

# The Java Virtual Machine

Norman Matloff and Thomas Fifield\*  
University of California at Davis  
© 2001-2007, N. Matloff

December 11, 2006

## Contents

<b>1</b>	<b>Background Needed</b>	<b>3</b>
<b>2</b>	<b>Goal</b>	<b>3</b>
<b>3</b>	<b>Why Is It a “Virtual” Machine?</b>	<b>3</b>
<b>4</b>	<b>The JVM Architecture</b>	<b>4</b>
4.1	Registers . . . . .	4
4.2	Memory Areas . . . . .	5
<b>5</b>	<b>First Example</b>	<b>6</b>
5.1	Java Considerations . . . . .	6
5.2	How to Inspect the JVM Code . . . . .	7
5.3	The Local Variables Section of main() . . . . .	8
5.4	The Call of Min() from main() . . . . .	8
5.5	Accessing Arguments from within Min() . . . . .	9
5.6	Details on the Action of Jumps . . . . .	9
5.7	Multibyte Numbers Embedded in Instructions . . . . .	9
5.8	Calling Instance Methods . . . . .	10
5.9	Creating and Accessing Arrays . . . . .	11
5.10	Constructors . . . . .	11
5.11	Philosophy Behind the Design of the JVM . . . . .	12

---

\*Thomas, a student in ECS 50, wrote portions of Section 7 (excluding the source code for **NumNode.java**), and added the corresponding new instructions to Appendix A.

*CONTENTS*

*CONTENTS*

5.11.1	Instruction Structure . . . . .	12
5.11.2	Stack Architecture . . . . .	12
5.11.3	Safety . . . . .	13
<b>6</b>	<b>Another Example</b>	<b>13</b>
<b>7</b>	<b>Yet Another Example</b>	<b>16</b>
<b>A</b>	<b>Overview of JVM Instructions</b>	<b>21</b>
<b>B</b>	<b>References</b>	<b>26</b>

### 3 WHY IS IT A “VIRTUAL” MACHINE?

## 1 Background Needed

In what follows, it is assumed that the reader has at least a rudimentary knowledge of Java. See the author’s Java tutorial at <http://heather.cs.ucdavis.edu/~matloff/java.html> for a 30-minute introduction to the Java language. But really, for our purposes, anyone who understands C++ should have no trouble following the Java code here.

## 2 Goal

Keep in mind that our goal here is to study the Java (virtual) *machine*, not the Java language. We wish to see how the machine works, and—this is very important—see why its developers chose to design certain features of the machine the way they did.

## 3 Why Is It a “Virtual” Machine?

You may have heard about the **Java virtual machine** (JVM), associated with the Java language. What is really going on here?

The name of any Java source file has a **.java** suffix, just like C source file names end in **.c**. Suppose for example our Java program source code is all in one file, **x.java**, and that for instance we are doing our work on a PC running Linux.

We first compile, using the Java compiler, **javac**:

```
1 % javac -g x.java
```

(The **-g** option saves the symbol table for use by a debugger.)

This produces the executable Java file, **x.class**, which contains machine language, called **byte code**, to run on a “Java machine.” But we don’t have such a machine.

Instead, we have a program that *emulates* the operation of such a machine. This, of course, is the reason for the ‘V’ in “JVM.” The emulator (**interpreter**) program is named **java**. Note that in our case here, **java** will be a program running the Intel machine language of our PC.<sup>1</sup>

We then run our program:

```
1 % java x
```

Note that Java not only runs on a virtual machine, it also is in some sense running under a virtual operating system. For example, the analog of C’s **printf()** function in Java is **System.out.println()**. Recall that (if our real machine is running UNIX) a **printf()** call in a C program calls the **write()** function in the OS. But this does not happen in the Java case; our OS is running on our real machine, but our Java program is running on the JVM. What actually happens is that **System.out.println()** makes a call to what amounts to an OS in the JVM, and the latter calls **write()** on the real machine.

---

<sup>1</sup>And, for that matter, **javac** would be an Intel machine-language program too.

## 4 THE JVM ARCHITECTURE

By the way, JVM chips—i.e. chips that run Java byte code—do exist, but they are not in common use. Also, note that newer versions of GCC include GCJ, which will compile Java to machine code for your machine, e.g. Intel machine language if you are on a PC. This is really nice, as it produces faster-running programs, and it is far more common than Java chips, but still, the vast majority of Java code is executed on emulators.

### 4 The JVM Architecture

The JVM is basically a stack-based machine, with a 32-bit word size, using 2s complement arithmetic.

The term *stack-based machine* means that almost all operands are on the stack, and they are implicit to the instruction. For example, contrast the Intel instruction

```
1 addl %eax, %ebx
```

to the JVM instruction

```
1 iadd
```

In the Intel case, the programmer has a choice of which two things to add together, and explicitly states them. In the JVM case, the programmer has no choice; the two operands must be the top two elements of the stack. And since those operands must be the top two elements of the stack, there is no need to specify them in the instruction, so the instruction consists only of an op code, no explicit operands.<sup>2</sup>

Before continuing, it's vital that we discuss what the term “the” stack means above. Unlike the case of C/C++/assembly language, where there is a stack for each *process*, here we will have a stack for each *method call*.<sup>3</sup> So, the term “the stack” in the material here must always be thought of as meaning “the stack for whatever method is currently running.” This is referred to as the **operand stack**.

On the other hand, there *is* something analogous to the stack you have worked with before in C/C++/assembly language. Recall that in that situation, all the function calls use a common stack, and all the data for a given function call—its arguments, return value and local variables—comprise the **stack frame** for that function call. Java has this too. There is a stack frame for each method call, and they are placed in the overall stack in the same last-in, first-out fashion that we saw for function calls in C/C++/assembly language. Each stack frame in Java will in turn consist of the operand stack for that method call, a section for arguments and local variables, and some other data.

#### 4.1 Registers

The JVM register set<sup>4</sup> is fairly small:

- **pc**: program counter

---

<sup>2</sup>Historical note: A number of famous machines popular in the past, such as the Burroughs mainframe and HP 3000 minicomputer series, were stack-based.

