

# CS1Bh Lecture Note 7

## Compilation I: Java Byte Code

High-level programming languages are compiled to equivalent low-level programs which are executed on a given machine. The process of compiling a program has many stages. It first analyses the input program to look for errors of various kinds. If none of these are present then the program will be accepted for translation. The translation process itself has different stages, making many different transformations on the form of the program in order to move it towards the equivalent low-level program which the compiler will write as its output.

In this lecture note we first discuss the various forms of the compilation process for the Java language. We then have a brief look at Java Byte Code, the machine language for the Java Virtual Machine.

### 7.1 Java compilation

The purpose of compilation is to produce an *executable* version of a program. There exist three different approaches to Java compilation. The first is compilation to Java byte code. The second is a two-stage process going through byte code to native code. The third bypasses Java byte code altogether, going directly to native machine code.

#### 7.1.1 Byte-code compilation

The standard approach to Java compilation is to compile to Java byte code and then to execute this on a Java Virtual Machine (JVM). The JVM *interprets* the Java byte codes to run the program. Compilation to byte code brings several genuine advantages. One advantage is that compiled programs are *portable* in the sense that they can be run on a number of different computer systems (such as Linux, Windows, Solaris, and many others) for which a JVM is available. The Java language is the only programming language in widespread use today which provides this capability. At best, programs written in other programming languages need to be recompiled if they are moved from one machine to another. More often, they need to be modified as well. The portability which Java provides is brought about because only the JVM needs to be ported from one machine to another in order to be able to port any Java byte code program between machines.

Another advantage which Java byte code brings is that compiled Java programs are

relatively small when compared to the compiled program representations of programs which do not use an intermediate byte code format. This is because many of the library methods which need to be included in the executable form of non-Java programs do not need to be included in Java byte code. They are provided only once in the Java Virtual Machine itself, not once in every compiled program. Thus the size of a Java application executable might more typically be measured in kilobytes whereas the size of a non-Java application executable is often measured in megabytes. The size of an executable file is an issue for the modern, networked applications of today. If part or all of the program's code will need to be downloaded over a computer network before it is executed then keeping the sizes of compiled programs small will provide a significant benefit. Communication over information over a network is frequently the bottleneck in a computer system, taking much longer to perform than disk input/output or memory loads and stores.

Against its benefits, the use of byte code brings a disadvantage. It is often argued that interpretation of byte code programs is much slower than execution of *native* code, compiled only for the machine which we are actually using. Users of computer programs want efficient products: it is frustrating to use a computer program which pauses during execution or which cannot keep up with the speed of user input. The approach used to combine the usefulness of byte code with the efficiency of native code is called *just-in-time* compilation.

### 7.1.2 Just-in-time compilation

A just-in-time compiler (a JIT) compiles Java byte code to the native machine code of the machine which we are using (for example, an Intel Pentium processor or a SUN Sparc chip). This compilation is performed at the time when the request is made to execute the program (hence the name "just in time"). Java virtual machines include a JIT compiler which first compiles the Java byte codes to native code and then runs the native code.

It would seem that this method of compilation combines the benefits of the byte code approach with the speed of the native code approach but still it has a drawback. A JIT runs in "user time", which means that the time taken to compile the byte codes to native code is seen by the user as a delay in starting the execution of the program. This visibility to the user severely constrains the amount of time which the JIT compiler can take to run. However, the work of producing very efficient native code requires a number of different time-consuming analyses. A JIT cannot take the time to perform these so it must settle for producing compiled native code whose run-time performance could have been improved, had more time been available. Because the time to compile the program from byte codes is perceived as part of the run time of the application and because the compiled native code cannot be highly optimised, it might be that the final run time of the application is not much better than if the program had been interpreted as byte codes in the first place, saving the overhead of the time taken by JIT compilation. For this reason many Java virtual machines (such as SUN's *java* and Microsoft's *jaview*) provide JIT compilation as an option, which can be turned off.

