Languages & Translators

LINGI2132

JVM Bytecode Instructions

Nicolas LAURENT
Université catholique de Louvain

Java to Bytecode

```java
public class Main {
    public int foo() {
        System.out.println("Hello, world!");
        return 42;
    }
}
```

```
> javac Main.java
> javap -v -p Main.class
```

or use https://javap.yawk.at

▼ **Main.class**

```
Classfile /tmp/6369527420976196219/classes/Main.class
  Last modified Apr 7, 2021; size 470 bytes
  SHA-256 checksum cc6d07f8dabc50d0867300b378ac6b3e638938c7e9db576885c9dd7625931624
  Compiled from "Main.java"
public class Main
  minor version: 0
  major version: 60
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #21                         // Main
  super_class: #2                         // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
```

Constant Pool

```
Constant pool:
   #1 = Methodref          #2.#3          // java/lang/Object."<init>":()V
   #2 = Class              #4             // java/lang/Object
   #3 = NameAndType        #5:#6          // "<init>":()V
   #4 = Utf8               java/lang/Object
   #5 = Utf8               <init>
   #6 = Utf8               ()V
   #7 = Fieldref           #8.#9          // java/lang/System.out:Ljava/io/PrintStream;
   #8 = Class              #10            // java/lang/System
   #9 = NameAndType        #11:#12        // out:Ljava/io/PrintStream;
  #10 = Utf8               java/lang/System
  #11 = Utf8               out
  #12 = Utf8               Ljava/io/PrintStream;
  #13 = String             #14            // Hello, world!
  #14 = Utf8               Hello, world!
  #15 = Methodref          #16.#17        // java/io/PrintStream.println:(Ljava/lang/String;)V
  #16 = Class              #18            // java/io/PrintStream
  #17 = NameAndType        #19:#20        // println:(Ljava/lang/String;)V
  #18 = Utf8               java/io/PrintStream
  #19 = Utf8               println
  #20 = Utf8               (Ljava/lang/String;)V
  #21 = Class              #22            // Main
  #22 = Utf8               Main
  #23 = Utf8               Code
  #24 = Utf8               LineNumberTable
  #25 = Utf8               LocalVariableTable
  #26 = Utf8               this
  #27 = Utf8               LMain;
  #28 = Utf8               foo
  #29 = Utf8               ()I
  #30 = Utf8               SourceFile
  #31 = Utf8               Main.java
```

Constructor

```
44 {
45   public Main();
46     descriptor: ()V
47     flags: (0x0001) ACC_PUBLIC
48     Code:
49       stack=1, locals=1, args_size=1
50         start local 0 // Main this
51          0: aload_0
52          1: invokespecial #1                  // Method java/lang/Object."<init>":()V
53          4: return
54         end local 0 // Main this
55       LineNumberTable:
56         line 2: 0
57       LocalVariableTable:
58         Start  Length  Slot  Name   Signature
59             0       5     0  this   LMain;
```

Method foo

```
61    public int foo();
62      descriptor: ()I
63      flags: (0x0001) ACC_PUBLIC
64      Code:
65        stack=2, locals=1, args_size=1
66          start local 0 // Main this
67             0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
68             3: ldc           #13                 // String Hello, world!
69             5: invokevirtual #15                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
70             8: bipush        42
71            10: ireturn
72          end local 0 // Main this
73        LineNumberTable:
74          line 4: 0
75          line 5: 8
76        LocalVariableTable:
77          Start  Length  Slot  Name   Signature
78              0      11     0  this   LMain;
```

Execution Model / Data

- Data storage
    - Locals (parameter, local variables)
    - Stack (no registers)
    - Objects with fields
    - Static fields
- Methods
    - Static methods
    - Virtual methods

Execution Model / Data

- The JVM deals with 32 bit values
    - `long` and `double` values occupy 2 slots
- Constant pool
    - Strings, including
        - string literals used in programs
        - (full) class names, field & method names
        - method & field (type) descriptors
    - big integers, and floating-point numbers
    - dynamically computed constant (new, mostly for optimization)

JVM Instructions

- Each instruction has a name
    - optionally some immediate byte operands
        - typically they reference the constant pool
        - or offsets (for jumps)
        - rarely: immediate (small) integer values (max short)
    - they can modify the stack (pop values, push values, reorder them)
- One-byte **opcode**: max 256 instructions!
    - Could do variable-length opcodes if needed.
    - Instruction set very stable
- https://docs.oracle.com/javase/specs/jvms/se16/html/jvms-6.html
- https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

Bytecode Instructions (all 200 of them!)

