```
   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.

Let’s first look at the Local Variables section of Main()’s stack frame:

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

Now consider the call to Min. The code

```
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 will be pushed onto Main()'s Operand Stack, as follows. The **iload_1** ("integer load") instruction pushes slot 1 to the operand stack. Since slot 1 in main() contains X, this means that X will be pushed onto the operand stack. The instruction in offset 15 will then push Y.

If Min() had been an instance function, i.e. not declared **static**, the first argument pushed would have been **this**, a pointer to the object on which Min() is being invoked.

The call itself is then done by the instruction **invokestatic** in offset 16. (For a nonstatic method call, we would use **invokevirtual**.) This instruction is three bytes in length, as can be seen by the fact that the following instruction begins at offset 19. The instruction's two-byte operand, in this case 3, serves as a pointer to an entry corresponding to Min() in the Constant Pool of the Method Area. In this way, the JVM will know where the first instruction of Min() is located, and the **pc** will be set accordingly, causing Min() to begin execution.

The actions of **invokestatic** is to pop the arguments off the caller's (in this case, Main()'s) Operand Stack, [6] and place them in the Local Variables section of the callee's (in this case Min()'s) Stack.

The **istore_3** instruction following the call, in offset 19, pops the top of Main()'s Operand Stack and places it into slot 3, in our case Z. [7]

The bytecode in Min() is similar. The main new instruction here 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_1** and **iload_2** instructions in Min(), which took them from Min()'s Local Variables area, and they in turn had been placed there by Main() when executing **invokestatic**.

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, our target is offset 10 (an **iload_1** instruction). Since our **if_icmpge** instruction is in offset 2, the distance will be 8, i.e. 0x0008. Those latter two bytes comprise the second and third bytes of the instruction.

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.

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 0xa20008, and this string of three bytes should appear in the file Minimum.class. Running the command

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

[6] Note that this means that JVM does its own stack cleanup.

[7] Explained below, the value now popped had been placed there by the **ireturn** instruction in Min().