Operational Semantics

Lecture 2: (Java) Bytecode

Simon Castellan

SOS, Master Recherche Science Informatique, U. Rennes 1

2021-2022

Semantics of Java bytecode

An example of an operational semantics in the form of an abstract machine (the Java virtual machine).

References:

- Tim Lindholm , Frank Yellin, Gilad Bracha, Alex Buckley. Java Virtual Machine Specification, Java SE 8 Edition, 2015.
- Robert F. Stärk, Joachim Schmid, Egon Börger. Java and the Java Virtual Machine – Definition, Verification, Validation. 2001.
- S. Freund , J. Mitchell : A Type System for the Java Bytecode Language and Verifier, Journal of Automated Reasoning, Volume 30, Issue 3-4, Pages: 271 - 321. 2003.

Outline

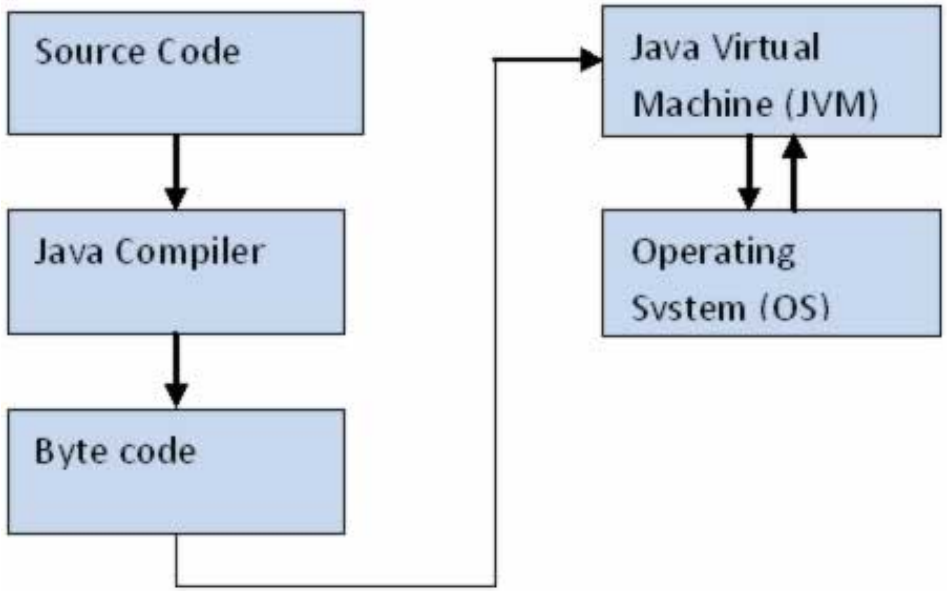
Java

Java: a class-based, object-oriented programming language.

```
public class Bicycle{
    private int gear;
    private int id;
    private int speed;

    private static int numberOfBicycles = 0;

    public Bicycle(int startSpeed, int startGear){
        gear = startGear;
        speed = startSpeed;
        id = ++numberOfBicycles;
     }

    public void setGear(int newValue){
      gear = newValue;
    }
}
public class MountainBike extends Bicycle {
  // the MountainBike subclass has one field
    public int seatHeight;
    ...
}
```

JVM and Java bytecode

To enhance portability, Java defines a bytecode language that is interpreted by the Java virtual machine.



Java bytecode: factorial

The JVM is a stack machine with registers.