### 7.1.3 Native compilation

If we are willing to give up the portability and compact size of executable program provided by Java byte code then an option is to compile our Java source code directly to native code for our chosen machine. In this case the time taken by the compilation is performed only while the program is being developed and not every time that the program is executed. In this case it is possible to spend more time running the compiler, performing the time-consuming analyses which may lead to a more highly optimised native code program as a result.

Such a compiler for Java is *gcj*, the GNU compiler for Java. This is part of a family of compiler programs which includes compilers for the languages C and C++. Some parts of the code of these compilers are shared and development tools such as the program debugger can be used on programs written in these languages and compiled with one of the GNU family of compilers.

## 7.2 Java Byte Code

A Java program stored in a `.java` file is compiled to produce a `.class` file which is interpreted by the Java interpreter (its *virtual machine*). We now examine the contents of a compiled `.class` file to find what it contains. The answer is that it contains another program, equivalent to ours, expressed in a simpler programming language. This simpler programming language is called *Java byte code*. It is not usual for programmers to program in Java byte code directly, rather they program in Java and then compile their programs into Java byte code.

In order to inspect the output from the Java compiler, we use a program called a *disassembler*. This converts the Java byte code program from a form suitable for efficient interpretation into a textual form suitable for us to read and study. Given a compiled Java class in the file `Loops.class` the Java disassembler is invoked with a `javap` command such as the following.

```
javap -c Loops
```

(The `-c` option disassembles the code.)

### 7.2.1 Loops in Java byte code

Consider the following Java methods, each of which loops ninety-nine times without doing anything.

```
void for99() {
    for (int i = 0 ; i < 99 ; i++){
        ;
    }
}
```

```
void while99() {
    int i = 0;
    while (i < 99) {
        i++;
    }
}
```

In general we think of a **while** loop and a **for** loop as being equivalent in that any iteration which can be coded using one of the loop constructs can also be coded using the other. In the case of the particular two loops above we consider them to be equivalent in that both initialise the loop control variable to zero and then go up in steps of one until the variable reaches ninety-nine.

The Java byte code language has neither a **for** loop nor a **while** loop. It encodes iteration using conditional and unconditional jumps (by “jump” we mean the infamous “goto” statement which Dijkstra so disliked). The two forms of loops are equivalent in another sense in that after compilation they produce identical sequences of byte code instructions. Below we show the two byte code programs which are produced from these methods, inspected by the disassembler. (In an attempt to avoid confusion between the two languages, we use different type faces for them. We use typewriter font for Java, as always, and *italic* for Java byte code.)

<b>Method</b> void for99()	<b>Method</b> void while99()
0 <i>iconst_0</i>	0 <i>iconst_0</i>
1 <i>istore_1</i>	1 <i>istore_1</i>
2 <b>goto</b> 8	2 <b>goto</b> 8
5 <i>iinc</i> 1 1	5 <i>iinc</i> 1 1
8 <i>iload_1</i>	8 <i>iload_1</i>
9 <i>bipush</i> 99	9 <i>bipush</i> 99
11 <b>if_icmplt</b> 5	11 <b>if_icmplt</b> 5
14 <b>return</b>	14 <b>return</b>

These byte code instruction sequences manipulate a stack of operands and the memory where the values of variables are stored. The instructions *iconst*, *iload* and *bipush* push operands on top of the operand stack. The *istore* instruction and the *iinc* instruction update the memory. The instructions **goto** and **if\_icmplt** (if integer compare less than) cause transfers of control to the numbered line. There are six integer comparison instructions in Java byte code, **if\_icmpeq**, **if\_icmpne**, **if\_icmplt**, **if\_icmple**, **if\_icmpgt** and **if\_icmpge** (for equal, not equal, less than, less than or equal to, greater than, and greater than or equal to). The **return** instruction is just like its Java counterpart. We now trace through the bytecode program, line by line.

