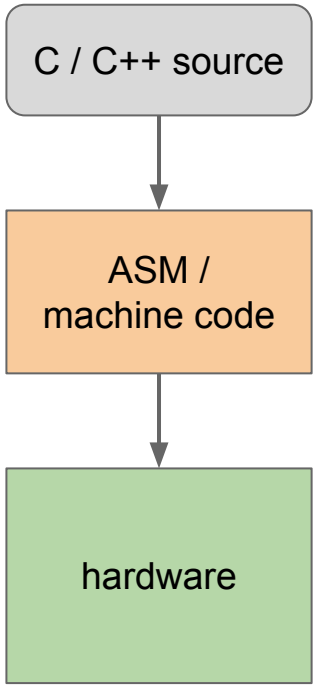
JVM

CSC 236

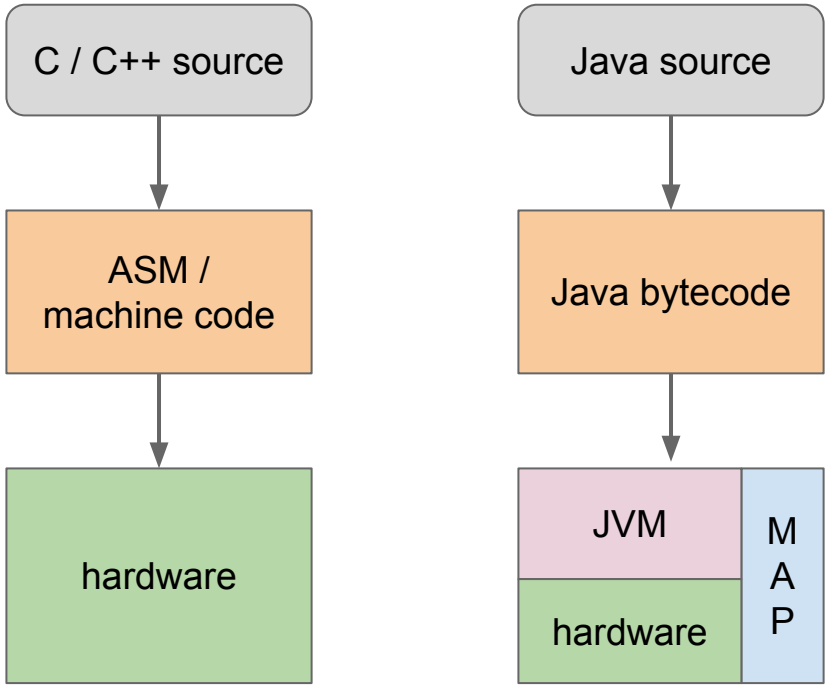
Java virtual machine

- Java compile
  - Generates *bytecode*
  - From source
- JVM execute
  - Bytecode
  - On any machine
- Motto: write once run anywhere
  - Eliminate problem of incompatible executables

