions consisting of an opcode followed by a 16-bit index into the constant pool. The constant pool entry contains additional information about the static variable's location and type. In order to access a static object, the JVM first uses the index to locate the symbol table information in the constant pool, and then the symbol table entry to locate the actual data to push on the stack (`getstatic`) or to pop the value on the stack into (`putstatic`). This means accessing class static objects is actually more work than accessing local variables in a method.

Consider the following sample Java program:

```
public class TestStatic
{
    public static byte b = 0;
      public static short s = 1;
      public static int i = 2;
      public static long l = 3;
      public static float f = 4.0f;
      public static double d = 5.0;
      public static String st = "6";
      public static void main( String[] args )
      {
            byte lb = b;
            short ls = s;
            int li = i;
            long ll = l;
            float lf = f;
            double ld = 5.0;
            String lst = st;
      }
}
```

Here's the code the Java compiler emits to initialize the `TestStatic` static objects prior to executing the `main()` method:[2]

```
static {};
  Code:
   0:   iconst_0
   1:   putstatic   #2;  //Field b:B
   4:   iconst_1
   5:   putstatic   #3;  //Field s:S
   8:   iconst_2
   9:   putstatic   #4;  //Field i:I
   12:  ldc2_w      #10; //long 3l
   15:  putstatic   #5;  //Field l:J
```

2. The `javap` tool produces this output from a compiled Java class file, as discussed in the section "Using the Java Bytecode Disassembler to Analyze Java Output" on page 130.

```
   18:  ldc         #12; //float 4.0f
   20:  putstatic   #6;  //Field f:F
   23:  ldc2_w      #7;  //double 5.0d
   26:  putstatic   #13; //Field d:D
   29:  ldc         #14; //String 6
   31:  putstatic   #9;  //Field st:Ljava/lang/String;
   34:  return
}
```

The instructions at offsets 0 and 1 push `0` onto the stack and then store that `0` into static variable `b`. The instructions at offsets 4 and 5 push `1` onto the stack and store that value into static variable `s`. And so on. This corresponds to the initializers attached to each of the `public static` variables in the `TestStatic` class.

The Java compiler emits the following code for the `main()` function:

```
public static void main(java.lang.String[]);
Code:
   0:   getstatic   #2; //Field b:B
   3:   istore_1
   4:   getstatic   #3; //Field s:S
   7:   istore_2
   8:   getstatic   #4; //Field i:I
   11:  istore_3
   12:  getstatic   #5; //Field l:J
   15:  lstore  4
   17:  getstatic   #6; //Field f:F
   20:  fstore  6
   22:  getstatic   #7; //Field d:D
   25:  dstore  7
   27:  getstatic   #8; //Field st:Ljava/lang/String;
   30:  astore  9
32:  return
```

The `getstatic` instructions in this code sequence fetch the data for each of the static variables and push that data onto the stack. The `istore`, `lstore`, `fstore`, `dstore`, and `astore` instructions store the value on the TOS into local variables in the `main()` method's stack frame (into variables corresponding to `lb`, `ls`, `li`, `ll`, `lf`, `ld`, and `lst`, respectively).

As noted, accessing static objects is extra work for the JVM interpreter. If your Java code executes in interpretive mode most of the time (that is, you're not using the JIT or some other Java compiler that translates Java into native machine code), avoid using static objects unless they're absolutely necessary, as they're likely to hamper performance.

D.3.3 Java Class Field Data Access

The JVM provides two instructions to access data members of a class: `putfield` (which pops the stack and stores that data into an object's data member) and `getfield` (which pushes data from an object's data member). Both instructions are 3 bytes long, consisting of a 1-byte opcode and a 2-byte index. The 2-byte index is a 16-bit unsigned integer that provides an index into the class's constant pool. That entry in the constant pool is a symbol table entry (much like the index following the `getstatic` and `putstatic` opcodes). This provides type and other descriptor information, including a value that specifies the offset of the data field within the actual object in memory. However, the descriptor doesn't know where the object actually sits in memory (because there could be many different instances of it).

To resolve this issue, the `getfield` and `putfield` instructions (unlike the `getstatic` and `putstatic` instructions) require a reference to the object on the stack. The `putfield` instruction also requires the data to store into the data field on the stack (that is, push the object reference first, then push the data to store, and finally execute the `putfield` instruction).

Consider the following example of a Java class:

```
class Example
{
    public byte b;
    public short s;
    public int i;
    public long l;
    public float f;
    public double d;
    public string st;

    Example()
    {
        b = 0;
        s = 1;
        i = 2;
        l = 3;
        f = 4.0f;
        d = 5.0;
        st = "6";
    }
}
```

Here's the JBC that the compiler emits for the `Example` constructor:

```
Example();
  Code:
   0:   aload_0
   1:   invokespecial   #1;  //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   iconst_0
   6:   putfield        #2;  //Field b:B
   9:   aload_0
   10:  iconst_1
   11:  putfield        #3;  //Field s:S
   14:  aload_0
   15:  iconst_2
   16:  putfield        #4;  //Field i:I
   19:  aload_0
```

```
   20:  ldc2_w         #5;  //long 3l
   23:  putfield       #7;  //Field l:J
   26:  aload_0
   27:  ldc            #8;  //float 4.0f
   29:  putfield       #9;  //Field f:F
   32:  aload_0
   33:  ldc2_w         #10; //double 5.0d
   36:  putfield       #12; //Field d:D
   39:  aload_0
   40:  ldc            #13; //String 6
   42:  putfield       #14; //Field st:Ljava/lang/String;
   45:  return
}
```

The `aload_0` instruction (discussed a little later) at offset 4 loads the `this` pointer onto the stack. The `putfield` instruction uses this pointer to the newly allocated `Example` object (created immediately prior to calling this class constructor) to access the object's data members.

The instructions at offsets 5 and 6 push the value `0` onto the stack (`iconst_0`), and then store the `0` into `this.b` using the `putfield` instruction. The `putfield` operand (`#2` in this case) is the index into the constant pool that holds the symbol table entry for field `b`; the `javap` application is nice enough to print this symbol table information for us during disassembly. The instructions at offset 9/10/11, 14/15/16, 19/20/23, 26/27/29, and 32/33/36 store the appropriate constant values into the `s`, `i`, `l`, `f`, `d`, and `st` data members, respectively.