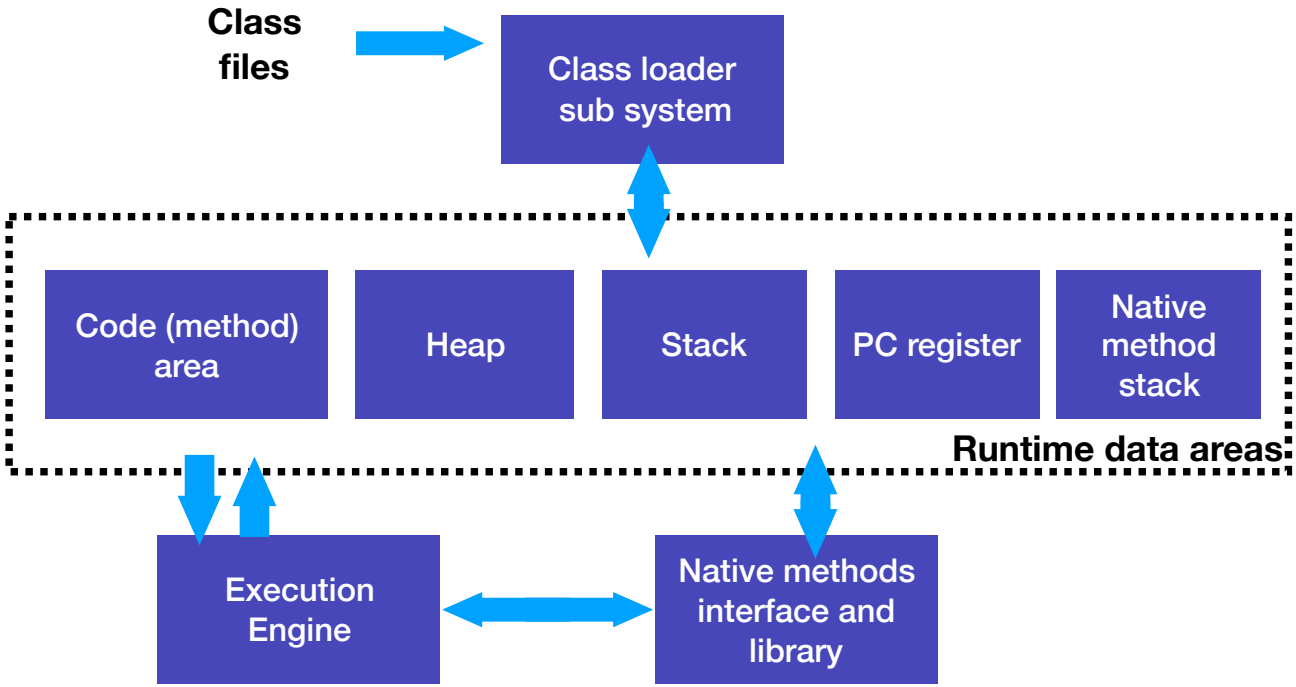
Source code

```
public static int factorial(int n) {
  int res = 1 ;
  for (int i = 1; i <= n; i++) { res = res * i; }
  return res;
}
```

Bytecode (as produced by `javap -v -c Factorial.class`)

```
int factorial(int); descriptor: (I)I flags: ACC_PUBLIC, ACC_STATIC, stack=2, locals=3, args_size=1
         0: iconst_1                    // push the integer constant 1
         1: istore_1                    // store it in register 1 (the res variable)
         2: iconst_1                    // same for register 2 (the i variable)
         3: istore_2
         4: goto          14            // go to PC 14
         7: iload_1                     // push value of register 1 (res)
         8: iload_2                     // push value register 2 (i)
         9: imul                        // multiply the two integers at top of stack
        10: istore_1                    // pop result and store it in register 1
        11: iinc          2, 1          // increment register 2 (i) by 1
        14: iload_2                     // evaluate loop condition
        15: iload_0                     //
        16: if_icmple     7             // iterate or exit loop
        19: iload_1                     // push value of register 1 (res)
        20: ireturn                     // return the top integer to the caller
```

The Java virtual machine state



The JVM has several components:

Code component includes bytecode of all methods of all classes loaded.

Program counter points to the next instruction to execute.

Heap (memory) of objects created while program is running.

Frame stack of methods under execution. Each frame contains an operand stack and local variables (registers). (*ignored in our modelisation*)

The Java virtual machine specification

iadd *iadd*

Operation Add `int`

Format

iadd

Forms *iadd* = 96 (0x60)

Operand Stack ..., *value1*, *value2* →

..., *result*

Description Both *value1* and *value2* must be of type `int`. The values are popped from the operand stack. The `int` *result* is *value1* + *value2*. The *result* is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type `int`. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Despite the fact that overflow may occur, execution of an *iadd* instruction never throws a run-time exception.