<sup>3</sup>Recall that in Java, we use the term *method* instead of *function*.

<sup>4</sup>Remember, in the case of a real Java chip, these would be real registers, but in the JVM setting, these registers, as well as the instruction set, are simulated by the **java** program.

- **optop**: pointer to the top of the operand stack for the currently-active method
- **frame**: pointer to the stack frame of the currently-active method
- **vars**: pointer to the beginning of the local variables of the currently-active method

## 4.2 Memory Areas

- the (general) Stack:

A method call produces a new stack frame, which is pushed onto the Stack for that program,<sup>5</sup> and a return pops the frame from the program's Stack.

A stack frame is subdivided into the following.

- a Local Variables section:

All the local variables and arguments for the method are stored here, one per **slot** (i.e. one variable per word), with the arguments stored first and then the locals. The arguments and locals are stored in order of their declaration. In the case in which the method is an instance method,<sup>6</sup> slot 0 will contain a pointer to the object on which this method is operating, i.e. the object referred to as **this** in the program's source code.

- an Operand Stack section:

This is the area on which the method's instructions operate. As noted, almost all JVM instructions are stack-based; e.g. an "add" instruction pops the top two elements of the stack, adds them, and pushes the sum back onto the stack. So the operand stack portion of a method's stack frame is what we are referring to when we refer to an instruction as operating on "the" stack.

- a Frame Data section:

We will not go into the details of this, but for example the Frame Data section of a called method's stack frame will include a pointer to the caller's stack frame. This enables return to the caller when the called method finishes, and enables the latter to put the return value, if any, into the caller's stack frame.

- the Method Area:

The classes used by the executing program are stored here. This includes:

- the bytecode and access types of the methods of the class (similar to the **.text** segment of a UNIX program, except for the access types)
- the **static** variables of the class (their values and access types, i.e **public**, **private** etc.)

The **pc** register points to the location within the Method Area of the JVM instruction to be executed next.

The Method Area also includes the Constant Pool. It contains the string and numeric literals used by the program, e.g. the 1.2 in

<sup>5</sup>More precisely, for that thread, since many Java programs are threaded.

<sup>6</sup>In general object-oriented programming terminology, a function is called an **instance** function if it applies specifically to an instance of a class. In C++ and Java, this is signified by not being declared **static**. The alternate form is **class** methods, which apply to all objects of that class, and is signified in C++/Java by being declared **static**.

## 5 FIRST EXAMPLE

```
1 float W = 1.2;
```

and also contains information on where each method is stored in the Method Area, as well as the static variables for the various classes.

- the Heap:

This is where Java objects exist. Whenever the Java **new** operation is invoked to create an object, the necessary memory for the object is allocated within the heap.<sup>7</sup> This space will hold the instance variables for the object, and a pointer to the location of the object's class in the Method Area.

## 5 First Example

Consider the following source code, **Minimum.java**:

```
1 public class Minimum {
2
3     public static void main(String[] CLArgs)
4
5     { int X,Y,Z;
6
7         X = Integer.parseInt(CLArgs[0]);
8         Y = Integer.parseInt(CLArgs[1]);
9         Z = Min(X,Y);
10        System.out.println(Z);
11    }
12
13    public static int Min(int U, int V)
14
15    { int T;
16
17        if (U < V) T = U;
18        else T = V;
19        return T;
20    }
21 }
```

### 5.1 Java Considerations

In Java, there are no global quantities of any kind. That not only means no global variables, but also no free-standing, functions—every function must be part of some class. So for example we see here that even **main()** is part of a class, the class **Minimum**. (That class in turn must be named after the name of the source file, **Minimum.java**.)

The argument for **main()**, **CLArgs**, is the set of command-line arguments, i.e. what we normally name **argv** in C/C++.<sup>8</sup> There is nothing like **argc**, though. The reason is that **CLArgs** is of type **String []**, i.e. an array of strings, and in Java even arrays are objects. Array objects include a member variable showing the length of the array, so information analogous to **argc** is incorporated in **CLArgs**, specifically in **CLArgs.length**—the value of the **length** member variable of the array class, for the instance **CLArgs** of this class.

---

<sup>7</sup>Just as in C, where a call to **malloc()** results in memory space being allocated from the heap, and just as in C++, where a call to **new** results in space being taken from the heap.

<sup>8</sup>Note that in C/C++ we can name that parameter whatever we want, though it is customary to call it **argv**.

The function `Integer.parseInt()` is similar to `atoi()` in C/C++. Note, though, since Java does not allow free-standing functions, `parseInt()` must be part of a class, and it is—the `Integer` class, which has lots of other functions as well.

We use the compiler, `javac`, to produce the class file, `Minimum.class`. The latter is what is executed, when we run the Java interpreter, `java`.

## 5.2 How to Inspect the JVM Code

We can use another program, `javap`, to disassemble the contents of `Minimum.class`:<sup>9</sup>

```
% javap -c Minimum
Compiled from Minimum.java
public class Minimum extends java.lang.Object {
    public Minimum();
    public static void main(java.lang.String[]);
    public static int Min(int, int);
}

Method Minimum()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 return

Method void main(java.lang.String[])
  0 aload_0
  1 iconst_0
  2 aaload
  3 invokestatic #2 <Method int parseInt(java.lang.String)>
  6 istore_1
  7 aload_0
  8 iconst_1
  9 aaload
 10 invokestatic #2 <Method int parseInt(java.lang.String)>
 13 istore_2
 14 iload_1
 15 iload_2
 16 invokestatic #3 <Method int Min(int, int)>
 19 istore_3
 20 getstatic #4 <Field java.io.PrintStream out>
 23 iload_3
 24 invokevirtual #5 <Method void println(int)>
 27 return

Method int Min(int, int)
  0 iload_0
  1 iload_1
  2 if_icmpge 10
  5 iload_0
  6 istore_2
  7 goto 12
 10 iload_1
 11 istore_2
 12 iload_2
 13 ireturn
```

Note that each “line number” is actually an offset, i.e. the distance in bytes of the given instruction from the beginning of the given method.

