Compiling Techniques
Lecture 10: Introduction to Java ByteCode

Christophe Dubach

10 November 2015
Coursework: Block and Procedure
Table of contents

1 Introduction
   - Overview
   - Java Virtual Machine
   - Frames and Function Call

2 Java ByteCode
   - JVM Types and Mnemonics
   - Arguments and Operands

3 Details
   - Variables
   - Function Call and Return value
   - Control Flow
From Java source code to the JVM

Java code (.java) → Java compiler ($javac) → Java ByteCode (.class) → Class Loader → Java Virtual Machine ($java)

Class Loader → Interpreter, JIT compiler → Execution Engine
From SmallC source code to the JVM

SmallC code (.c)

Java code (.java)

SmallC compiler

Java compiler ($javac)

Java ByteCode (.class)

Class Loader

Interpreter

JIT compiler

Execution Engine

Java Virtual Machine ($java)
Java (or SmallC) compiler: compile the source code into Java ByteCode

Classloader: loads the bytecode from class files into the Runtime

Execution Engine: executes the ByteCode
  - Interpreter: interprets the ByteCode
  - JIT compiler: compiles the ByteCode into native instruction on the fly (JIT = Just-In-Time)
The Java Virtual Machine is an abstract computing machine. It has an instruction set and manipulates various memory areas at run time. The Java Virtual Machine knows nothing of the Java programming language, only of a particular binary format, the class file format. A class file contains Java Virtual Machine instructions (or Java ByteCode) as well as other information.
In the JVM, a frame:

- stores data and partial results;
- performs dynamic linking;
- returns values for methods;
- and dispatches exceptions.
- **Local Variables**: array of variables declared locally in a method (includes parameters)
- **Operand Stack**: LIFO (Last-In Last-Out) stack of operands

Java Stack ≠ Operand Stack

The operand stack is used to perform computation. The Java stack is used to keep track of function calls.
Frame for foo

Local Variables

| a | b |

Operand Stack

| 13 |
|    |
|  7 |

New frame for bar

Local Variables

| x=7 | y=13 |

Operand Stack

Example

```java
void bar(int x, int y) {
 ...
}

void foo(int a) {
    int b;
    bar(7, 13);
    // push 7
    // push 13
    // call bar
    ...
}
```
What is Java ByteCode?

Java ByteCode is the virtual instruction set of the Java virtual machine.

- One-byte instruction
- 256 possible opcodes (200+ in use)
- Stack-based computation

Suggested reading:

- Instructions listing: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
Java byte code instructions operates on 9 main different types:

- the 8 primitives types: byte, char, short, int, long, float, double
  - boolean, byte, short and char are sometimes treated as int and reference
  - a reference is a pointer to an Object in the heap
- long and double values takes two slots in the operand stack and local variables (all the other types takes one)
A mnemonic is a textual form of an operation
- Each mnemonic is encoded as a byte in the class file
- This byte is called an operation code or opcode

Examples:
- iadd: add two integers
- fmul: multiply two floats
- iload_1: load a long value from the local variable 1
# Mnemonics

<table>
<thead>
<tr>
<th>Prefix/Suffix</th>
<th>Type</th>
<th>Size (in byte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>s</td>
<td>short</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>char</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>int</td>
<td>4</td>
</tr>
<tr>
<td>l</td>
<td>long</td>
<td>8</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>a</td>
<td>reference</td>
<td>4 or 8</td>
</tr>
</tbody>
</table>

Instructions dealing with the stack or local variables start with a letter corresponding to the type. Examples:

- `iadd`: adds two integers
- `dadd`: add two doubles
### Arguments

Arguments follows an instruction

**Examples:**
- `bipush 5`: load a byte onto the stack
- `ldc "Hello World!": load a constant from the constant pool
- `iconst_0`: load 0 onto the stack

### Example

<table>
<thead>
<tr>
<th>Operand Stack</th>
<th>after <code>bipush 5</code></th>
<th>after <code>bipush 8</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Operands

Operands are taken from the operand stack and resulting value is produced on the stack

Examples:
- `iadd`

Example

<table>
<thead>
<tr>
<th>Operand Stack</th>
<th>after</th>
<th>Operand Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>iadd</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Christophe Dubach

Compiling Techniques
Exercise

- Write the ByteCode for the following expression: 5*(3+4)
- Write down the status of the stack after each instruction
Local variables can be retrieved via load/store instructions.

Frame for foo

<table>
<thead>
<tr>
<th>Local Variables</th>
<th>Operand Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

Example

```java
... foo(33) ...  
void foo(int a) {
    bipush 5
    istore 1
    bipush 7
    iload 0
    iadd
}
```
Function Call and Return Value

Function call
A function call is performed with one of the invoke instructions (dynamic, interface, special, static, virtual). When a function call occurs, a new frame is created and the arguments taken from the operand stack become local variables.

Return value
When a value is returned by the function, the current frame is destroyed and the return value is passed back to the callee onto its operand stack.
Example: SmallC/Java

```c
int add(int x, int y) {
    return x+y;
}
void main() {
    add(7,13);
}
```

Example: Java ByteCode

```java
int add(int x, int y) {
    iload_0
    iload_1
    ireturn
}
void main() {
    bipush 7
    bipush 13
    invokevirtual "add(I)I"
}
```
Branching instructions

Branching instructions takes a label that points to the place where to jump to. Java ByteCode offers both unconditional and conditional branches.

Unconditional branch:

- goto

Conditional branches:

- if_icmpeq jump if two stack operands are equals
- if_icmplt jump if the first operand is less than the second one
- ...
- ifeq jump if operand is 0
- ifge jump if operand is greater than 0
- ...

Christophe Dubach
Compiling Techniques
First the two parameters are loaded onto the operand stack using `iload_1` and `iload_2`. `if_icmple` then compares the top two values on the operand stack. This operand branches to byte code 7 if the first operand is less than or equal to the second one.
Next lecture:

- Introduction to code generation