Outline

Idealized bytecode syntax

Instruction ::=

| `nop`	*Does nothing*
Stack manipulation	
| `const` *c*	*Push constant*
| `pop`	*Pop and discard top value*
| `dup`	*Duplicate top entry*
| `dup2`	*Duplicate top two entries*
| `swap`	*Swap top two entries*
| `numop` *op*	*Arithmetic operation*
Local registers manipulation	
| `load` *x*	*Push value of x*
| `store` *x*	*Pop top entry into x*

Some bytecodes are typed (`iconst`, `aload`) to indicate whether they operate on integers, references, arrays, etc.

Idealized bytecode syntax (continued)

Instruction ::= ...

Manipulation of program counter	
| `ifle` *pc*	*Conditional jump*
| `goto` *pc*	*Inconditional jump*
Object manipulation	
| `new` *cl*	*Creation of object*
| `putfield` *f*	*Object field modification*
| `getfield` *f*	*Object field access*

Real Java bytecode: more "if"s: `if_eq`, `if_ge`, `if_nonnull`, `if_acmpeq`,...

Fields *f* are of the form *C* : `name` : *t* (*C*: class declaring the field; and *t* is its type)

Program structure

A program P is:

- a **set of instructions**: $\text{instructionAt}_P(m, pc)$ fetches the opcode at program counter pc in P
- a set of **classes** describing the list of fields: $\text{Fields}(c)$ describes the list of fields of class c.

A function in the code of method m in program P.

Outline

Operational semantics

- Define semantic domains
- Define the SOS transition system
- Transition system parametrised by a specific program P.
- **Difficulty**: Low-level control flow (jumps).

Operational semantics

Semantic domains

$$
\begin{array}{rcll}
\text{Value} & = & \text{num } n & n \in \mathbb{Z} \\
 & & \text{ref } r & r \in \text{Reference} \\
 & & \text{null} & \\
\text{Stack} & = & \text{Value}^* & \text{(a list of values, can be empty)} \\
\text{LocalVar} & = & \text{Var} \hookrightarrow \text{Value} & \\
\text{CallFrame} & = & \text{ProgCount} \times \text{LocalVar} \times \text{Stack} & \\
\text{Object} & = & (\text{FieldName} \hookrightarrow \text{Value}) & \\
\text{Heap} & = & \text{Reference} \hookrightarrow \text{Object} & \\
\text{State} & = & \text{Heap} \times \text{CallFrame} &
\end{array}
$$

Operational semantics

Example rule (nop)

$$\frac{\text{instructionAt}_P(pc) = \textbf{nop}}{\langle\!\langle h, \left\langle \text{pc} , l, s \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \text{pc+1} , l, s \right\rangle\!\rangle}$$

Operational semantics – Basic stack operations.

We use $c, n \ldots$ for numeric values, loc for references, and v for arbitrary values.

$$\frac{\text{instructionAt}_P(pc) = \texttt{nop}}{\langle\!\langle h, \left\langle \boxed{\text{pc}}, l, s \right\rangle \rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \boxed{\text{pc+1}}, l, s \right\rangle \rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{const}\ c}{\langle\!\langle h, \left\langle \boxed{\text{pc}}, l, \boxed{s} \right\rangle \rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \boxed{\text{pc+1}}, l, \boxed{c :: s} \right\rangle \rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{pop}}{\langle\!\langle h, \left\langle \boxed{\text{pc}}, l, \boxed{v :: s} \right\rangle \rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \boxed{\text{pc+1}}, l, \boxed{s} \right\rangle \rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{dup}}{\langle\!\langle h, \left\langle \boxed{\text{pc}}, l, \boxed{v :: s} \right\rangle \rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \boxed{\text{pc+1}}, l, \boxed{v :: v :: s} \right\rangle \rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{dup2}}{\langle\!\langle h, \left\langle \boxed{\text{pc}}, l, \boxed{v_1 :: v_2 :: s} \right\rangle \rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \boxed{\text{pc+1}}, l, \boxed{v_1 :: v_2 :: v_1 :: v_2 :: s} \right\rangle \rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{swap}}{\langle\!\langle h, \left\langle \boxed{\text{pc}}, l, \boxed{v_1 :: v_2 :: s} \right\rangle \rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \boxed{\text{pc+1}}, l, \boxed{v_2 :: v_1 :: s} \right\rangle \rangle\!\rangle}$$