<sup>9</sup>So, this listing here is similar to the output of `gcc -S` in a C/C++ context.

### 5.3 The Local Variables Section of `main()`

Here is what the Local Variables Section of `main()`'s stack frame looks like:

slot	variable
0	pointer to CLArgs
1	X
2	Y
3	Z

### 5.4 The Call of `Min()` from `main()`

Now consider the call to `Min()`. The code

```
1      Z = Min(X, Y);
```

gets compiled to

```
14 iload_1
15 iload_2
16 invokestatic #3 <Method int Min(int, int)>
19 istore_3
```

As in a classical architecture, the arguments for a call are pushed onto `main()`'s operand stack, as follows. The `iload_1` (“integer load”) instruction in offset 14 of `main()` pushes slot 1 to the operand stack. Since slot 1 in `main()` contains X, this means that X will be pushed onto `main()`'s operand stack. The instruction in offset 15 will then push Y.<sup>10</sup>

The `invokestatic` instruction is the function call. By looking up information which is stored regarding `Min()`, this instruction knows that `Min()` has two arguments.<sup>11</sup> Thus the instruction knows that it must pop two values from `main()`'s operand stack, which are the arguments (placed there earlier by `iload_1` and `iload_2`), and it places them in the first two slots of `Min()`'s Local Variables Area.

Note that in the instruction `iload_1`, for example, the ‘1’ is a part of the instruction name, not an operand. In fact, none of `iload_0`, `iload_1`, `iload_2` and `iload_3` has an operand; there is a separate instruction, i.e. separate op code, for each slot.

But there is no `iload_5`, `iload_6` and so on. What if you need to load from some other slot, say slot 12? Then you would use the `iload` instruction, which is fully general, i.e. can load from any slot, including but not limited to slots 0, 1, 2 and 3. Note that `iload_0` etc. are one-byte instructions, while `iload` is a two-byte instruction, with the second byte being the slot number, so we should use those special instructions for the cases of slots 0, 1, 2 and 3. The designers of the JVM knew that most accesses to local variables would be to slots 0-3, so they decided to include special one-byte instructions to load from those slots, thus making for smaller code. Similar statements hold for `istore`, etc.

<sup>10</sup>So, the arguments must be pushed in the order in which they appear in the call, unlike C/C++.

<sup>11</sup>Again unlike C/C++, where the number of arguments in the call is unknown to the compiler.

## 5.5 Accessing Arguments from within `Min()`

Now, upon entry to `Min()`, here is what that function's Local Variables Section will look like:

slot	variable
0	U
1	V
2	T

Since the arguments `U` and `V` are in slots 0 and 1, they will be accessed with instructions like `iload_0` and `iload_1`.

The main new instruction in `Min()` is `if_icmpge` (“if integer compare greater-than-or-equal”) in offset 2. Let's refer to the top element of the current (i.e. `Min()`'s) operand stack as `op2` and the next-to-top element as `op1`. The instruction pops these two elements off the operand stack, compares them, and then jumps to the branch target if  $op1 \geq op2$ . Again, keep in mind that these items on `Min()`'s operand stack were placed there by the `iload_0` and `iload_1` instructions.

`Min()`'s `ireturn` instruction then pops the current (i.e. `Min()`'s) operand stack and pushes the popped value on top of the caller's (i.e. `main()`'s) operand stack. In the case of methods which do not have return values, we use the `return` instruction instead of `ireturn`.

## 5.6 Details on the Action of Jumps

For jump instructions, the branch target is specified as the distance from the current instruction to the target. As can be seen in the JVM assembler code above, the target for the `if_icmpge` instruction in offset 2 of `Min()` is offset 10 (an `iload_1` instruction). Since our `if_icmpge` instruction is in offset 2, the distance will be 8, i.e. `0x0008`.<sup>12</sup> Those latter two bytes comprise the second and third bytes of the instruction. Note that the branch target must be within the current method; JVM will announce a runtime error if not.

From the Sun JVM specifications (see below), we know that the op code for the `if_icmpge` instruction is `0xa2`. Thus the entire instruction should be `a2 00 08`, and this string of three bytes should appear in the file `Minimum.class`. Running the command

```
1 % od -t x1 Minimum.class
```

on a UNIX machine, we see that `a2 00 08` is indeed in that file, in bytes 1255-1257 octal (685-687 decimal).

## 5.7 Multibyte Numbers Embedded in Instructions

Note in the jump example above, the instruction was `a0 00 08`, with `a0` being the op code and `00 08` being the distance to the jump target. Note that the order of bytes in memory will be that seen above, i.e. `a0 00 08`, so `00` will be in a lower-address byte than `08`. So, the question arises as to whether the embedded constant is intended to be `0008`, i.e. 8, or `8000`. The answer is the former.

In other words, numbers embedded within instructions are meant to be interpreted as big-endian.

<sup>12</sup>Unlike the Intel (and typical) case, the jump distance in JVM is calculated from the jump instruction, not from the one following it.

## 5.8 Calling Instance Methods

When an instance method is called, we use the **invokevirtual** instruction instead of **invokestatic**. To understand how that works, recall that in general OOP programming, an instance method includes an extra “unofficial” argument in the form of a pointer to the instance on which the method is invoked, i.e. the famous **this** pointer in C++ (and Java).

Concretely, suppose a class **C** includes an instance member method **M()**, say with two arguments. Then the call

```
M(A, B)
```

from within **C** would in essence have three arguments—the two explicit ones, **A** and **B**, and also implicitly, a pointer to the current instance of **C**. As with any stack-based calling system, e.g. C++ running on an Intel machine, that extra parameter must be pushed onto the stack too, just like the explicit ones. This is also done in the case of the JVM.