Here's the JVM bytecode for the main function in this example:

```
public static void main(java.lang.String[]);
  Code:
   0:   new            #2; //class Example
   3:   dup
   4:   invokespecial  #3; //Method Example."<init>":()V
   7:   astore_1
   8:   aload_1
   9:   getfield       #4; //Field Example.b:B
   12:  istore_2
   13:  aload_1
   14:  getfield       #5; //Field Example.s:S
   17:  istore_3
   18:  aload_1
   19:  getfield       #6; //Field Example.i:I
   22:  istore  4
   24:  aload_1
   25:  getfield       #7; //Field Example.l:J
   28:  lstore  5
   30:  aload_1
   31:  getfield       #8; //Field Example.f:F
   34:  fstore  7
   36:  aload_1
   37:  getfield       #9; //Field Example.d:D
   40:  dstore  8
   42:  aload_1
```

```
   43:  getfield       #10; //Field Example.st:Ljava/lang/String;
   46:  astore  10
   48:  return

}
```

The code sequence is similar to the constructor given earlier. The main difference is that this code stores the results into local variables (using `istore_<i>`, `lstore`, `fstore`, `dstore`, and `astore` instructions). For example, the instruction at offset 8 loads a reference to `x` onto the stack (using the `aload_1` instruction), gets the data from `x.b` using the `getfield` instruction at offset 9, and then stores the value (now on the stack) into the local variable `lb` using the `istore_2` instruction. It takes the combination of the reference to `x` (pushed on the stack by the `aload_1` instruction), plus `getfield`'s `#4` operand, to compute the actual destination address where `getfield` retrieves the value of `x.b`.

D.3.4 Accessing Local Values in the Current Stack Frame

The JVM dedicates a fair percentage of the instruction set to accessing local variables in a method's current stack frame.

The *stack frame* is an array of 32-bit (double word) values holding parameter and local variable values during a function invocation. (Double and long values consume two double words in the stack frame.) The JVM references values in the stack frame by their index into this array of 32-bit values. During compilation, the Java compiler associates the first element of the stack frame (index 0) with the first parameter (if it exists), the second element of the stack frame (index 1) with the second parameter, and so on. After consuming all the parameters, the Java compiler associates the next batch of indexes with the local variables (typically as it encounters them during compilation). (Double and long variables and parameters each consume two indexes.)

The JVM provides separate load instructions for integer variables (`iload`), long variables (`lload`), float variables (`fload`), double variables (`dload`), and reference objects (array, objects, and strings—`aload`), along with a complementary form of the store instruction for each: `istore`, `lstore`, `fstore`, `dstore`, and `astore`. The base form of these instructions is 2 bytes long, consisting of an opcode and an unsigned 8-bit index into the current stack frame. As very few methods have more than 256 local variables and parameters, the 2-byte form usually suffices. However, there is a special 4-byte form of these instructions consisting of a special *wide* prefix, the opcode, and a 2-byte index that allows up to 65,536 different slots in the stack frame. If you have more than 65,536 different variables in a single function, perhaps you should rethink how you're coding your application![3]

3. Note that arrays, objects, and strings are reference objects and only consume only one slot in the stack frame; they do not consume one slot for each array element, character in the string, or field in the object. The stack frame slot for these objects contains a reference to the actual object data that is on the heap, not in the stack frame.

The JVM provides special 1-byte forms of these instructions that provide access to the first four slots in the stack frame. These instructions are `xload_0`, `xload_1`, `xload_2`, `xload_3`, `xstore_0`, `xstore_1`, `xstore_2`, and `xstore_3`, where `x` is one of {`i`, `l`, `f`, `d`, `a`}. Because the parameters appear first in the stack frame, this provides short (and quick) access to the first four parameters.[4] If there are fewer than four parameters, these instructions provide short (and quick) access to the first couple of local variables.

The Java compiler always allocates (at least) 32 bits for local variables even if they are `byte`, `char`, `boolean`, `short`, or any other types that require fewer than 32 bits. Java doesn't bother trying to save a few bytes in memory when allocating smaller objects. Even if you allocate hundreds of these small variables, it won't have a meaningful impact on the memory usage in modern machines.

Of course, the situation is different if you have an array of bytes (or other small data type). As you'll soon see, Java provides special instructions to access elements of an array, which avoids wasted memory when you're dealing with potentially large arrays.

D.3.5 Accessing Array Data in Java

The JVM provides eight instructions that load an element of an array onto the stack and a corresponding set of eight instructions that pop a value from the stack and store it to an array element. Table D-2 lists the instructions and their respective data types.

Table D-2: Array Load and Store Instructions

Instruction	Stack operands	Description
`aaload`	*`arrayRef`*, *`index`*	*`arrayRef`* is a pointer to an array of reference objects. *`index`* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the array reference element at *`arrayRef`*`[`*`index`*`]` back onto the stack.
`aastore`	*`arrayRef`*, *`index`*, *`reference`*	*`arrayRef`* is a pointer to an array of reference objects. *`index`* is an unsigned index into that array. *`reference`* is a reference (pointer) value. This instruction pops *`reference`* off the stack and then pops the next two operands and stores the *`reference`* value into the array element at *`arrayRef`*`[`*`index`*`]`.

(continued)

4. Keep in mind that double and long parameters consume two slots; therefore, passing double or long parameters reduces the number of available slots for parameters or local variables.

Table D-2: Array Load and Store Instructions (continued)

Instruction	Stack operands	Description
`baload`	*`arrayRef`*, *`index`*	*`arrayRef`* is a pointer to an array of 8-bit values (such as `byte` or `boolean`). *`index`* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `byte` element at *`arrayRef`*`[`*`index`*`]` back onto the stack. This instruction sign-extends the 8-bit array element to 32 bits prior to pushing the value onto the stack.
`bastore`	*`arrayRef`*, *`index`*, *`byteValue`*	*`arrayRef`* is a pointer to an array of 8-bit objects. *`index`* is an unsigned index into that array. *`byteValue`* is a 32-bit value. This instruction pops *`byteValue`* off the stack, pops the next two operands off the stack, and then stores the LO 8 bits of *`byteValue`* into the array element at *`arrayRef`*`[`*`index`*`]`.
`caload`	*`arrayRef`*, *`index`* (TOS)	*`arrayRef`* is a pointer to an array of (16-bit Unicode) characters. *`index`* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `char` element at *`arrayRef`*`[`*`index`*`]` back onto the stack. This instruction zero-extends the 16-bit `char` value to 32 bits prior to pushing it onto the stack.
`castore`	*`arrayRef`*, *`index`*, *`charValue`* (TOS)	*`arrayRef`* is a pointer to an array of 16-bit character objects. *`index`* is an unsigned index into that array. *`charValue`* is a 32-bit value (presumably containing a 16-bit Unicode character code in its LO 16 bits). This instruction pops the *`charValue`* value off the stack, pops the next two operands off the stack, and then stores the LO 16 bits of *`charValue`* into the array element at *`arrayRef`*`[`*`index`*`]`.
`daload`	*`arrayRef`*, *`index`* (TOS)	*`arrayRef`* is a pointer to an array of double-precision (64-bit) floating-point values. *`index`* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `double` element at *`arrayRef`*`[`*`index`*`]` back onto the stack. This instruction pushes 64 bits onto the stack.
`dastore`	*`arrayRef`*, *`index`*, *`doubleValue`* (64 bits on TOS)	*`arrayRef`* is a pointer to an array of 64-bit `double` objects. *`index`* is an unsigned index into that array. *`doubleValue`* is a 64-bit double-precision floating-point value (occupying two 32-bit entries on the stack). This instruction pops the 64-bit *`doubleValue`* off the stack, pops the next two operands off the stack, and then stores *`doubleValue`* into the array element at *`arrayRef`*`[`*`index`*`]`.

