



Compilation I: Java Byte Code and the Java Virtual Machine

Javier Esparza (Stephen Gilmore)
School of Informatics
The University of Edinburgh

28th October 2003

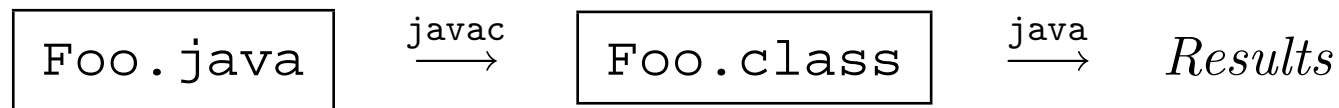
Compilation

- *High-level* programming languages are *compiled* into *low-level* languages which can be *executed* by a given machine
- Traditional compilation: the low-level language is the *native language* of the microprocessor
 - Fast execution, but code non portable, and large compiled files
 - GNU compiler *gcj*
- Modern approach: the low-level language is the language of a *virtual machine*.

- Translation + simulation of the virtual machine:
 - Java programs are translated into *Java Byte Code*, which is *interpreted* by the *Java Virtual Machine* (JVM), which is *simulated* by each real machine
 - Portable and small compiled files (`class` files), slow execution.
 - *javac* compiler + *java* simulator
- Just-in-time compilation:
 - Translation into bytecode + ‘just-in-time’ translation into native code
 - portable, small files, but native code is less optimized, since it has to be compiled in *user time*.
 - Options in *java* or Microsoft’s *jview*

Compiling Java programs

- A Java program is compiled with the `javac` command, and produces a *class* file which can be executed with the `java` command.