- Line 0 The integer constant zero is pushed on top of the stack.
- Line 1 The top of the stack is stored into variable number one (the variable *i*).
- Line 2 Jump to line 8, avoiding incrementing *i* before the first comparison.
- Line 5 Increment local variable one by 1 (*i*++).
- Line 8 Read the current value of *i* and push it on top of the stack.
- Line 9 Push the integer constant 99 on top of the stack.
- Line 11 Compare the top two items on the stack and jump if need be (*i* < 99).
- Line 14 Return `void` when the end of the method is reached.

## 7.2.2 Conditionals in Java byte code

Of course different programs, even if they achieve the same effect, will usually give rise to different byte code sequences when compiled. We consider now two different ways of implementing a method to compute the *absolute value* of an integer (the absolute

value of an negative integer  $-n$  is  $n$  whereas the absolute value of a positive integer  $n$  is  $n$  itself.

We write two versions of a method to compute the absolute value of  $n$ . These versions are called `absFirst()` and `absSecond()`. The difference between them is whether we test for being negative or test for being positive. In the first case we place the value  $-n$  on the true limb of the conditional and the value  $n$  on the false limb. In the second case we instead place  $n$  on the true limb of the conditional and  $-n$  on the false limb of the conditional.

<pre>int absFirst(int n) {     if (n &lt; 0)         return -n;     else         return n; }</pre>	<pre>int absSecond(int n) {     if (n &gt; 0)         return n;     else         return -n; }</pre>
--	---

These give rise to the following byte code sequences when compiled.

<b>Method</b> <i>int absFirst(int)</i>	<b>Method</b> <i>int absSecond(int)</i>
0 <i>iload_1</i>	0 <i>iload_1</i>
1 <b>ifge</b> 7	1 <b>ifle</b> 6
4 <i>iload_1</i>	4 <i>iload_1</i>
5 <i>ineg</i>	5 <b>ireturn</b>
6 <b>ireturn</b>	6 <i>iload_1</i>
7 <i>iload_1</i>	7 <i>ineg</i>
8 <b>ireturn</b>	8 <b>ireturn</b>

The case of comparing with zero occurs so commonly in programs that specialised versions of the comparison operators are provided for this, **ifeq**, **ifne**, **iflt**, **ifge**, **ifgt**, **ifle**. The instruction *ineg* negates the integer on the top of the stack. The instruction **ireturn** returns the integer result on top of the stack.

### 7.2.3 Side-effecting expressions

The expressions `++x` and `x++` both add one to `x` and return an integer result. They differ in that the expression `++x` returns the value of `x` *after* adding one whereas `x++` returns the value of `x` *before* adding one. The following methods differ only in this way.

<pre>int plusPlusX(int x) {     return ++x; }</pre>	<pre>int xPlusPlus(int x) {     return x++; }</pre>
---	---

In the method `plusPlusX()`, `x` is first incremented (line 0) and its new value is then loaded onto the operand stack (line 3). The integer which remains on the top of the stack when the method returns is the new value of `x`. In the method `xPlusPlus()`, `x` is first loaded onto the operand stack (line 0) and its old value remains there while it is incremented (line 4). The integer which remains on the top of the stack when the method returns is the old value of `x`.

<b>Method</b> <i>int plusPlusX(int)</i>	<b>Method</b> <i>int xPlusPlus(int)</i>
0 <i>iinc 1 1</i>	0 <i>iload_1</i>
3 <i>iload_1</i>	1 <i>iinc 1 1</i>
4 <b><i>ireturn</i></b>	4 <b><i>ireturn</i></b>

## 7.2.4 Break and continue

The control operators to break out of a loop or to continue with the next iteration have simple translations in Java byte code instructions. Each causes a transfer of control, the **break** to the next statement after the loop and the **continue** to the update operation which precedes the loop condition evaluation. The following example illustrates this process.

```
void breakContinue() {
    for (int i = 0 ; i < 99 ; i++) {
        if (i < 90) continue;
        else break;
    }
}
```

```
Method void breakContinue()
0 iconst_0
1 istore_1
2 goto 17
5 iload_1
6 bipush 90
8 if_icmpge 23
11 goto 14
14 iinc 1 1
17 iload_1
18 bipush 99
20 if_icmplt 5
23 return
```