(continued)

Table D-2: Array Load and Store Instructions (continued)

Instruction	Stack operands	Description
`faload`	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to an array of single-precision (32-bit) floating-point values. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `float` element at *arrayRef*[*index*] back onto the stack.
`fastore`	*arrayRef*, *index*, *floatValue* (TOS)	*arrayRef* is a pointer to an array of 32-bit `float` objects. *index* is an unsigned index into that array. *floatValue* is a 32-bit value. This instruction pops the *floatValue* off the stack, pops the next two operands off the stack, and then stores *floatValue* into the array element at *arrayRef*[*index*].
`iaload`	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to an `int` array. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `int` element at *arrayRef*[*index*] back onto the stack.
`iastore`	*arrayRef*, *index*, *intValue* (TOS)	*arrayRef* is a pointer to an array of 32-bit `int` objects. *index* is an unsigned index into that array. *intValue* is a 32-bit value. This instruction pops *intValue* off the stack, pops the next two operands off the stack, and then stores *intValue* into the array element at *arrayRef*[*index*].
`laload`	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to an array of (64-bit) `long int` values. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `long int` element at *arrayRef*[*index*] back onto the stack. This instruction pushes 64 bits onto the stack.
`lastore`	*arrayRef*, *index*, *longIntValue* (64 bits on TOS)	*arrayRef* is a pointer to an array of 64-bit `long int` objects. *index* is an unsigned index into that array. `long`*IntValue* is a 64-bit value (occupying two 32-bit entries on the stack). This instruction pops the 64-bit `long`*IntValue* off the stack, pops the next two operands off the stack, and then stores the `long`*IntValue* value into the array element at *arrayRef*[*index*].
`saload`	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to a (16-bit) `short` array. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the 16-bit `short int` element at *arrayRef*[*index*] back onto the stack. This instruction sign-extends the `short int` to 32 bits while pushing the value onto the stack.

(continued)

Table D-2: Array Load and Store Instructions (continued)

Instruction	Stack operands	Description
`castore`	*arrayRef*, *index*, *shortValue* (TOS)	*arrayRef* is a pointer to an array of 16-bit `short int` objects. *index* is an unsigned index into that array. *shortValue* is a 32-bit value (that holds a short integer in its LO 16 bits). This instruction pops the 32-bit *shortValue* off the stack, pops the next two operands off the stack, and then stores the LO 16 bits of *shortValue* into the array element at *arrayRef*[*index*].

D.4 Java VM Conditional Control Flow

The JVM provides a set of `if` instructions to compare a pair of 32-bit signed values on the stack and transfer control based on the result of the comparison. These instructions are 3 bytes long, consisting of a 1-byte opcode and a 2-byte signed displacement value. The instructions are:

`if_icmpeq` Compares TOS to NOS (next on stack) and transfers control if TOS == NOS.

`if_icmpne` Compares TOS with NOS and transfers control if TOS != NOS.

`if_icmplt` Compares TOS with NOS and transfers control if NOS < TOS.

`if_icmple` Compares TOS with NOS and transfers control if NOS ≤ TOS.

`if_icmpgt` Compares TOS with NOS and transfers control if NOS > TOS.

`if_icmpge` Compares TOS with NOS and transfers control if NOS ≥ TOS.

As the JVM converts `byte`s, `short`s, and `char`s to 32 bits when pushing their values onto the stack, the JVM also uses these `if_icmp`*`xx`* instructions for those data types.

If the result of the comparison is `true`, these instructions will sign-extend the 16-bit displacement immediately following the opcode to 32 bits and add this value to the current JVM program counter value. The JVM will then fetch the next opcode from the new address held in the program counter register. If the result of the comparison is `false`, execution will continue with the opcode immediately following the third byte of these instructions.

To compare references (pointers), the JVM provides two special instructions, `if_acmpeq` and `if_acmpne`. These two instructions are also 3 bytes long (1-byte opcode and 2-byte displacement) and transfer control if the two reference values on TOS and NOS are equal (`if_acmpeq`) or not equal (`if_acmpne`). The concept of less than or greater than doesn't really apply to references, so the JVM does not provide any tests other than for (in)equality.

To compare `long int`, `float`, and `double` types, you first use a comparison instruction to compare the two values on the stack. The comparison instructions pop the two values on the stack and push the value `-1` if NOS < TOS, `0` if

NOS == TOS, or `1` if NOS > TOS. Then you use one of the following (3-byte) `if` instructions to transfer control based on the result on the TOS:

`ifeq` Pops the value on TOS and transfers control if the value is equal to `0`.

`ifne` Pops the value on TOS and transfers control if the value is not equal to `0`.

`iflt` Pops the value on TOS and transfers control if the value is less than `0`.

`ifle` Pops the value on TOS and transfers control if the value is less than or equal to `0`.

`ifgt` Pops the value on TOS and transfers control if the value is greater than `0`.

`ifge` Pops the value on TOS and transfers control if the value is greater than or equal to `0`.

The `lcmp` instruction pops two 64-bit `long` values off the stack and pushes `-1`, `0`, or `+1` based on the comparison of NOS to TOS (greater than, equal to, or less than, respectively).

Because `float` and `double` computations can produce illegal results (`NaN`, or "not a number"), there are actually two separate float and double comparisons: `fcmpl`/`fcmpg` and `dcmpl`/`dcmpg`. These instructions compare the `float` or `double` values on TOS and NOS and leave `-1`, `0`, or `+1` on the stack whenever the floating-point values are legitimate. However, if either operand on the stack is `NaN` prior to the execution of these instructions, `fcmpl` and `dcmpl` will leave `-1` on the stack, while `fcmpg` and `dcmpg` will leave `+1`.

