How JVM works

Outline

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

Frames

or stack frames

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

same principle as registers

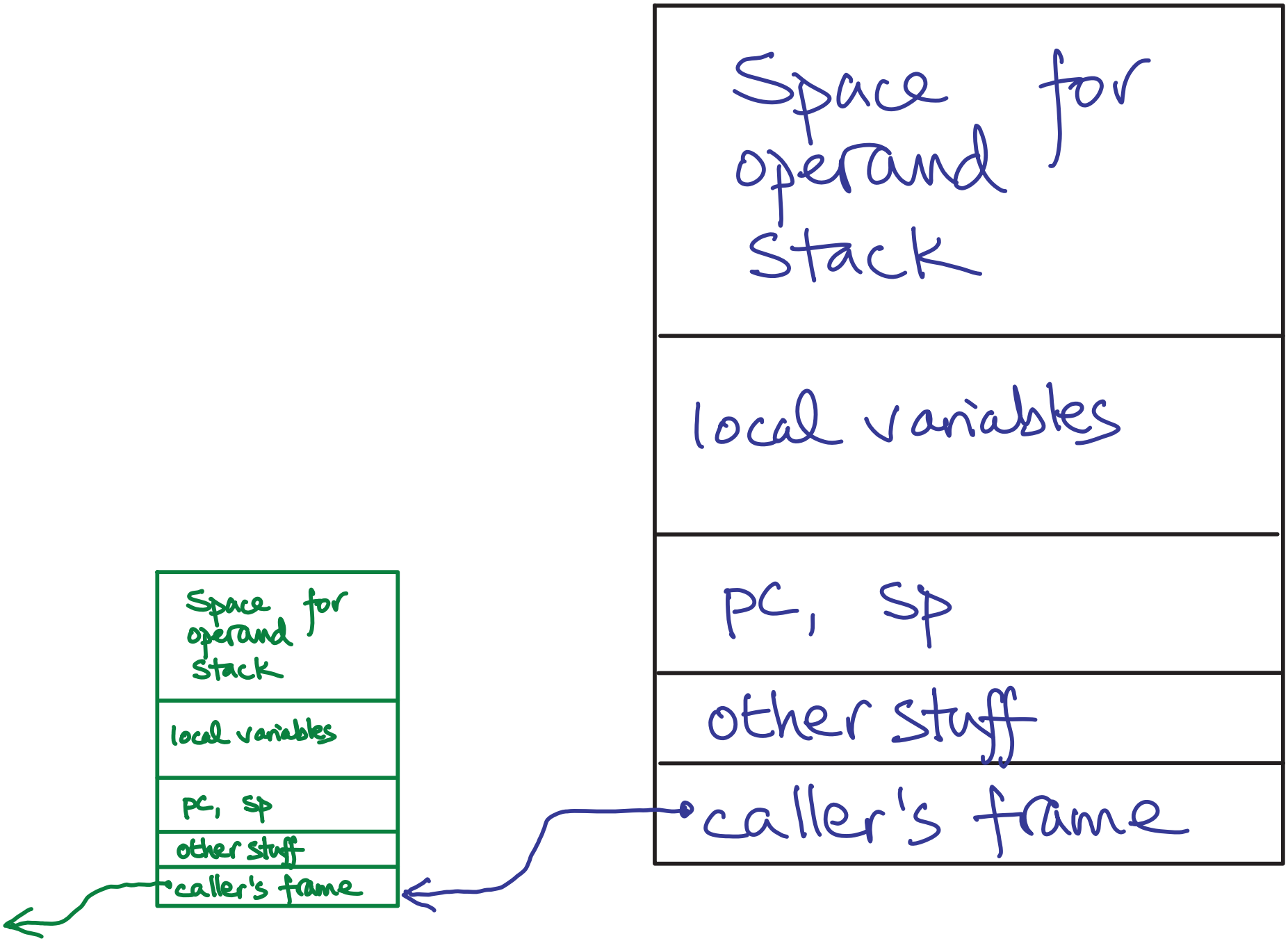
The frame has —

- storage space for local variables
- space for an operand stack
- program counter & stack pointer

for virtual machine, not CPU

- reference back to frame of caller
- other stuff

Frame


Space for operand stack
local variables
pc, sp
other stuff
caller's 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 (1 bit), byte (1 byte), short (2), int (4), char (2), float (4)
- also enough for any reference (4) value

- for long (8), double (8) 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 ③

What is not a local variable?