**Exercise:** Compile the following Java program and then disassemble it and see if you can understand the workings of the bytecodes for the methods `strict()` and `shortCircuit()`.

```
class BooleanExpressions {
    boolean strict(int x, int y) {
        return (x == 0) & (y == 0);
    }
    boolean shortCircuit(int x, int y) {
        return (x == 0) && (y == 0);
    }
}
```

## 7.2.5 Simple method invocation

The simplest type of method to invoke in Java is a static method with no parameters. Below we show the Java source code for a class with three static methods and the relevant part of the compiled byte code for this class. The methods of the class are referred to by number so that method `first()` is #1, method `second()` is #2 and

method `third()` is #3. Returning an integer result from an integer method is achieved by leaving the integer result on top of the operand stack.

```

class Methods {
    static int first() {
        return second();
    }
    static int second() {
        return third();
    }
    static int third() {
        return 3;
    }
}

```

```

class Methods
Method int first()
    0 invokestatic #2
    3 ireturn

Method int second()
    0 invokestatic #3
    3 ireturn

Method int third()
    0 iconst_3
    1 ireturn

```

The operation of the method `third()` is the easiest to understand here. It simply places the integer constant 3 on top of the stack (using the ***iconst*** instruction) and returns it as its integer result (using ***ireturn***). The `first()` method simply invokes the `second()` method (using the ***invokestatic*** instruction) and then returns the result of that. The method `second()` is similar.

## 7.2.6 Invoking methods with parameters

More often, we use methods when we have some data from which we want to compute a result. We invoke the method and pass the data as a parameter to the method. This means that in the byte code we first see the parameter to the method being evaluated (Java calls *by value*) and then the method is invoked. In method `add2()` below the expression `i+1` is first evaluated and then the method `add1()` is invoked.

```

class Parameters {
    static int seven() {
        return add2(5);
    }
    static int add2(int i) {
        return add1(i + 1);
    }
    static int add1(int i) {
        return i + 1;
    }
}

```

```

class Parameters
Method int seven()
    0 iconst_5
    1 invokestatic #2
    4 ireturn

Method int add2(int)
    0 iload_0
    1 iconst_1
    2 iadd
    3 invokestatic #3
    6 ireturn

Method int add1(int)
    0 iload_0
    1 iconst_1
    2 iadd
    3 ireturn

```

Seen from inside the method, formal parameters are simply numbered, just as local variables are. Thus the methods `add2()` and `add1()` refer to the integer variable numbered zero (using ***iload\_0***).

## 7.2.7 Invoking non-static methods

Something very significant is missing from our view of Java Byte Code to this point, namely objects. Objects are manipulated in the Java Virtual Machine as *addresses*, thus instead of **iload** we find **aload** and similar instructions. If we do not mark the method `same()` below as being static then it is a non-static (or *virtual*) method which can refer to the object with which it is associated using **this**.

```

class Virtual {
    int same(Object o) {
        if (this.equals(o))
            return 1;
        else
            return 0;
    }
}

```

**Method** `int same(java.lang.Object)`

```

0 aload_0
1 aload_1
2 invokevirtual #4
5 ifeq 10
8 iconst_1
9 ireturn
10 iconst_0
11 ireturn

```

This method loads two addresses onto the operand stack, address zero corresponding to **this** and address one corresponding to the formal parameter, `o`. It invokes the `equals()` method on these objects and tests the result. It returns 1 if the objects are equal and 0 if they are not.

## References

The Java Virtual Machine is described in the book *The Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin, Addison-Wesley, Second edition, 1999.

Stephen Gilmore.  
Javier Esparza, February 6, 2003.