D.5 The Java VM Instruction Set

Unlike the other (real) machines, such as the ARM, PowerPC, and 80x86, the Java VM doesn't really have a *minimal* instruction set. It's designed specifically for use by the Java language, so it's worthwhile to go over the entire instruction set.

Table D-3: Java VM Instructions

Instruction	Stack operands	Description
`aaload`	*`arrayRef`*, *`index`*	*`arrayRef`* is a pointer to an array of reference objects. *`index`* is an unsigned index into that array. This instruction replaces the items on the stack with the specified reference array element.
`aastore`	*`arrayRef`*, *`index`*, *`refValue`*	*`arrayRef`* is a pointer to an array of reference values. *`index`* is an unsigned index into that array. *`refValue`* is a reference value. This instruction pops the operands off the stack and stores the *`refValue`* into the specified array element.
`aconst_null`		This instruction pushes `null` onto the stack.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`{wide}` `aload` *`index`*	*`arrayRef`*, *`index`*, *`shortValue`* (TOS)	*`index`* is an unsigned byte (immediately following the `aload` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain an object reference. This instruction pushes that reference onto the stack. If the optional `{wide}` instruction prefix occurs immediately before `aload` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

`aload_0`
`aload_1`
`aload_2`
`aload_3`
		Special single-byte versions of `aload` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`anewarray` *`index`*$_{16}$	*`count`*	*`index`* is a 2-byte value following the `anewarray` opcode (the HO byte immediately follows the `anewarray` opcode, while the LO byte follows the HO byte in the instruction stream). This is an offset into the constant pool holding a symbolic reference to a class, array, or interface type. The *`count`* operand on the stack is an unsigned integer specifying the number of elements of the array to create. This instruction allocates storage for the array and leaves an array reference sitting on the TOS.
`areturn`	*`objRef`*	*`objRef`* is an object reference that is compatible with the return type of the current method/function. The system pops this reference off the stack, destroys the activation record of the current method, pushes the *`objRef`* onto the stack frame of the caller, and returns control to the caller.
`arrayLength`	*`arrayRef`*	Pops the reference to an array and replaces it with the length (in elements) of that array.
`{wide}` `astore` *`index`*	*`objRef`*	*`index`* is an unsigned byte (immediately following the `astore` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain an object reference. This instruction stores the object reference on the stack to the local variable at the specified offset. If the optional `{wide}` instruction prefix occurs immediately before `astore` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

`astore_0`
`astore_1`
`astore_2`
`astore_3`
	`objRef`	Special single-byte versions of `astore` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`athrow`	*`objRef`*	Throws an exception. Value on TOS must be an exception (sub)class object reference. Note that this instruction does not remove the *`objRef`* from the TOS.
`baload`	*`arrayRef`*, *`index`*	*`arrayRef`* is a pointer to an array of 8-bit values (such as `byte` or `boolean`). *`index`* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `byte` element at *`arrayRef`*`[`*`index`*`]` back onto the stack. This instruction sign-extends the 8-bit array element to 32 bits prior to pushing the value onto the stack.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`bastore`	*arrayRef*, *index*, *byteValue*	*arrayRef* is a pointer to an array of 8-bit objects. *index* is an unsigned index into that array. *byteValue* is a 32-bit value. This instruction pops *byteValue* off the stack, pops the next two operands off the stack, and then stores the LO 8 bits of *byteValue* into the array element at *arrayRef*[*index*].
`bipush sbyte`		`sbyte` is a signed 8-bit value (`-128..+127`). The `bipush` instruction sign-extends this to an `int` value and pushes it onto the stack.
`caload`	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to an array of (16-bit Unicode) characters. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the char element at *arrayRef*[*index*] back onto the stack. This instruction zero-extends the 16-bit char value to 32 bits prior to pushing it onto the stack.
`castore`	*arrayRef*, *index*, *charValue* (TOS)	*arrayRef* is a pointer to an array of 16-bit character objects. *index* is an unsigned index into that array. *charValue* is a 32-bit value (presumably containing a 16-bit Unicode character code in its LO 16 bits). This instruction pops the *charValue* value off the stack, pops the next two operands off the stack, and then stores the LO 16 bits of *charValue* into the array element at *arrayRef*[*index*].
`checkcast` $index_{16}$	*objRef*	*index* is a 2-byte value following the `checkcast` opcode (the HO byte immediately follows the `checkcast` opcode, while the LO byte follows the HO byte in the instruction stream). This is an offset into the constant pool holding a symbolic reference to a class, array, or interface type. The `checkcast` instruction verifies that *objRef* is of the type specified by $index_{16}$. If so, the stack is left unchanged; otherwise, this instruction throws an exception.
`d2f`	*doubleValue*	The double-precision floating-point value on TOS is converted to a single-precision floating-point value.
`d2i`	*doubleValue*	The double-precision floating-point value on TOS is converted to an integer value.
`d2l`	*doubleValue*	The double-precision floating-point value on TOS is converted to a long (64-bit) integer value.
`dadd`	*doubleValue*, *doubleValue*	This instruction pops the two double-precision values off the TOS, adds them, and pushes the double-precision sum back onto the TOS.
`daload`	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to an array of double-precision (64-bit) floating-point values. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `double` element at *arrayRef*[*index*] back onto the stack. This instruction pushes 64 bits onto the stack.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`dastore`	*`arrayRef`*, *`index`*, *`doubleValue`* (64 bits on TOS)	*`arrayRef`* is a pointer to an array of 64-bit `double` objects. *`index`* is an unsigned index into that array. *`doubleValue`* is a 64-bit double-precision floating-point value (occupying two 32-bit entries on the stack). This instruction pops the 64-bit *`doubleValue`* off the stack, pops the next two operands off the stack, and then stores *`doubleValue`* into the array element at *`arrayRef`*`[`*`index`*`]`.
`dcmpg`	*`doubleValue`*, *`doubleValue`*	This instruction compares the two double-precision values on TOS, NOS. It pushes `true` if NOS > TOS, and `false` otherwise (or if either or both operands are `NaN`).
`dcmpl`	*`doubleValue`*, *`doubleValue`*	This instruction compares the two double-precision values on TOS, NOS. It pushes `true` if NOS < TOS, and `false` otherwise (or if either or both operands are `NaN`).
`dconst_0` `dconst_1`		Pushes the double-precision constant value `0.0` or `1.0` onto the TOS.
`ddiv`	*`doubleValue`*, *`doubleValue`*	This instruction divides NOS by TOS and pushes the double-precision quotient back onto the TOS.
`{wide}` `dload` *`index`*		*`index`* is an unsigned byte (immediately following the `dload` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset and *`offset`* + 1 must contain a double-precision value. This instruction pushes that value onto the stack. If the optional `{wide}` instruction prefix occurs immediately before `dload` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