① Instance variables
(non-static fields)

② Class variables
(static fields)

} both declared outside method

How many local variables does this constructor have?

```
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;
    }
```

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

How many local variables does the constructor have?

THREE

```
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;
    }
```

0 "this"

1 parameter

2

instance variables

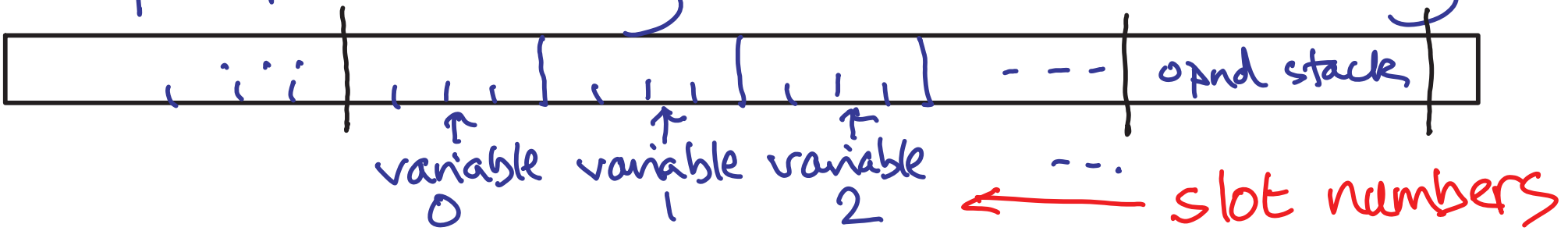
class variable

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



- 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 bytes = 2 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

~256 different ones! I shan't describe them all

Each instruction has at least one byte
- opcode

May have more operand(s)

Two separate meanings of "operand" - here, & entry on operand stack

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

..., val1, val2 ⟹ ..., val1 + val2

BUT different opcodes for different types

mnemonic	opcode (hex)	type	
iadd	60	int	i
ladd	61	long	l
fadd	62	float	f
dadd	63	double	d

Similarly for other operations.

Also sometimes b – byte s – short
a – address (reference)

Errors?

What happens if ...

- use fadd on int entries?
- there's only one entry on stack? → stack underflow
- 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

push N ← one operand byte

... → ..., N

b for byte - 1 byte operand

↘

bipush one operand byte

↑

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:

- iconst_m1
- iconst_0
- iconst_1
- ⋮
- iconst_5

} no operand needed

... ⟹ ..., -1, 0, ⋮, 5

pushes -1, 0, 1, 2, 3, 4 or 5

Load

load variable onto stack

push variable

e.g. iload slot number

1-byte operand

int

push int local variable at given slot onto stack

Similarly lload, fload, dload, aload

long float double ref (address)

JRE verifier: must use types consistently

e.g. can't load as integer then 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 2byte offset

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

operand is an offset

Similarly: goto_w 4 byte offset

(wide)

Conditional jumps

As before,
but using offsets
for operand N.

Also, e.g.

if_icmpeq
↑
int comparison

Conditional jumps – e.g.
ifeq N – jumps to N if val = 0
..., val → ...
if_cmpeq N – jumps to N if val1 = val2
(compare)
..., val1, val2 → ...

jump if val { = < ≤ ≠ > ≥ } 0

eq
lt
le
ne
gt
ge

6 operators ifeq, iflt, ifle etc;
also if_cmpeq, if_cmplt, etc.