Thus we can’t use **invokestatic** in the JVM case, because we must account for that “extra” argument. The **invokestatic** instruction, as explained earlier, pops the arguments off the caller’s stack and places them into the callee’s Local Variables Section. That’s not good enough in the case of an instance method, because that “extra” argument must be popped and placed into the callee’s Local Variables Section too. So, the designers of the JVM also included an **invokevirtual** instruction to do this; it transfers *all* the arguments from the caller’s stack to the callee’s Local Variables Section, both the explicit arguments and the “extra” one.

So, from the callee’s point of view, the item in its slot 0 will be **this**, the pointer to the current object, and in fact the compiler will translate references to **this** to accesses to slot 0.

The only example we have so far of a call made via **invokevirtual** is in the compiled code for our source line in **main()**,

```
System.out.println(Z);
```

The compiled code is

```
20 getstatic #4 <Field java.io.PrintStream out>
23 iload_3
24 invokevirtual #5 <Method void println(int)>
```

That 3 in offset 23 refers to slot 3, the location of our local variable **Z**, our argument to **println()**.

The code

```
System.out.println(Z);
```

calls the **println()** method on the object **System.out**. The official Java documentation indicates that the latter is an instance of the class **java.io.PrintStream**, which in turn is a subclass of the class **java.io.FilterOutputStream**. That latter class includes a member variable **out**, which is basically a pointer to the file we are writing to. In this case, we are writing to the screen (in UNIX parlance, **stdout**).

With that in mind, look at the instruction in offset 20. From our discussion above, you can see that this instruction must be placing a pointer to the object **System.out** on the stack, and that is exactly what the **getstatic** instruction is doing. It is getting a class (i.e. **static**) variable from the given object, and pushing it onto the stack.

## 5.9 Creating and Accessing Arrays

Arguments and local variables which are arrays are maintained as addresses within the Local Variables Area. Array read accesses work by pushing the address of the array and the desired index onto the operand stack, and then executing one of the special array instructions. Array writ accesses are done the same way, but with an additional push for the value to be stored.

There are also special array instructions which create the array storage itself when **new** is invoked.

Consider the following example:

```

1 public int gy(int x)
2 {   int y[],z;
3     y = new int[5];
4     z = x + 2;
5     y[3] = x;
6     return 0;
7 }
```

(It does nothing of interest, but will serve as a convenient simple example.)

That code is translated to:

```

Method int gy(int)
  0 iconst_5
  1 newarray int
  3 astore_2
  4 iload_1
  5 iconst_2
  6 iadd
  7 istore_3
  8 aload_2
  9 iconst_3
 10 iload_1
 11 iastore
 12 iconst_0
 13 ireturn
```

Here you see the **newarray** instruction is used to create space for the array, specifically enough for 5 **ints**; the 5 was pushed onto the operand stack by the **iconst\_5** (“integer constant”) instruction.

The assignment to **y[3]** is done by the code in offsets 7-11. Make sure you see how this works. In particular, the **iastore** (“integer array store”) instruction first pops the stack to determine which value to store (slot 1, i.e. **x**), then pops again to determine which array element to store to (index 3), and then pops once more to determine which array to use (slot 2, i.e. **y**).

In the code for **main()** in our **Min** example above, **CLArgs** is an array of strings, thus an array of arrays, just as in C/C++. The **aload\_0** (“address load”) instruction pushes the address in slot 0 i.e. the address of **CLArgs**, onto the current operand stack. Similarly, **iconst\_0** (“integer constant”) pushes the constant 0. All this is preparation for accessing the array element **CLArgs[0]**. However, since that element is a string, thus an address again, we do not use **iastore** as before, instead using the **aastore** instruction (“address array store”).

## 5.10 Constructors

When the constructor for a new object is called, we need to execute a **invokespecial** instruction.

## 5.11 Philosophy Behind the Design of the JVM

Remember, we are here to learn about the Java machine, not the Java language. One aspect of that concerns why the designers of the JVM made certain choices as to the structure of the machine.

Before we continue, note that the JVM *is* a machine, just like Intel, MIPS or the others. While it's true that we most often just use an emulator program, JVM chips have indeed been built. So for example one could certainly write a C compiler for the machine, i.e. write a compiler that translates C source code to JVM machine code.

### 5.11.1 Instruction Structure

Most instructions are a single byte in length, but there are a few multi-byte instructions as well. In any case, the first byte is the op code, and any operands are in the bytes that follow. As mentioned earlier, almost all JVM instructions involve stack operations, so there are no explicit operands, which is why most instructions are a single byte long. This has other implications, explained below.

### 5.11.2 Stack Architecture

You may wonder why the developers of the JVM chose the stack-based architecture. Actually, the literature is not clear about this.

One claim is that the choice of this architecture was made in order to make it easy to write JVM interpreters for native machines that had few or no registers. If for example the developers had chosen to have the JVM have 32 registers, then on machines with fewer registers the authors of Java interpreters would not be able to model the JVM virtual registers by the real registers of the native machines. Instead, the JVM registers would have to be modeled by variables in memory. This does not sound like a very convincing argument. For one thing, it makes it more difficult for authors of interpreters on machines that do have a lot of registers to write fast interpreters.

Another claim is that by having most instructions only a single byte in length, the overall program size is smaller. This would be an important consideration if it were true. Java code is often transported over the network. This happens behind the scenes when you access a Java-powered Web page. The Web site will download Java code for some operation to your machine (the one where you are running the browser), and that code will be executed by the Java interpreter on your machine.

So smaller code would mean less network download delay. But is Java code smaller? Recall that RISC instructions are also very simple, but that that actually tends to make code size larger than on CISC machines, since each instruction does less work. Since the stack architecture means a lot of pushes and pops in order to manoeuvre data items for arithmetic, it may well be the case that code size is larger for the JVM.

However,<sup>13</sup> the JVM situation is very different from the RISC one, because with the former we are worried about conserving network bandwidth. JVM byte code is likely to be highly compressible—certain instructions like, say, `iload_0` are probably quite frequent—so that download times could be made quite low.

---

<sup>13</sup>As pointed out by student Daniel Wolfe.