`dload_0`
`dload_1`
`dload_2`
`dload_3`
		Special single-byte versions of `dload` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`dmul`	*`doubleValue`*, *`doubleValue`*	This instruction multiplies NOS by TOS and pushes the double-precision product back onto the TOS.
`dneg`	*`doubleValue`*	This instruction negates the value on TOS.
`drem`	*`doubleValue`*, *`doubleValue`*	This instruction divides NOS by TOS and pushes the double-precision remainder back onto the TOS.
`dreturn`	*`doubleValue`*	*`doubleValue`* is a double-precision floating-point value. The current method/function must return a double result. The system pops this value off the stack, destroys the activation record of the current method, pushes the *`doubleValue`* onto the stack frame of the caller, and returns control to the caller.
`{wide}` `dstore` *`index`*	*`doubleValue`*	*`index`* is an unsigned byte (immediately following the `astore` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset (and *`offset`* + 1) must contain a double value. This instruction stores the double value on TOS to the local variable at the specified offset (and *`offset`* + 1). If the optional `{wide}` instruction prefix occurs immediately before `dstore` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description

`dstore_0`
`dstore_1`
`dstore_2`
`dstore_3`
	objRef	Special single-byte versions of `dstore` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`dsub`	*doubleValue*, *doubleValue*	This instruction subtracts TOS from NOS and pushes the double-precision difference back onto the TOS.
`dup`	*value*	The `dup` instruction duplicates the value on the TOS.
`dup_x1`	*value1*, *value2*	This instruction duplicates the value on TOS and pushes the duplicate value two entries down on the stack. That is, after `dup_x1` execution, the stack will contain *value1*, *value2*, *value1*.
`dup_x2`	$value1_{64}$, $value2_{64}$ or $value1_{32}$, $value2_{32}$, $value3_{32}$	Depending on the type of the operands on the TOS, this instruction duplicates the value on TOS and inserts it two or three entries down the stack. The difference has to do with whether the TOS is a `long`/`double` (that is, 64 bits) or some other data type (32 bits).
`dup2`	$value1_{32}$, $value2_{32}$ or $value_{64}$	Duplicates 64 bits on the TOS. If there are two 32-bit items on TOS, this instruction duplicates both of them; if there is a 64-bit value on TOS, this instruction duplicates only the single 64-bit item on TOS.
`dup2_x1`	$value1_{32}$, $value2_{32}$, $value3_{32}$ or $value1_{64}$, $value2_{32}$	Duplicates 64 bits on the TOS (a single `double`/`long` value or two 32-bit items) and pushes the result 32 bits below.
`dup2_x2`		Duplicates 64 bits on the TOS (a single `double`/`long` value or two 32-bit items) and pushes the result below (see the Java VM manual for complete details).
`f2d`	*floatValue*	Converts 32-bit single-precision value on TOS to a double.
`f2i`	*floatValue*	Converts 32-bit single-precision value on TOS to an `int`.
`f2l`	*floatValue*	Converts 32-bit single-precision value on TOS to a (64-bit) `long`.
`fadd`	*floatValue*, *floatValue*	Computes single-precision sum of NOS + TOS and leaves single-precision sum on TOS.
`faload`	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to an array of single-precision (32-bit) floating-point values. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `float` element at *arrayRef*[*index*] back onto the stack.
`fastore`	*arrayRef*, *index*, *floatValue* (TOS)	*arrayRef* is a pointer to an array of 32-bit `float` objects. *index* is an unsigned index into that array. *floatValue* is a 32-bit value. This instruction pops the *floatValue* off the stack, pops the next two operands off the stack, and then stores *floatValue* into the array element at *arrayRef*[*index*].
`fcmpg`	*singleValue*, *singleValue*	This instruction compares the two single-precision values on TOS, NOS. It pushes `true` if NOS > TOS, and `false` otherwise (or if either or both operands are `NaN`).

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`fcmpl`	*`singleValue`*, *`singleValue`*	This instruction compares the two single-precision values on TOS, NOS. It pushes `true` if NOS < TOS, and `false` otherwise (or if either or both operands are `NaN`).

`fconst_0`
`fconst_1`
`fconst_2`
		Pushes the single-precision constant value `0.0`, `1.0`, or `2.0` onto the TOS.

`fdiv`	*`singleValue`*, *`singleValue`*	This instruction divides NOS by TOS and pushes the single-precision quotient back onto the TOS.
`{wide}` `fload` *`index`*		*`index`* is an unsigned byte (immediately following the `fload` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain a single-precision value. This instruction pushes that value onto the stack. If the optional `{wide}` instruction prefix occurs immediately before `fload` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

`fload_0`
`fload_1`
`fload_2`
`fload_3`
		Special single-byte versions of `fload` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`fmul`	*`singleValue`*, *`singleValue`*	This instruction multiplies NOS by TOS and pushes the single-precision product back onto the TOS.
`fneg`	*`singleValue`*	This instruction negates the value on TOS.
`frem`	*`singleValue`*, *`singleValue`*	This instruction divides NOS by TOS and pushes the single-precision remainder back onto the TOS.
`freturn`	*`singleValue`*	*`singleValue`* is a single-precision floating-point value. The current method/function must return a float result. The system pops this value off the stack, destroys the activation record of the current method, pushes the *`singleValue`* onto the stack frame of the caller, and returns control to the caller.
`{wide}` `fstore` *`index`*	*`singleValue`*	*`index`* is an unsigned byte (immediately following the `fstore` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain a float value. This instruction stores the float value on TOS to the local variable at the specified offset. If the optional `{wide}` instruction prefix occurs immediately before `fstore` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