Load/Store

- Push (**load**) variables on the stack, **store** stack value in variables

- Pattern: (A/F/D/I/L)(LOAD/STORE)[_0/_1/_2/_3]
    - Address (Object), Float, Double, Integer, Long
    - e.g. ALOAD, ISTORE_0
    - 50 instructions!
    - Simple form: one parameter (one byte) to give the variable index.

Load/Store (extra)

- IINC
    - Parameters: two bytes
    - Adds the parameter value to the integer variable.
- WIDE
    - "modifier" to put in front of LOAD/STORE/IINC to access a wider range of local variables.
    - Use three bytes of variable space index instead of just one.
    - Preserve opcode space (will be a common theme)

Constants

- Push constants onto the stack (20 instructions)
    - ACONST_NULL
    - ICONST(_M1/_0/_1/_2/_3/_4/_5)
    - FCONST(_0/_1/_2)
    - (L/D)CONST(_0/_1)
    - LDC, LDC_W, LDC2_W
    - BIPUSH, SIPUSH
- In theory, you could do everything with LDC[2]_W
- But give extremely common operations their own opcode to reduce executable size!

Type Conversions

- Everything has to be explicit (e.g. `1 + 2L` in Java)
- value → value

- (I/D/F/L)2(I/D/F/L) (excluding identity)
- I2(B/C/S) (truncation)

Arithmetic/Binary Operations

- value, value → value

- (D / F / I / L)(ADD / DIV / MUL / SUB / NEG / REM)
- (I / L)(AND / OR / SHL / SHR / USHR / XOR)

Stack Manipulation

- DUP[2][_X1/_X2]
    - Duplicates a value.
    - DUP2: Duplicates two values / a double-sized value.
    - X1: value2, value1 → value1, value2, value1
    - X2: value3, value2, value1 → value1, value3, value2, value1
- POP[2]
- SWAP

Arrays

- (A/F/D/I/L/C/B/S)A(LOAD/STORE)
    - load: arrayref, index → value
      store: arrayref, index, value →
    - BALOAD: bytes or booleans! (otherwise booleans are int)
    - Why not store booleans as bits?
- [A]NEWARRAY
- MULTIANEWARRAY
    - parameters: type (constant pool ref), dimensions
    - count1, count2, ... → array
- ARRAYLENGTH

Field Access

- GETFIELD, PUTFIELD
    - Argument: field reference (constant pool ref)
        - Itself made of a class, name, field descriptor (type)
    - objectref → value / objectref, value → _

- GETSTATIC, PUTSTATIC
    - same
    - _ → value / value → _

Comparisons

- (D/F)CMP(G/L)
    - A > B → 1 , A < B → -1 , A == B → 0
    - A or B == NaN? G → 1 , L → -1
- LCMP
    - No ICMP! Directly embedded into jumps (next slide).
- INSTANCEOF
    - Parameter: class reference (constant pool ref)
    - True == 1, False == 0!
- CHECKCAST
    - Not really a comparison, throws an exception.

Jumps & Conditionals

- Parameter is always an offset in bytes (2 bytes, 4 for _W).

- GOTO[_W]
- IF_ACMP(EQ/NE) (two stack operands)
- IF_ICMP(EQ/NE/GE/GT/LE/LT) (two stack operands)
- IF(EQ/NE/GE/GT/LE/LT) (single stack operand)
    - EQ = 0, GT > 0, LT < 0
- IF[NON]NULL

Object Creation

- NEW
    - Parameter: class reference

- Calling the constructor?
    - INVOKESPECIAL <init>
    - The bytecode verifier checks that a constructor is called.