## 6 ANOTHER EXAMPLE

### 5.11.3 Safety

Note that there are separate instructions **istore.0** and **astore.0**. Each pops the operand stack and places the popped value into slot 0. But the first assumes the popped value is an integer, while the second assumes an address.<sup>14</sup> In other words, unlike “normal” machines, in which the hardware is unaware of types, the JVM is quite aware and proactive. The JVM will check types (which are stored with the value), and a runtime error will occur if, for example, one tries to execute **astore.0** when the top of the operand stack contains an integer type.

Java does allow **type coercion** as in C/C++. Consider for instance the code

```
int I = 12; float X;
...
X = I + (float) 3.8;
```

Here the variable **I** is converted to type **float** in the compiled code, using the JVM’s **i2f** instruction.

Getting back to the safety issue, one concern is that Java code might be “hacked” during its transport over the network. There are various mechanisms within the JVM to check that the code is the same as when it was sent; none of these is perfect, of course, but it does provide some protection.

## 6 Another Example

This one finds the row in a matrix which has minimum sum.

```
1 public class Min2 {
2
3     public static int ABC[][]; // elements will be within [0,1000000]
4
5     public static void main(String[] CLArgs)
6
7     { int I,J;
8
9         ABC = new int[10][10];
10        // fill it with something, just an example
11        for (I = 0; I < 10; I++)
12            for (J = 0; J < 10; J++)
13                ABC[I][J] = 2 * I + J;
14        System.out.println(Min());
15    }
16
17    public static int Min()
18
19    { int Row,Col,MinSoFar,Sum;
20
21        MinSoFar = 1000000; // note restrictions on ABC above
22        for (Row = 0; Row < 10; Row++) {
23            Sum = 0;
24            for (Col = 0; Col < 10; Col++) {
25                Sum += ABC[Row][Col];
26                if (Sum > MinSoFar) break;
27            }
28            if (Sum < MinSoFar)
29                MinSoFar = Sum;
```

---

<sup>14</sup>Formally called a **reference**.

## 6 ANOTHER EXAMPLE

```
30     }
31     return MinSoFar;
32 }
33 }
```

Compiled from Min2.java

```
public class Min2 extends java.lang.Object {
    public static int ABC[][];
    public Min2();
    public static void main(java.lang.String[]);
    public static int Min();
}
```

Method Min2()

```
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 return
```

Method void main(java.lang.String[])

```
0 bipush 10
2 bipush 10
4 multianewarray #2 dim #2 <Class [[I>
8 putstatic #3 <Field int ABC[][]>
11 iconst_0
12 istore_1
13 goto 45
16 iconst_0
17 istore_2
18 goto 36
21 getstatic #3 <Field int ABC[][]>
24 iload_1
25 aaload
26 iload_2
27 iconst_2
28 iload_1
29 imul
30 iload_2
31 iadd
32 iastore
33 iinc 2 1
36 iload_2
37 bipush 10
39 if_icmplt 21
42 iinc 1 1
45 iload_1
46 bipush 10
48 if_icmplt 16
51 getstatic #4 <Field java.io.PrintStream out>
54 invokestatic #5 <Method int Min()>
57 invokevirtual #6 <Method void println(int)>
60 return
```

Method int Min()

```
0 ldc #7 <Integer 1000000>
2 istore_2
3 iconst_0
4 istore_0
5 goto 52
8 iconst_0
9 istore_3
10 iconst_0
11 istore_1
12 goto 36
15 iload_3
```

## 6 ANOTHER EXAMPLE


```
16 getstatic #3 <Field int ABC[][]>
19 iload_0
20 aaload
21 iload_1
22 iaload
23 iadd
24 istore_3
25 iload_3
26 iload_2
27 if_icmple 33
30 goto 42
33 iinc 1 1
36 iload_1
37 bipush 10
39 if_icmplt 15
42 iload_3
43 iload_2
44 if_icmpge 49
47 iload_3
48 istore_2
49 iinc 0 1
52 iload_0
53 bipush 10
55 if_icmplt 8
58 iload_2
59 ireturn
```

So, for example, look at 13 of the source code,

```
1 ABC[I][J] = 2 * I + J;
```

This single Java statement compiles to a remarkable 10 JVM machine instructions! They are in offsets 21-32 of **main()**. Below is an overview of what they do:

```
21 push address of ABC
24 push I
25 pop I, address of ABC; push address of ABC[I]
26 push J
27 push 2
28 push I
29 pop I, 2; push 2*I
30 push J
31 pop J, 2*I; push 2*I+J
32 pop 2*I+J, J, address of ABC[I]; do ABC[I][J]=2*I+J
```

As you read this, recall that a two-dimensional array is considered an array of arrays. For example, row **I** of **ABC**, i.e. **ABC[I]**, is an array. Recall also that an array name, when used without a subscript, is the address of the beginning of that array. Here **ABC[I]**, considered as an array, has no subscript, while for instance **ABC[I][J]** has the subscript **J**. So, **ABC[I]** is an address. Thus **ABC** is an array of addresses! Hence the use of **aaload** in offset 25. 

The **multianewarray** instruction sets up space for the array, pushing the address on the stack. The **putstatic** instruction then pops that address off the stack, and places it in the entry for this **static** variable in the Method Area.