`fstore_0`
`fstore_1`
`fstore_2`
`fstore_3`
	`singleValue`	Special single-byte versions of `fstore` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`fsub`	*`singleValue`*, *`singleValue`*	This instruction subtracts TOS from NOS and pushes the single-precision difference back onto the TOS.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
getfield *index*$_{16}$	*objRef*	*index* is a 2-byte value following the getfield opcode (the HO byte immediately follows the opcode, while the LO byte follows the HO byte in the instruction stream). This is an offset into the constant pool holding a symbolic reference to a field. The getfield instruction replaces *objRef* by the value of the specified field.
getstatic *index*$_{16}$		*index* is a 2-byte value following the getstatic opcode (the HO byte immediately follows the getstatic opcode, while the LO byte follows the HO byte in the instruction stream). This is an offset into the constant pool holding a symbolic reference to a static field. The getfield instruction pushes the value of the specified static field onto TOS.
goto *index*$_{16}$		*index* is a 2-byte signed integer value following the goto opcode (the HO byte immediately follows the goto opcode, while the LO byte follows the HO byte in the instruction stream). This transfers control to the bytecode that is located at the specified offset from the current instruction.
goto_w *index*$_{32}$		*index* is a 4-byte signed integer value following the goto_2 opcode (the HO byte immediately follows the opcode, and the remaining bytes follow, in order, down to the LO byte). This transfers control to the bytecode that is located at the specified offset from the current instruction.
i2b	*intValue*	The integer value on TOS is truncated to 8 bits; the result is then sign-extended to 32 bits and pushed back onto the stack.
i2c	*intValue*	The integer value on TOS is truncated to 8 bits; the result is then zero-extended to 32 bits and pushed back onto the stack.
i2d	*intValue*	The integer value on TOS is converted to a double, and the result is pushed back onto the stack.
i2f	*intValue*	The integer value on TOS is converted to a float, and the result is pushed back onto the stack.
i2l	*intValue*	The integer value on TOS is sign-extended to 64 bits and pushed back onto the stack.
i2s	*intValue*	The integer value on TOS is truncated to 16 bits; the result is then sign-extended to 32 bits and pushed back onto the stack.
iadd	*intValue*, *intValue*	This instruction computes the sum of NOS + TOS, leaving the sum on TOS.
iaload	*arrayRef*, *index* (TOS)	*arrayRef* is a pointer to an int array. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the int element at *arrayRef*[*index*] back onto the stack.
iand	*intValue*, *intValue*	This instruction computes the bitwise logical AND of NOS + TOS, leaving the result on TOS.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`iastore`	*arrayRef*, *index*, *intValue*	*arrayRef* is a pointer to an array of 32-bit `int` objects. *index* is an unsigned index into that array. *intValue* is a 32-bit value. This instruction pops the *intValue* off the stack, pops the next two operands off the stack, and then stores *intValue* into the array element at *arrayRef*[*index*].

`iconst_m1`
`iconst_0`
`iconst_1`
`iconst_2`
`iconst_3`
`iconst_4`
`iconst_5`
		These instructions push the 32-bit integer constant –1, 0, 1, 2, 3, 4, or 5 onto the TOS.

`idiv`	*intValue*, *intValue*	This instruction divides NOS by TOS and pushes the integer quotient back onto the TOS.
`ifacmpeq` $offset_{16}$	*objRef*, *objRef*	If NOS is equal to TOS, then transfer control to the location specified by $offset_{16}$. $offset_{16}$ is a signed 16-bit offset specifying a displacement from the current instruction.
`ifacmpne` $offset_{16}$	*objRef*, *objRef*	If NOS is not equal to TOS, then transfer control to the location specified by $offset_{16}$. $offset_{16}$ is a signed 16-bit offset specifying a displacement from the current instruction.
`ificmpeq` $offset_{16}$ `ificmpne` $offset_{16}$ `ificmplt` $offset_{16}$ `ificmple` $offset_{16}$ `ificmpgt` $offset_{16}$ `ificmpge` $offset_{16}$	*intValue*, *intValue*	Compares integer value NOS to TOS, then transfers control to the location specified by $offset_{16}$ if the particular condition is `true`. $offset_{16}$ is a signed 16-bit offset specifying a displacement from the current instruction.
`ifeq` $offset_{16}$ `ifne` $offset_{16}$ `iflt` $offset_{16}$ `ifle` $offset_{16}$ `ifgt` $offset_{16}$ `ifge` $offset_{16}$	*intValue*	Compares 32-bit `int` on TOS to `0`, then transfers control to the location specified by $offset_{16}$ if the particular condition is `true`. $offset_{16}$ is a signed 16-bit offset specifying a displacement from the current instruction.
`ifnonnull` $offset_{16}$		Compares TOS to `null`, then transfers control to the location specified by $offset_{16}$ if not equal. $offset_{16}$ is a signed 16-bit offset specifying a displacement from the current instruction.
`ifnull` $offset_{16}$		Compares TOS to `null`, then transfers control to the location specified by $offset_{16}$ if equal. $offset_{16}$ is a signed 16-bit offset specifying a displacement from the current instruction.
`{wide}` `iinc` *index*, *const*		*index* is an unsigned byte (immediately following the `iinc` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain an integer value. *const* is a signed 8-bit integer constant. The `iinc` instruction adds the constant to the local variable specified by *index*. If the `{wide}` prefix is present, then *index* is a 16-bit index into the local stack frame.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
{wide} iload *index*		*index* is an unsigned byte (immediately following the iload opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain an integer value. This instruction pushes that value onto the stack. If the optional {wide} instruction prefix occurs immediately before iload *index*, then *index* is an unsigned 16-bit integer offset.
iload_0 iload_1 iload_2 iload_3		Special single-byte versions of iload *index* that encode offsets 0, 1, 2, or 3 into the instruction opcode.
imul	*intValue*, *intValue*	This instruction multiplies NOS by TOS and pushes the integer product back onto the TOS.
ineg	*intValue*	This instruction negates the integer value on TOS.
instanceof *indext*16	*objRef*	*indext*16 is a 2-byte value following the instanceof opcode (the HO byte immediately follows the instanceof opcode, while the LO byte follows the HO byte in the instruction stream). This is an offset into the constant pool holding a symbolic reference to a class, array, or interface type. The instanceof instruction pushes true (1) or false (0) on the stack if the *objRef* value is an instance of the class specified by *indext*16.
invokedynamic $index_{16}$	*arg1*, *arg2*, . . .	$index_{16}$ is an unsigned integer specifying an entry into the constant pool specifying the method to invoke. The runtime constant pool entry specifies the name (signature) for the method and the class it is associated with. The Java VM uses this instruction to invoke lambda methods (unnamed function blocks) that don't have a specific runtime object instance associated with them.
invokeinterface $index_{16}$, *count*	*objRef*, *arg1*, *arg2*, . . .	The $index_{16}$ operand provides a 16-bit index into the constant pool for the interface type and the method within that type. The *count* operand specifies the number of arguments passed to the method. This instruction pops the arguments from the stack, creates a new activation record for the function using the popped arguments, and then invokes the specified method associated with the the *objRef*.
invokespecial $index_{16}$	*objRef*, *arg1*, *arg2*, . . .	The $index_{16}$ operand provides a 16-bit index into the symbolic constant pool for the method to invoke. The Java VM uses this instruction to call superclass (ancestor) methods of the object's current class.
invokestatic $index_{16}$	*arg1*, *arg2*, . . .	The $index_{16}$ operand provides a 16-bit index into the symbolic constant pool for the static method to invoke. This instruction pops the arguments, creates a new activation record for the static method (including the popped arguments), and then calls the specified method.
invokevirtual $index_{16}$	*objRef*, *arg1*, *arg2*, . . .	The $index_{16}$ operand provides a 16-bit index into the symbolic constant pool for the method to invoke. The Java VM uses this instruction to call the specified method of the *objRef*'s class.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`ior`	*`intValue`*, *`intValue`*	Computes the inclusive-OR of NOS and TOS and leaves the result on TOS.
`irem`	*`intValue`*, *`intValue`*	This instruction divides NOS by TOS and pushes the integer remainder back onto the TOS.
`ireturn`	*`intValue`*	*`intValue`* is an integer value. The current method/function must return an `int` result. The system pops this value off the stack, destroys the activation record of the current method, pushes the *`intValue`* onto the stack frame of the caller, and returns control to the caller.
`ishl`	*`intValue`*, *`intValue`*	This instruction shifts the value in NOS to the left the number of bit positions specified by (the LO 5 bits of) TOS and pushes the 32-bit result back onto the TOS.
`ishr`	*`intValue`*, *`intValue`*	This instruction shifts the value in NOS to the right the number of bit positions specified by (the LO 5 bits of) TOS and pushes the 32-bit result back onto the TOS. The right shift is an arithmetic right shift, copying the sign bit into bit position 30 after each shift operation.
`{wide}` `istore` *`index`*	*`intValue`*	*`index`* is an unsigned byte (immediately following the `istore` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain an `int` value. This instruction stores the *`intValue`* on TOS to the local variable at the specified offset. If the optional `{wide}` instruction prefix occurs immediately before `istore` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