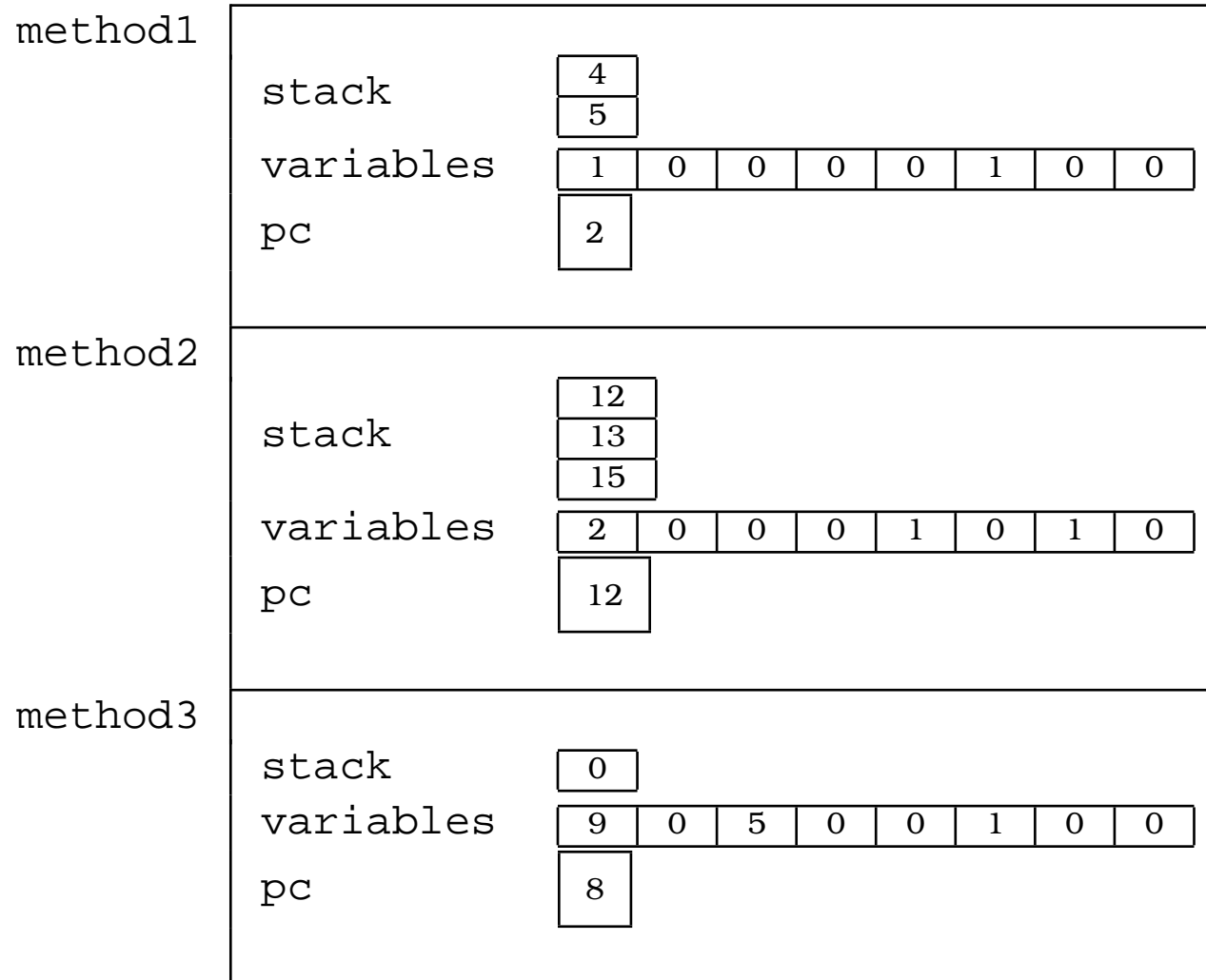


- We can look at our Java source code with an editor, or by printing it on the printer but to look at the compiled class file, we use a *disassembler*.



The Java Virtual Machine (simplified)

- The JVM manipulates a stack of *frames*
 - A new frame is pushed on top of the stack when a method is invoked
 - A frame is popped when the method returns
- A *frame* contains the current state of a method. It consists of:
 - a collection of *variables* 1,2,3,
 - a *program counter* (a special variable), and
 - a *stack* of values, used to perform operations



Java bytecode programs

- A bytecode program is a list of *instructions*, each one composed by a *label*, an *instruction name*, and 0, 1, or 2 *operands*. In (a simplified view of) an *execution cycle* the virtual machine
 - reads the program counter, which contains the label of the next instruction to be executed, and
 - executes the instruction, which may involve popping/pushing operands from the stack, reading/updating a variable, and updating the program counter.
- Example: the JVM instruction `iload 23` reads the current value of variable 23, pushes it on top of the stack, and increases the program counter by 1.

For loops in Java byte code

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

Method void for99()

0 iconst_0

1 istore_1

2 **goto** 8

5 iinc 1 1

8 iload_1

9 bipush 99

11 **if_icmplt** 5

14 **return**

Method *void for990*

0	<code>iconst_0</code>	push 0 on top of the stack
1	<code>istore_1</code>	pop the stack, and store the result in variable 1
2	<code>goto 8</code>	go to the instruction with label 8
5	<code>iinc 1 1</code>	add 1 to variable 1
8	<code>iload_1</code>	push the value of variable 1 on top of the stack
9	<code>bipush 99</code>	push 99 on top of the stack
11	<code>if_icmplt 5</code>	compare the top two items of the stack if second < first go to the instruction with label 5
14	<code>return</code>	return void

While loops in Java byte code

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

```
Method void while99()  
0  iconst_0  
1  istore_1  
2  goto 8  
5  iinc 1 1  
8  iload_1  
9  bipush 99  
11 if_icmplt 5  
14 return
```

Conditionals in Java byte code

```
int absFirst(int n) {  
    if (n < 0)  
        return -n;  
    else  
        return n;  
}
```

Method *int* absFirst(*int*)

0 **iload_1**

1 **ifge** 7

4 **iload_1**

5 **ineg**

6 **ireturn**

7 **iload_1**

8 **ireturn**

Side-effecting expressions

```
int plusPlusX(int x) {
    return ++x;
}
```

```
int xPlusPlus(int x) {
    return x++;
}
```

Method int plusPlusX(int)

0 iinc 1 1

3 iload_1

4 **ireturn**

Method int xPlusPlus(int)

0 iload_1

1 iinc 1 1

4 **ireturn**

Break and continue

```
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
```

An example

```
int method1(int n) {  
    for (int i = 0 ; i < 15 ; i++) {  
        for (int j = 0 ; j < 9 ; j++) {  
            n += 4;  
        }  
    }  
    return n;  
}
```

```
Method int method1(int)
  0  iconst_0          //
  1  istore_2         // int i = 0
  2  goto 25         //
  5  iconst_0         //
  6  istore_3         // int j = 0
  7  goto 16         //
 10  iinc 1 4         // n += 4;
 13  iinc 3 1         // j++;
 16  iload_2         //
 17  bipush 9         //
 19  if_icmplt 10     // j < 9
 22  iinc 2 1         // i++;
 25  iload_2         //
 26  bipush 15        //
 28  if_icmplt 5     // i < 15
 31  iload_1         //
 32  ireturn        // return n;
```

Simple method invocation

- The simplest type of method to invoke in Java is a static method with no parameters.
- A new frame is pushed on top of the stack.
- 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

Invoking methods with parameters

- We invoke the method and pass the data as a parameter to the method.
- 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.
- 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***).

```
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;
    }
}
```

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*

Invoking non-static methods

- 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`**.
- 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.

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

Method *int same(Object)*

0 *aload_0*

1 *aload_1*

2 *invokevirtual #4*

5 *ifeq 10*

8 *iconst_1*

9 *ireturn*

10 *iconst_0*

11 *ireturn*