## 7 YET ANOTHER EXAMPLE

### 7 Yet Another Example

This example builds a linked list of integers:

```
1 // this class is in the file NumNode.java
2
3 public class NumNode
4
5 { private static NumNode Nodes = null;
6   // valued stored in this node
7   int Value;
8   // "pointer" to next item in list
9   NumNode Next;
10
11   public NumNode(int V) {
12     Value = V;
13     Next = null;
14   }
15
16   public static NumNode Head() {
17     return Nodes;
18   }
19
20   public void Insert() {
21     if (Nodes == null) {
22       Nodes = this;
23       return;
24     }
25     if (Value < Nodes.Value) {
26       Next = Nodes;
27       Nodes = this;
28       return;
29     }
30     else if (Nodes.Next == null) {
31       Nodes.Next = this;
32       return;
33     }
34     for (NumNode N = Nodes; N.Next != null; N = N.Next) {
35       if (Value < N.Next.Value) {
36         Next = N.Next;
37         N.Next = this;
38         return;
39       }
40       else if (N.Next.Next == null) {
41         N.Next.Next = this;
42         return;
43       }
44     }
45   }
46
47   public static void PrintList() {
48     if (Nodes == null) return;
49     for (NumNode N = Nodes; N != null; N = N.Next)
50       System.out.println(N.Value);
51   }
52
53 }
```

```
Compiled from "NumNode.java"
public class NumNode extends java.lang.Object{
int Value;
```

## 7 YET ANOTHER EXAMPLE

```
NumNode Next;

public NumNode(int);
Code:
  0:  aload_0
  1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
  4:  aload_0
  5:  iload_1
  6:  putfield      #2; //Field Value:I
  9:  aload_0
 10:  aconst_null
 11:  putfield      #3; //Field Next:LNumNode;
 14:  return

public static NumNode Head();
Code:
  0:  getstatic     #4; //Field Nodes:LNumNode;
  3:  areturn

public void Insert();
Code:
  0:  getstatic     #4; //Field Nodes:LNumNode;
  3:  ifnonnull    11
  6:  aload_0
  7:  putstatic    #4; //Field Nodes:LNumNode;
 10:  return
 11:  aload_0
 12:  getfield     #2; //Field Value:I
 15:  getstatic    #4; //Field Nodes:LNumNode;
 18:  getfield     #2; //Field Value:I
 21:  if_icmpge   36
 24:  aload_0
 25:  getstatic    #4; //Field Nodes:LNumNode;
 28:  putfield     #3; //Field Next:LNumNode;
 31:  aload_0
 32:  putstatic    #4; //Field Nodes:LNumNode;
 35:  return
 36:  getstatic    #4; //Field Nodes:LNumNode;
 39:  getfield     #3; //Field Next:LNumNode;
 42:  ifnonnull    53
 45:  getstatic    #4; //Field Nodes:LNumNode;
 48:  aload_0
 49:  putfield     #3; //Field Next:LNumNode;
 52:  return
 53:  getstatic    #4; //Field Nodes:LNumNode;
 56:  astore_1
 57:  aload_1
 58:  getfield     #3; //Field Next:LNumNode;
 61:  ifnull      119
 64:  aload_0
 65:  getfield     #2; //Field Value:I
 68:  aload_1
 69:  getfield     #3; //Field Next:LNumNode;
 72:  getfield     #2; //Field Value:I
 75:  if_icmpge   92
 78:  aload_0
 79:  aload_1
 80:  getfield     #3; //Field Next:LNumNode;
 83:  putfield     #3; //Field Next:LNumNode;
 86:  aload_1
 87:  aload_0
 88:  putfield     #3; //Field Next:LNumNode;
 91:  return
 92:  aload_1
```

## 7 YET ANOTHER EXAMPLE

```
93:  getfield      #3; //Field Next:LNumNode;
96:  getfield      #3; //Field Next:LNumNode;
99:  ifnonnull     111
102: aload_1
103:  getfield      #3; //Field Next:LNumNode;
106: aload_0
107:  putfield      #3; //Field Next:LNumNode;
110:  return
111:  aload_1
112:  getfield      #3; //Field Next:LNumNode;
115:  astore_1
116:  goto         57
119:  return

public static void PrintList();
Code:
 0:  getstatic     #4; //Field Nodes:LNumNode;
 3:  ifnonnull     7
 6:  return
 7:  getstatic     #4; //Field Nodes:LNumNode;
10:  astore_0
11:  aload_0
12:  ifnull       33
15:  getstatic     #5; //Field java/lang/System.out:Ljava/io/PrintStream;
18:  aload_0
19:  getfield      #2; //Field Value:I
22:  invokevirtual #6; //Method java/io/PrintStream.println:(I)V
25:  aload_0
26:  getfield      #3; //Field Next:LNumNode;
29:  astore_0
30:  goto         11
33:  return

static {};
Code:
 0:  aconst_null
 1:  putstatic     #4; //Field Nodes:LNumNode;
 4:  return
}
```

You may notice some new instructions in this example. There are actually only six of them—**aconst\_null**, **ifnull**, **ifnonnull**, **getfield**, **putfield**, and **areturn**.

The first three of these all deal with the value `NULL`, the value for null pointers. The instruction **aconst\_null** is very simple. All it does is push the value of `NULL` onto the stack, just like **iconst\_4** would push the value of 4 onto the stack.<sup>15</sup>

The two instructions **ifnull** and **ifnonnull** are conditional jumps, just like **if\_cmpeq**. However, instead of comparing the first item on the stack to the second one, they jump if the value on the top of the stack is a `NULL` value (for **ifnull**) or not `NULL` (for **ifnonnull**).

Let's look at one of these new conditional jumps in action. Here's the first line of the function **PrintList()**:

```
if (Nodes == null) return;
```

This line compiles to the following three instructions:

---

<sup>15</sup>Recall that the 'a' means "address," while 'i' means "integer."

## 7 YET ANOTHER EXAMPLE

```
0:  getstatic      #4; //Field Nodes:LNumNode;
3:  ifnonnull      7
6:  return
```

It's interesting to note that an “If Nodes **is** NULL...” source code line generates a “If Nodes is **not** NULL...” instruction (**ifnonnull**) in the compiled code! The reason? It takes fewer instructions to do it that way. Consider the alternative:

```
public static void PrintList();
Code:
0:  getstatic      #4; //Field Nodes:LNumNode;
3:  ifnull          9
6:  goto            10
9:  return
```

The instruction **areturn**<sup>16</sup> does basically the same thing as **ireturn**, except that it doesn't return an integer as a return value. Instead, it returns an object reference—a pointer to something other than your usual types like **int** and **float**.<sup>17</sup> For example, the statement

```
1 return Nodes;
```

in **Head()** compiles to

```
0:  getstatic      #4; //Field Nodes:LNumNode;
3:  areturn
```


The **getstatic** instruction pushes the value of **Nodes** onto the stack, and then the instruction **areturn** pops that value, and pushes it onto the stack of the function which called **Head()**.