`istore_0`
`istore_1`
`istore_2`
`istore_3`
	`intValue`	Special single-byte versions of `istore` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`isub`	*`intValue`*, *`intValue`*	This instruction subtracts TOS from NOS and pushes the `int` difference back onto the TOS.
`iushr`	*`intValue`*, *`intValue`*	This instruction shifts the value in NOS to the right the number of bit positions specified by (the LO 5 bits of) TOS and pushes the 32-bit result back onto the TOS. The right shift is a logical right shift, shifting `0`s into the HO bit position after each shift.
`ixor`	*`intValue`*, *`intValue`*	Computes the exclusive-OR of NOS and TOS and leaves the result on TOS.
`jsr` $index_{16}$		The $index_{16}$ operand provides a signed 16-bit offset from the `jsr` instruction. The `jsr` instruction pushes a return address onto the stack and then jumps to the Java VM bytecode at the offset specified by the instruction.
`jsr_w` $index_{32}$		The $index_{32}$ operand provides a signed 32-bit offset from the `jsr_w` instruction. The `jsr_w` instruction pushes a return address onto the stack and then jumps to the Java VM bytecode at the offset specified by the instruction.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`l2d`	*`longValue`*	Converts the 64-bit long value on TOS to a double-precision floating-point value.
`l2f`	*`longValue`*	Converts the 64-bit long value on TOS to a single-precision floating-point value.
`l2i`	*`longValue`*	Converts the 64-bit long value on TOS to a 32-bit integer value.
`ladd`	*`longValue`*, *`longValue`*	This instruction computes the sum of NOS + TOS, leaving the sum on TOS.
`laload`	*`arrayRef`*, *`index`*	*`arrayRef`* is a pointer to a `long` array. *`index`* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the `long` element at *`arrayRef`*`[`*`index`*`]` back onto the stack.
`land`	*`longValue`*, *`longValue`*	This instruction computes the bitwise logical AND of NOS + TOS, leaving the result on TOS.
`lastore`	*`arrayRef`*, *`index`*, *`longValue`*	*`arrayRef`* is a pointer to an array of 64-bit `long` objects. *`index`* is an unsigned index into that array. *`longValue`* is a 64-bit value. This instruction pops the *`longValue`* off the stack, pops the next two operands off the stack, and then stores *`intValue`* into the array element at *`arrayRef`*`[`*`index`*`]`.
`lcmp`	*`longValue`*, *`longValue`*	This instruction compares the 64-bit NOS to TOS and leaves the 32-bit value `-1`, `0`, or `+1` on the stack if NOS < TOS, NOS = TOS, or NOS > TOS, respectively.
`lconst_0` `iconst_1`		These instructions push the 64-bit integer constants `0` or `1` onto the TOS.
`ldc` *`index`*		*`index`* is an unsigned 8-bit constant that specifies the index of an integer, float, string reference, object reference, or method reference in the constant pool. This instruction pushes the specified constant onto the stack.
`ldc_w` *`index`*		*`index`* is an unsigned 16-bit constant that specifies the index of an integer, float, string reference, object reference, or method reference in the constant pool. This instruction pushes the specified constant onto the stack.
`ldc2_w` *`index`*		*`index`* is an unsigned 16-bit constant that specifies the index of a `long` or `double` constant in the constant pool. This instruction pushes the specified constant onto the stack.
`ldiv`	*`longValue`*, *`longValue`*	This instruction divides NOS by TOS and pushes the integer quotient back onto the TOS.
`{wide}` `lload` *`index`*		*`index`* is an unsigned byte (immediately following the `lload` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain a `long` value. This instruction pushes that value onto the stack. If the optional `{wide}` instruction prefix occurs immediately before `lload` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description