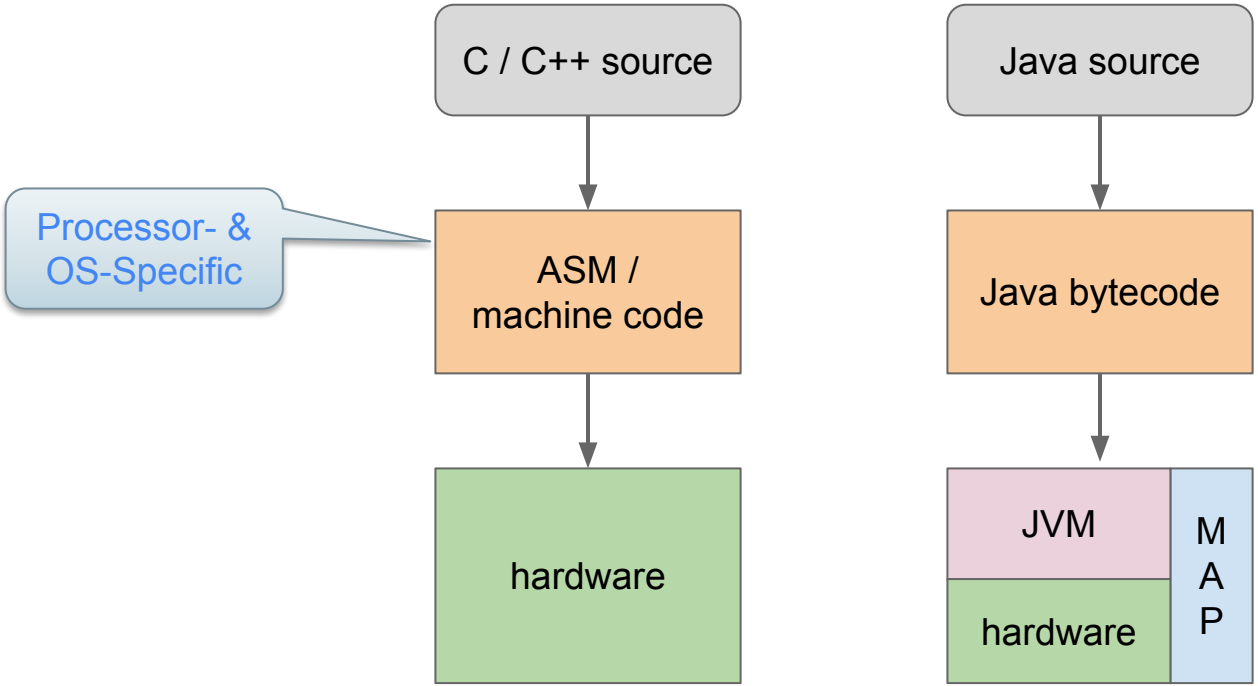
JVM — Java Virtual Machine



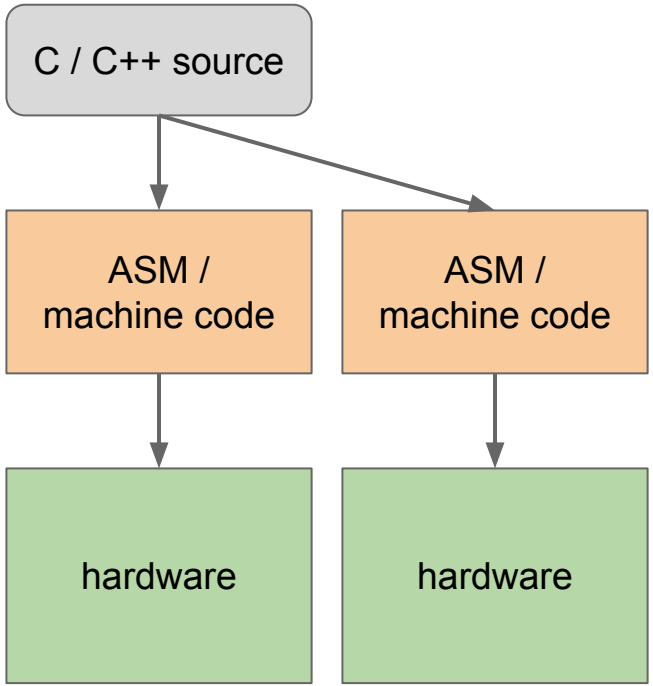
JVM — Java Virtual Machine



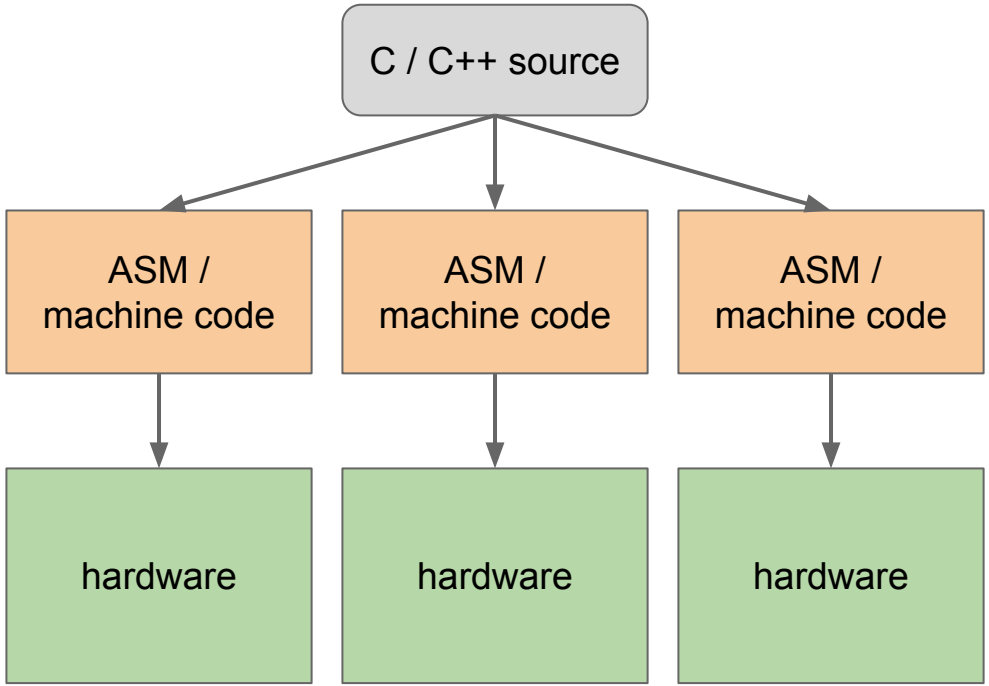
Traditional (compiled) language vs. Java