The last two new instructions, **getfield** and **putfield** work basically the same as **getstatic** and **putstatic**, except that they load and save values to and from non-static member variables (“fields”) in a class. For instance, in the line

```
1 if (Value < Nodes.Value) {
```

the fetching of **Value**, whose “full name” is **this.Value**, compiles to

```
1 11: aload_0
2 12: getfield      #2; //Field Value:I
```

The **aload\_0** instruction pushes the address in Slot 0, which is **this**, onto the stack. The **getfield** instruction—yes, keep in mind that this is a JVM machine instruction—then gets the **Value** field from within the class instance pointed to by **this**. 

The file **Intro.java** illustrates the usage of the **NumNode** class:

---

<sup>16</sup>That's a *return*, not *are* turn.

<sup>17</sup>Again, the ‘a’ stands for “address.”

## 7 YET ANOTHER EXAMPLE

```
1 // usage: java Intro nums
2
3 // reads integers from the command line, storing them in a linear linked
4 // list, maintaining ascending order, and then prints out the final list
5 // to the screen
6
7 public class Intro
8
9 { public static void main(String[] Args) {
10     int NumElements = Args.length;
11     for (int I = 1; I <= NumElements; I++) {
12         int Num;
13         Num = Integer.parseInt(Args[I-1]);
14         NumNode NN = new NumNode(Num);
15         NN.Insert();
16     }
17     System.out.println("final sorted list:");
18     NumNode.PrintList();
19 }
20 }
```

### It compiles to

Compiled from Intro.java

```
public class Intro extends java.lang.Object {
    public Intro();
    public static void main(java.lang.String[]);
}
```

Method Intro()

```
0 aload_0
1 invokespecial #1 <Method java.lang.Object()>
4 return
```

Method void main(java.lang.String[])

```
0 aload_0
1 arraylength
2 istore_1
3 iconst_1
4 istore_2
5 goto 35
8 aload_0
9 iload_2
10 iconst_1
11 isub
12 aaload
13 invokestatic #2 <Method int parseInt(java.lang.String)>
16 istore_3
17 new #3 <Class NumNode>
20 dup
21 iload_3
22 invokespecial #4 <Method NumNode(int)>
25 astore 4
27 aload 4
29 invokevirtual #5 <Method void Insert()>
32 iinc 2 1
35 iload_2
36 iload_1
37 if_icmple 8
40 getstatic #6 <Field java.io.PrintStream out>
43 ldc #7 <String "final sorted list:">
45 invokevirtual #8 <Method void println(java.lang.String)>
48 invokestatic #9 <Method void PrintList()>
51 return
```

## A OVERVIEW OF JVM INSTRUCTIONS

There are again some new instructions to discuss here. First, consider the Java statement

```
1 int NumElements = Args.length;
```

in **Intro.java**. Java is a more purely object-oriented language than C++, and one illustration of that is that in Java arrays are objects. One of the member variables in the **Array** class is **length**, the number of elements in the array. Thus the compiler translates the fetch of **Args.length** to

```
1 0 aload_0
2 1 arraylength
```

Note again that **arraylength** is a JVM machine instruction. This is not a subroutine call.

Now consider the statement

```
1 NumNode NN = new NumNode(Num);
```

It compiles to

```
17 new #3 <Class NumNode>
20 dup
21 iload_3
22 invokespecial #4 <Method NumNode(int)>
25 astore 4
```

The JVM instruction set includes an instruction **new**, which allocates space from the heap for the object to be created of class **NumNode**. The instruction pushes a pointer to that space. Next, the **dup** (“duplicate”) instruction pushes a second copy of that pointer, to be used later in the compiled code for

```
1 NN.Insert();
```

But before then we still need to call the constructor for the **NumNode** class, which is done in offset 22, after pushing the parameter in offset 21. The assignment of the pointer to **NN** then is done in offset 25.

## A Overview of JVM Instructions

In the following, *top* will refer to the element at the top of the operand stack, and *nexttop* will refer to the element next to it.

- **aaload:**

Format: 0x32

Loads an element of an array of addresses (i.e. from a multidimensional array). Treats *nexttop* as a pointer to an array (i.e. a variable declared of array type), and *top* is an index into the array. The instruction loads the array element and pushes it onto the operand stack. In other words, *nexttop* and *top* are popped, and *nexttop*[*top*] is fetched and pushed onto the operand stack.

## A OVERVIEW OF JVM INSTRUCTIONS

- **arraylength:**

Format: 0xbe

Pops the operand stack to get the array address, and then pushes the length of the array.

- **aastore:**

Format: 0x53

Does the opposite of **aload**, popping the operand stack first to get *value*, then to get *index*, then finally to get *address*. It then stores *value* into element *index* of the array starting at *address*.

- **aconst\_null:**

Format: 0x01

Pushes the value `NULL` onto the stack.

- **aload:**

Format: 0x19 *8bitindex*

Treats the value in slot *8bitindex* as an address, and pushes it onto the stack.

- **aload\_0, aload\_1, aload\_2, aload\_3:**

Formats: 0x2a, 0x2b, 0x2c

The instruction **aload\_0**, is the same as **aload**, but specifically for slot 0. The others are analogous.

- **areturn:**

Format: 0xb0

Does exactly the same thing as **ireturn**, but instead of returning an integer, it returns an object reference.

- **astore:**

Format: 0x3a, *8bitindex*

Opposite of **aload**, popping the operand stack and placing the popped value (presumed to be an address) into the specified slot.

- **astore\_0, astore\_1, astore\_2, astore\_3:**

Formats: 0x4b, 0x4c, 0x4d, 0x4e

Same as **astore**, but specifically for slot 0, slot 1 etc..

- **bipush:**

Format: 0x10 *8bitinteger*

Pushes the given 8-bit integer onto the operand stack.