Operational semantics – Arithmetic and local variables.

$$\frac{\text{instructionAt}_P(pc) = \texttt{numop}\ op}{\langle\!\langle h, \left\langle \text{pc}, l, n_1 :: n_2 :: s \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \text{pc+1}, l, [\![op]\!](n_1, n_2) :: s \right\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{load}\ x}{\langle\!\langle h, \left\langle \text{pc}, l, s \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \text{pc+1}, l, l(x) :: s \right\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{store}\ x}{\langle\!\langle h, \left\langle \text{pc}, l, v :: s \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \text{pc+1}, l[x \mapsto v], s \right\rangle\!\rangle}$$

Operational semantics – Control transfer.

$$\frac{\text{instructionAt}_P(pc) = \texttt{ifle}\ pc' \qquad n \leq 0}{\langle\!\langle h, \left\langle \text{pc}, l, n :: s \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \text{pc}', l, s \right\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{ifle}\ pc' \qquad n > 0}{\langle\!\langle h, \left\langle \text{pc}, l, n :: s \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \text{pc+1}, l, s \right\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{goto}\ pc'}{\langle\!\langle h, \left\langle \text{pc}, l, s \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle \text{pc}', l, s \right\rangle\!\rangle}$$

Operational semantics – Objects.

NB: we present here a simplified object initialization procedure.

$$\frac{\text{instructionAt}_P(pc) = \texttt{new}\ cl \qquad \text{Fields}(cl) = \{f_1, \ldots, f_n\} \qquad loc \notin dom(h) \qquad o := [f_1 \mapsto \diamond, \ldots, f_n \mapsto \diamond] \qquad h' = h[loc \mapsto o]}{\langle\!\langle\ h\ , \left\langle\ \text{pc}\ , l, \ s\ \right\rangle\!\rangle \rightarrow \langle\!\langle\ h'\ , \left\langle\ \text{pc+1}\ , l, \ loc :: s\ \right\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{putfield}\ f \qquad h(loc) = o \qquad o' = o[f \mapsto v]}{\langle\!\langle\ h\ , \left\langle\ \text{pc}\ , l, \ v :: loc :: s\ \right\rangle\!\rangle \rightarrow \langle\!\langle\ h[loc \mapsto o']\ , \left\langle\ \text{pc+1}\ , l, \ s\ \right\rangle\!\rangle}$$

$$\frac{\text{instructionAt}_P(pc) = \texttt{getfield}\ f \qquad h(loc) = o}{\langle\!\langle h, \left\langle\ \text{pc}\ , l, \ loc :: s\ \right\rangle\!\rangle \rightarrow \langle\!\langle h, \left\langle\ \text{pc+1}\ , l, \ o(f) :: s\ \right\rangle\!\rangle}$$

The initial value $\diamond$ is 0 for integers and `null` for object references.

Operational semantics

- How would the initial state look like?
- Implement a simple bytecode interpreter (in OCaml, Scala. . .)

- Is this semantics deterministic, and why is it so?
- How much this semantics is blocking?
- How to ensure that some of these blocking configurations will never be reached?

Other language features

Aspects of the language not dealt with here include:

- Methods calls
- Visibility modifiers (`public`, `private`, `protected`,...)
- Multi-threading (`monitorenter`, `monitorexit`)
- Exceptions
- The constant pool
- Arrays
- Long integers