`lload_0`
`lload_1`
`lload_2`
`lload_3`
		Special single-byte versions of `lload` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`lmul`	*`longValue`*, *`longValue`*	This instruction multiplies NOS by TOS and pushes the `long` product back onto the TOS.
`ineg`	*`longValue`*	This instruction negates the `long` value on TOS.
`lookupswitch` *`default`*, *`npairs`*, *`pairs`*	*`switchValue`*	*`pairs`* is a list of tuples, each containing a signed 32-bit integer value and a signed 32-bit integer offset. *`default`* is a 32-bit offset. *`npairs`* is an unsigned 32-bit integer specifying the number of entries in the *`pairs`* table. This instruction pops the *`switchValue`* integer off the stack and searches for the value in the *`pairs`* list. If it finds the value, the instruction transfers control to the corresponding offset. If it does not find the value, the instruction transfers control to the offset specified by *`default`*.
`lor`	*`longValue`*, *`longValue`*	Computes the inclusive-OR of NOS and TOS and leaves the result on TOS.
`lrem`	*`longValue`*, *`longValue`*	This instruction divides NOS by TOS and pushes the integer remainder back onto the TOS.
`lreturn`	*`longValue`*	*`longValue`* is an integer value. The current method/function must return a `long` result. The system pops this value off the stack, destroys the activation record of the current method, pushes the *`longValue`* onto the stack frame of the caller, and returns control to the caller.
`lshl`	*`longValue`*, *`intValue`*	This instruction shifts the value in NOS to the left the number of bit positions specified by (the LO 5 bits of) TOS and pushes the 64-bit result back onto the TOS.
`lshr`	*`longValue`*, *`intValue`*	This instruction shifts the value in NOS to the right the number of bit positions specified by (the LO 5 bits of) TOS and pushes the 64-bit result back onto the TOS. The right shift is an arithmetic right shift, copying the sign bit into bit position 62 after each shift operation.
`{wide}` `lstore` *`index`*	*`longValue`*	*`index`* is an unsigned byte (immediately following the `lstore` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain a `long` value. This instruction stores the *`longValue`* on TOS to the local variable at the specified offset. If the optional `{wide}` instruction prefix occurs immediately before `lstore` *`index`*, then *`index`* is an unsigned 16-bit integer offset.

`lstore_0`
`lstore_1`
`lstore_2`
`lstore_3`
	`longValue`	Special single-byte versions of `lstore` *`index`* that encode offsets 0, 1, 2, or 3 into the instruction opcode.

`lsub`	*`longValue`*, *`longValue`*	This instruction subtracts TOS from NOS and pushes the `long` difference back onto the TOS.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`lushr`	*longValue*, *intValue*	This instruction shifts the value in NOS to the right the number of bit positions specified by (the LO 5 bits of) TOS and pushes the 64-bit result back onto the TOS. The right shift is a logical right shift, shifting `0`s into the HO bit position after each shift.
`lxor`	*longValue*, *longValue*	Computes the exclusive-OR of NOS and TOS and leaves the result on TOS.
`monitorenter`	*objRef*	Enter a monitor for object specified by *objRef* (that is, lock the object for exclusive use).
`monitorexit`	*objRef*	Leave a monitor for object specified by *objRef* (that is, free the object from exclusive use).
`multianewarray` *index*$_{16}$, *dimensions*	*count1*, *count2*, . . .	The *dimensions* operand is an unsigned byte that must be greater than or equal to `1` (specifying the number of dimensions for the array). The *index*$_{16}$ operand is an unsigned 16-bit integer providing an offset into the constant pool that holds the symbolic type information for the array's base type. The stack must hold dimensions integer values, where each value specifies the number of elements for that dimension of the array. This instruction allocates storage for the array and leaves an object reference to the array on TOS.
`new` *index*$_{16}$		The *index*$_{16}$ operand is an unsigned 16-bit integer providing an offset into the constant pool that holds the symbolic type information for the object's type. This instruction allocates storage for the new object and leaves an object reference on TOS. Note that this instruction will also initialize the object.

`newarray` *atype*	*count*	The *atype* operand is one of the following special values that specifies the element type:
		`4: boolean`
		`5: char`
		`6: float`
		`7: double`
		`8: byte`
		`9: short`
		`10: int`
		`11: long`
		The stack must hold the number of elements for the array. This instruction allocates storage for the array and leaves an object reference to the array on TOS.

`nop`		No operation; does nothing.
`pop`		Pops a 32-bit value off the stack.
`pop2`		Pops a 64-bit `double` or `long` off the stack.
`putfield` *index*$_{16}$	*objRef*, *value*	The *index*$_{16}$ operand is an unsigned 16-bit integer providing an offset into the constant pool that holds the symbolic type information for the object's type. This instruction stores the *value* on TOS into the field specified by *index*$_{16}$ in the object specified by *objRef*.

(continued)

Table D-3: Java VM Instructions (continued)

Instruction	Stack operands	Description
`putstatic` *index*$_{16}$	*value*	The *index*$_{16}$ operand is an unsigned 16-bit integer providing an offset into the constant pool that holds the symbolic type information for the static field. This instruction stores the *value* on TOS into the field specified by *index*$_{16}$.
`{wide}` `ret` *index*		*index* is an unsigned byte (immediately following the `ret` opcode in memory) that provides an offset into the current stack frame. The local variable at that offset must contain a return address. This instruction copies that return address into the Java VM and continues execution from there. Note that the `ret` instruction returns from a call created by the `jsr` and `jsr_w` instructions.
`return`		Returns from a method whose type is `void`. Destroys the activation record associated with the method and returns to the activation record of the method's caller.
`saload`	*arrayRef*, *index*	*arrayRef* is a pointer to a `short` array. *index* is an unsigned index into that array. This instruction pops the two operands off the stack and pushes the short element at *arrayRef*[*index*] back onto the stack (sign-extending it to 32 bits).
`sastore`	*arrayRef*, *index*, *shortValue*	*arrayRef* is a pointer to an array of 16-bit short objects. *index* is an unsigned index into that array. *shortValue* is a 16-bit value. This instruction pops the *shortValue* off the stack, pops the next two operands off the stack, and then stores *shortValue* into the array element at *arrayRef*[*index*].
`sipush` *const*		*const* is a 16-bit signed integer. This instruction sign-extends that constant to 32 bits and pushes the result onto the stack.
`swap`	*value*, *value*	This instruction swaps the two 32-bit values on NOS and TOS.
`tableswitch` *default*, *lowValue*, *highValue*, *jumpTable*	*switchValue*	*default* is a 32-bit signed offset. *lowValue* and *highValue* are 32-bit signed integers. If *switchValue* is outside the range *lowValue*..*highValue*, then this instruction transfers control to the location specified by *default* (offset from the current instruction). *jumpTable* is a table of (*highValue* – *lowValue* + 1) 32-bit signed offsets. If *switchValue* is in the range *lowValue*..*highValue*, then this instruction transfers control via the table entry at index (*switchValue* – *lowValue*).

D.6 For More Information

The full Java VM instruction set appears in the Java Virtual Machine Specification. You can find the latest edition of this document on Oracle's website or at *www.writegreatcode.com*.