Call and return

two parameters

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

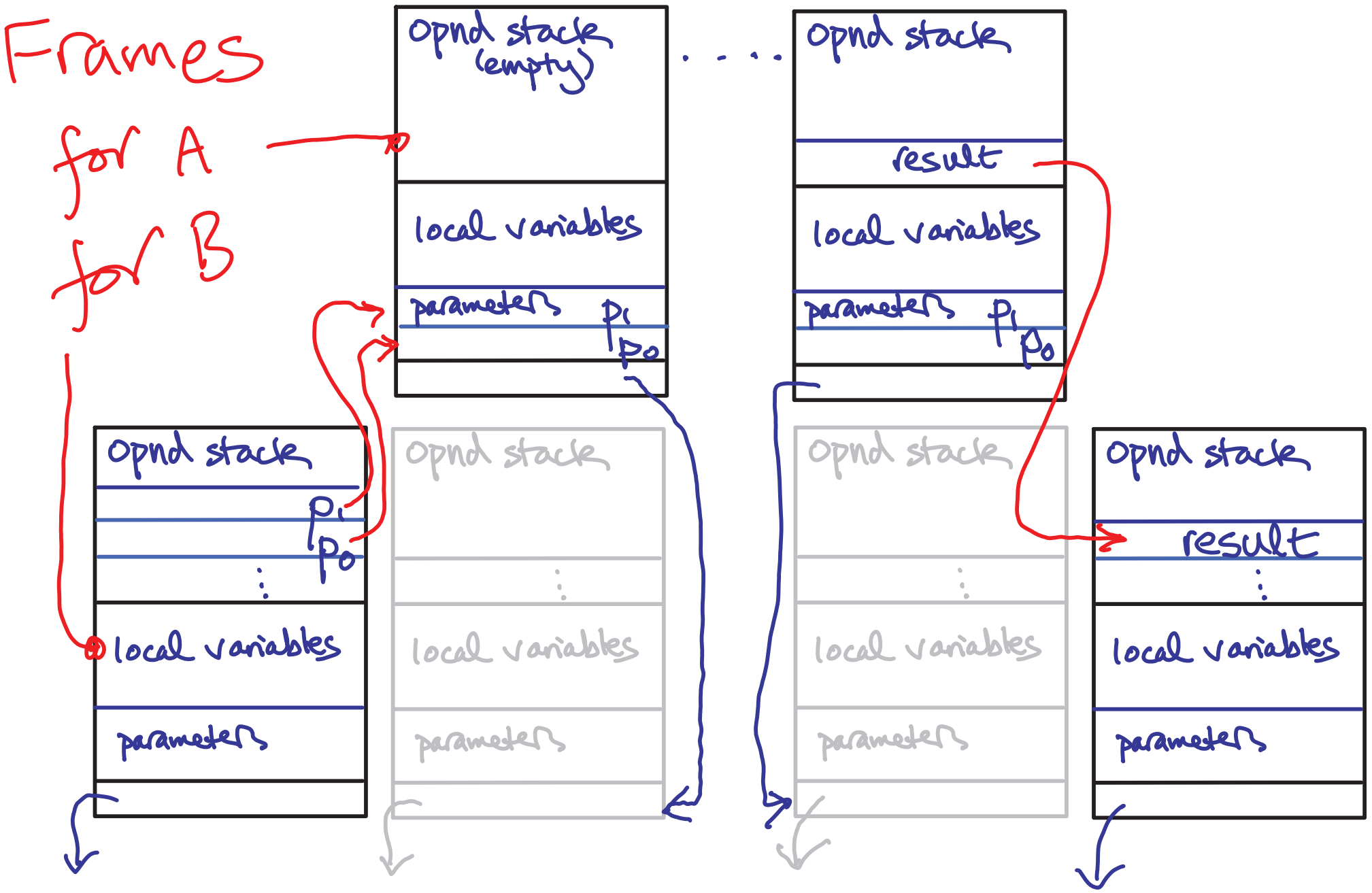
not void

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

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

..., p0, p1 → ..., result

B calls A ... A executes ... A returns to B



Current frame – for method being executed

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

Greyed 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