- **dup:**

Format: 0x59

Duplicates the top word on the operand stack, so the operand stack now has two copies of that word instead of one.

## A OVERVIEW OF JVM INSTRUCTIONS

- **getfield:**

Format: 0xb5 *16bitindex*

Opposite of **putfield**. Gets value of the given non-static item, then pushes it onto the operand stack.

- **getstatic:**

Format: 0xb2 *16bitindex*

Opposite of **putstatic**. Gets value of the given static item, then pushes it on the operand stack.

- **goto:**

Format: 0xa7 *16bitjumpdistance*

Unconditional jump. See **if\_icmpeq** for explanation of *16bitjumpdistance*.

- **i2f:**

Format: 0x86

Pops the top word on the stack, converts it from **int** to **float**, and pushes the new value back onto the stack.

- **iadd:**

Format: 0x60

Pops *nexttop* and *top*, and pushes the sum  $nexttop + top$ .

- **iaload:**

Format: 0x2e

Load an element of an integer array. See **aaload** above.

- **iastore:**

Format: 0x4f

Store to an element of an integer array. See **aastore** above.

- **iconst\_0, iconst\_1, iconst\_2, iconst\_3, iconst\_4, iconst\_5:**

Format: 0x3, 0x4, 0x5, 0x6, 0x7, 0x8

Pushes the integer constant 0, 1, 2 etc. onto the operand stack.

- **idiv:**

Format: 0x6c

Pops *nexttop* and *top*, and pushes the quotient  $nexttop / top$ .

- **if\_icmpeq:**

Format: 0x9f *16bitjumpdistance*

If  $top = nexttop$ , jumps the given distance to the branch target. The quantity *16bitjumpdistance* is a 2-byte, 2s complement signed number, measured from the jump instruction. Both *top* and *nexttop* must be integers; a runtime error occurs if no.

## A OVERVIEW OF JVM INSTRUCTIONS

- **if\_icmpge:**  
Format: 0xa2 16bitjumpdistance  
Same as **if\_icmpeq**, but jumps if  $nexttop \geq top$ .
- **if\_icmple:**  
Format: 0xa4 16bitjumpdistance  
Same as **if\_icmpeq**, but jumps if  $nexttop \leq top$ .
- **if\_icmplt:**  
Format: 0xa1 16bitjumpdistance  
Same as **if\_icmpeq**, but jumps if  $nexttop < top$ .
- **if\_icmpne:**  
Format: 0xa0 16bitjumpdistance  
Same as **if\_icmpeq**, but jumps if  $top \neq nexttop$ .
- **ifnonnull:**  
Format: 0xc7 16bitjumpdistance  
Same as the previous jump conditions, but jumps if  $top \neq \text{NULL}$ .
- **ifnull:**  
Format: 0xc6 16bitjumpdistance  
Same as the previous jump conditions, but jumps if  $top = \text{NULL}$ .
- **iinc:**  
Format: 0x84 8bitindex 8bitinteger  
Increments slot *8bitindex* by the amount *8bitinteger*.
- **iload:**  
Format: 0x15 8bitindex  
Same as **aload** but for integers instead of addresses.
- **iload\_0, iload\_1, iload\_2, iload\_3:**  
Formats: 0x1a, 0x1b, 0x1c, 0x1d  
Same as **aload\_0** etc. but for integers instead of addresses.
- **imul:**  
Format: 0x68  
Pops *nexttop* and *top*, and pushes the product  $nexttop \times top$ .
- **invokespecial:**  
Format: 0xb7 16bitindex  
Like **invokevirtual**, but for constructors, superclass and other special situations.

## A OVERVIEW OF JVM INSTRUCTIONS

- **invokestatic:**

Format: 0xb8 *16bitindex*

Method call. The quantity *16bitindex* serves as an index into the Constant Pool, pointing to the given method. Creates a new stack frame for the method. Pops the method's arguments from the caller's operand stack and places them into the method's Local Variables Section. Points the **frame** register to the method's stack frame, and jumps to the method.

- **invokevirtual:**

Format: 0xb6 *16bitindex*

Same as **invokestatic**, but the arguments include the "hidden" argument **this**, i.e. a pointer to the object this method is being invoked on.

- **ireturn:**

Format: 0xac

Return from method with return value. Pops integer from current operand stack, places it on the caller's operand stack, restores the **frame** register to point to the caller's stack frame, and jumps back to the caller.

- **istore:**

Format: 0x36 *8bitindex*

Same as **astore** but for integers instead of addresses.

- **istore\_0, istore\_1, istore\_2, istore\_3:**

Formats: 0x3b, 0x3c, 0x3d, 0x3e

Same as **astore\_0** etc. but for integers instead of addresses.

- **isub:**

Format: 0x64

Pops *nexttop* and *top*, and pushes the difference *nexttop - top*.

- **ldc:**

Format: 0x12 *8bitindex*

Gets an item from the Constant Pool and pushes it onto the operand stack.

- **new:**

Format: 0xbb *16bitindex*

Performs the Java **new** operation, with *16bitindex* being the index into the Constant Pool, pointing to the given class. Creates the given object using memory from the Heap, and then pushes the address of the new object on the operand stack.

- **putfield:**

Format: 0xb4 *16bitindex*

Pops the operand stack, and assigns the popped value to the non-static item given by *16bitindex*.

## B REFERENCES

- **putstatic:**  
Format: 0xb3 *16bitindex*  
Pops the operand stack, and assigns the popped value to the static item given by *16bitindex*.
- **return:**  
Format: 0xb1  
Same as **ireturn**, but for methods without a return value.
- **swap:**  
Format: 0x5f  
Swaps the top two words of the operand stack.

## B References

- Bill Venners' book, *Inside the Java Virtual Machine*. Available on the Web (with many helpful URL links) at [www.artima.com](http://www.artima.com).
- Sun's "official" definition of the JVM: *The Java Virtual Machine Specification*, by Lindholm and Yellin, also available on the Web, at <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>. Chapter 6 gives specs (mnemonics, op codes, actions, etc.) on the entire JVM instruction set.