Method Invocation

- Parameter is always a method reference (class, name, method descriptor (type))
- INVOKESTATIC
    - Static methods, no `this` on stack.
- INVOKEVIRTUAL
    - "Regular" virtual calls, `this` on stack.
    - Get the object's class (may be different from static type) and get the method implementation from its method table.
    - The specific method implementation is not known at compile time (and might change each time the code is called)!
- INVOKEINTERFACE
    - Same, but for interface methods..
- INVOKESPECIAL
    - *Static* linking, but with `this` pointer.
    - private methods, constructors, super methods
- INVOKEDYNAMIC
    - next slide

Invokedynamic (1)

- Branding: "for dynamic languages!". Yes, but why?
- The wormhole instruction
    - First call goes to a bootstrap method that returns a `java.lang.invoke.CallSite`
        - You have to write this yourself.
        - Then the instruction is "patched"
    - Next calls go to the call site directly
        - Can be optimized just like a regular call!
    - If you save the CallSite, you can change the target method!

Invokedynamic (2)

- Lets your write "polymorphic inline caches"
    - Will be explained in Truffle talk and/or optimization lecture.
    - Basically: always use a method tailored to the types that have actually been seen (+ a condition to check if the values are supported)

Misc Instructions

- [A/F/D/I/L]RETURN
- ATHROW
- MONITOR(ENTER/EXIT)
- NOP
- LOOKUPSWITCH / TABLESWITCH

Constructor

```
44 {
45   public Main();
46     descriptor: ()V
47     flags: (0x0001) ACC_PUBLIC
48     Code:
49       stack=1, locals=1, args_size=1
50         start local 0 // Main this
51          0: aload_0
52          1: invokespecial #1                  // Method java/lang/Object."<init>":()V
53          4: return
54         end local 0 // Main this
55       LineNumberTable:
56         line 2: 0
57       LocalVariableTable:
58         Start  Length  Slot  Name   Signature
59             0       5     0  this   LMain;
```

Method foo

```
61    public int foo();
62      descriptor: ()I
63      flags: (0x0001) ACC_PUBLIC
64      Code:
65        stack=2, locals=1, args_size=1
66          start local 0 // Main this
67             0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
68             3: ldc           #13                 // String Hello, world!
69             5: invokevirtual #15                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
70             8: bipush        42
71            10: ireturn
72          end local 0 // Main this
73        LineNumberTable:
74          line 4: 0
75          line 5: 8
76        LocalVariableTable:
77          Start  Length  Slot  Name   Signature
78              0      11     0  this   LMain;
```

JIT: Register Allocation

- Most instructions map straightforwardly to underlying machine code
- However, most CPU architecture use registers
    - Registers are explicit (vs in-memory-hierarchy caches)
    - Mapping the JVM stack to memory would be too expensive
    - Most machine code instruction must use registers!
    - Use some register allocation algorithm
    - Good intro with a Java flavor: https://chrisseaton.com/truffleruby/register-allocation/

Differences with Java

- (Generic) Type Erasure
    - The JVM doesn't have generic types.
    - `List<T>` is only `List`
    - Added backward-compatibly in Java 1.5
    - Generic types are not *reified*
- No checked exceptions
    - All exceptions act like RuntimeException in the JVM
- These two concerns are semantic analysis concerns.

Loading & Verification

- Class files are *loaded* into the JVM when required.
    - The first time the class is being referred.
    - Class references (from the constant pool) are strings!
- After loading the file, the bytecode is verified.
    - Misc verifications: constructors called, final variables not re-written after the constructor was called, ...
    - ***Typestate***: any instruction reachable through multiple paths
      (e.g. first instruction after an if-else)
      must be reached with the stack in the same state (size & types)

Typestate

Any instruction reachable through multiple paths
must be reached with the stack in the same state (size & types).

- Java knows the type of every local & every slot on the stack.
    - Very different from machine code!
- It's not possible to iterate an array to copy it on the stack. (Why?)
- The stack can't overflow if you don't recurse.

Conclusion

- JVM
    - stack-based computation
    - storage: stack + locals + constant pool + fields
    - built-in GC — at the cost of memory control
    - built-in JIT compiler
- JVM Bytecode
    - low-level stack operations, jumps
    - transfers between stack and locals, fields, constant pool
    - highly-abstracted polymorphic method calls

Next Time

Generating JVM Bytecode with ASM