How JVM works

- Frames - space for calculation
  - operand stack, local variables, ...
- Bytecode instructions
- Method calls create frames
- Heap for objects
- Symbolic references to methods
Frames

Each time you call a method in Java, it gets a new frame constructed for it. The frame has:

- Storage space for local variables
- Space for an operand stack
- Program counter & stack pointer
- Reference back to caller frame
- Other stuff

Same principle as registers for virtual machine, not gcc
Frame

Space for operand stack

local variables

PC, SP

other stuff

caller's frame
Entries are 4 bytes each on operand stack or local variables
- enough for bool, byte, short, int, char, float
- also enough for any reference value
- for long, double need two consecutive entries
What is a local variable?

1. "this" (for a non-static method)
   - Reference to the object "this"
   - Kept as local variable with index 0

2. Parameters of method
   - Indexes start 0 (static) or 1 (non-static)

3. Variables declared in method
   - Sometimes "local variable" specifically means 0
What is not a local variable?

1. Instance variables (non-static fields)
2. Class variables (static fields)

Both declared outside method
public class PosVal {
    private static int nextSerial = 0;
    private int serial;
    private int val; //invariant: val >= 0

    public PosVal(int initVal) {
        int v = initVal;
        if (v < 0) {
            v = -v;
        }
        val = v;
        serial = nextSerial;
        nextSerial += 1;
    }
}

**How many local variables does this constructor have?**

What is a local variable?

1. "this" (for a non-static method)
   - Reference to the object "this"
   - Rept as local variable with index 0
2. Parameters of method
   - Indexes start 0 (static) or 1 (non-static)
3. Variables declared in method
public class PosVal {
    private static int nextSerial = 0;
    private int serial;
    private int val; //invariant: val >= 0

    public PosVal(int initVal){
        int v = initVal;
        if (v < 0){
            v = -v;
        }
        val = v;
        serial = nextSerial;
        nextSerial += 1;
    }
}

THREE

1 parameter

0 "this"

instance variables

class variable

indexes = slot numbers
Local variables in JVM

JVM doesn't know the names of the local variables (from Java source).
- Refers to them by slot numbers starting at 0

```
...         ...  operand stack
```

- For non-static method, variable 0 is this
  Then parameters start at slot 1, then other local variables
- For static method, parameters start at slot 0
- For long, double use number of first of pair of slots
  \[ 8 \text{ bytes} = 2 \text{ slots} \]
Can operand stack ever overflow (run out of space)?

No.

- Compiler works out exactly how much space needed
- Loader checks each method to verify no overflow or underflow possible

StackOverflowError is different - a new frame is needed, but there's not enough memory.
Bytecode instructions

Each instruction has at least one byte
- opcode

May have more operand(s).

A single operand may be
2 or more bytes together as
an integer. Then "bigendian" - most
significant bytes come first.

For each opcode there's a human-readable mnemonic
### Arithmetic on operand stack

E.g. `add` - adds top two stack entries

\[ ... \text{val1, val2} \rightarrow ... \text{val1 + val2} \]

**BUT different opcodes for different types**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode (hex)</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iadd</code></td>
<td>60</td>
<td>int</td>
</tr>
<tr>
<td><code>ladd</code></td>
<td>61</td>
<td>long</td>
</tr>
<tr>
<td><code>fadd</code></td>
<td>62</td>
<td>float</td>
</tr>
<tr>
<td><code>dadd</code></td>
<td>63</td>
<td>double</td>
</tr>
</tbody>
</table>

Similarly for other operations.

Also sometimes `b` - byte, `s` - short,

\[ a \text{ - address (reference)} \]
Errors? What happens if ... 

- use fadd on int entries?
- there's only one entry on stack?
- other obvious mistakes?

No checks when operation is executed

BUT JRE verifies code when it loads a class
- checks types used consistently etc

Safeguard against security holes
Pushing constants on the stack

b for byte - 1 byte operand

\[ \text{push } N \leftrightarrow \text{one operand byte} \]

\[ \cdots \rightarrow \cdots, N \]

i for int - pushes 4 bytes

“sign extends” operand to 4 bytes

pushes result on stack

Similarly, sipush two operand bytes

short
Simple opcodes for common constants

e.g., for int

7 different opcodes

\{ 
\textit{i}const\_m1 \\
\textit{i}const\_0 \\
\textit{i}const\_1 \\
\vdots \\
\textit{i}const\_5
\}

no operand needed

\ldots \Rightarrow \ldots, \ldots, 0, \ldots, 5

pushes
\{-1, 0, 1, 2, 3, 4, 5\}
Load

load variable onto stack
e.g. iload slot number

push int local variable at given slot onto stack

Similarly lload, fload, dload, aload

JRE verifier: must use types consistently
e.g. can't load as integer than use as address

It's Java, not C++!
1-byte loads

Special opcodes with no operand for slot numbers 0, 1, 2, 3

E.g. iload_0, iload_1, iload_2, iload_3

Store

Reverse of load: pops top of operand stack into variable

E.g. istore slot number

astore_2
Jumps

Must be within current method.
Can't jump to a different method.

Unconditional

goto 2 byte offset

Operand is added to address of goto opcode to give address of next opcode to execute

Similarly: goto-w 4 byte offset
Conditional jumps

As before, but using offsets for operand \( N \).

Also, e.g.,

\[
\text{if}_{- \text{i cmp eq}}
\]

\[\text{int comparison}\]

Conditional jumps - e.g.

\[
\text{ifeq } N - \text{jumps to } N \text{ if } \text{val} = 0
\]

\[
\ldots, \text{val} \rightarrow \ldots
\]

\[
\text{iffc}_{- \text{i cmp eq}} N - \text{jumps to } N \text{ if } \text{val1} = \text{val2}
\]

\[
\ldots, \text{val1}, \text{val2} \rightarrow \ldots
\]

\[
\text{jump if } \text{val} \{= \}
\]

6 operators if eq, if lt, if ge, etc.
also if cmp eq, if cmp lt, etc.
Call and return

Say method B calls A(p0, p1) & A returns a result

1. B calculates actual parameter on its operand stack
2. Construct stack frame for A, with parameters taken from B’s operand stack
3. A calculates its result on its operand stack
4. Result transferred to B’s operand stack
5. Return to B, throw away A’s stack frame

On B’s operand stack: A(p0, p1) has effect of

\[ \ldots, \text{Po}, \text{p1} \rightarrow \ldots, \text{result} \]
B calls A ... A executes ... A returns to B

Frames for A

Frames for B

opnd stack (empty)
**Current frame** - for method being executed

- While A is executed, its frame is current
  - B's frame is not.

  [Greied out on previous slide]

- When A returns, its frame is destroyed
  - B's frame becomes current again
Saving return addresses

- Each frame has its own pc
- While A is executed, its pc is used
  - When A returns, B resumes with its old pc value
- B's local variables are also unchanged by A.

Linked frames have the effect of a return stack.

In fact, the chain of linked frames is officially called a stack in JVM.