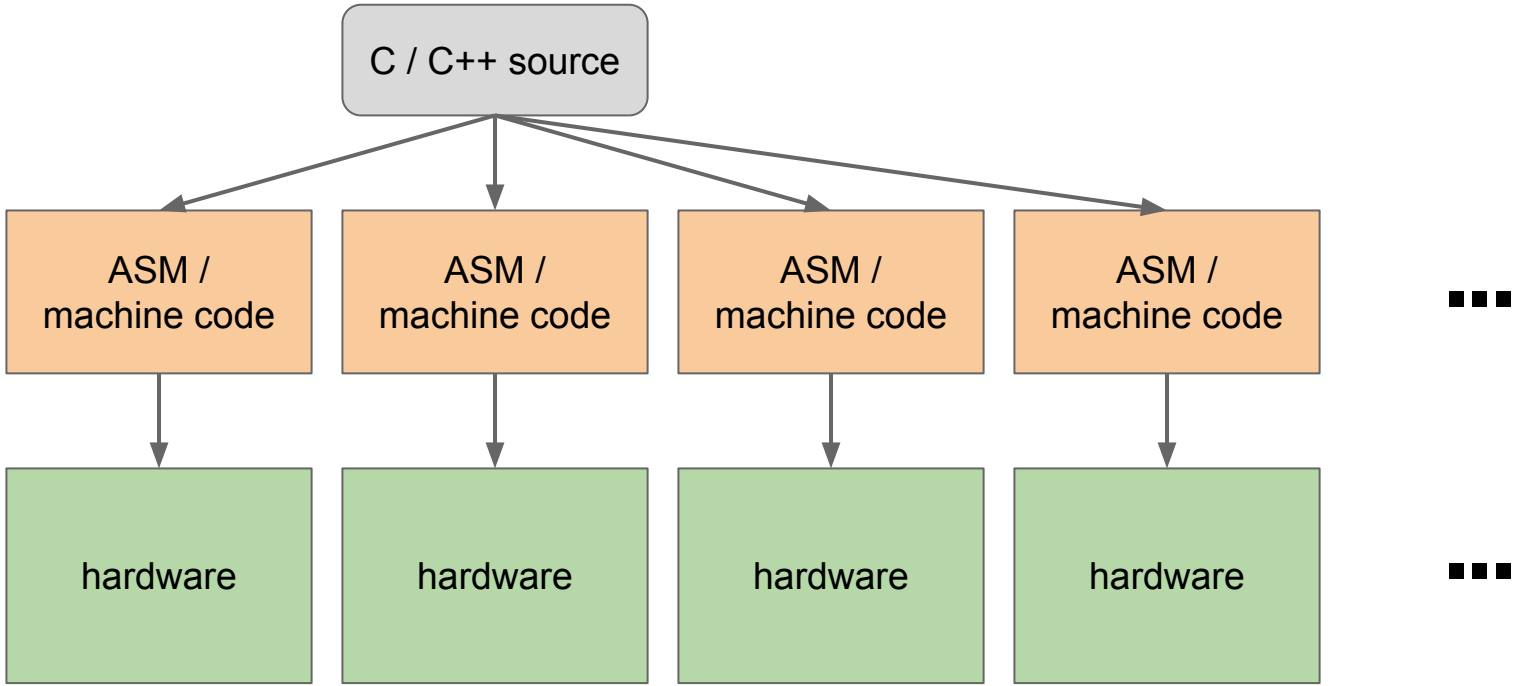
Traditional (compiled) language vs. Java



Traditional (compiled) language vs. Java

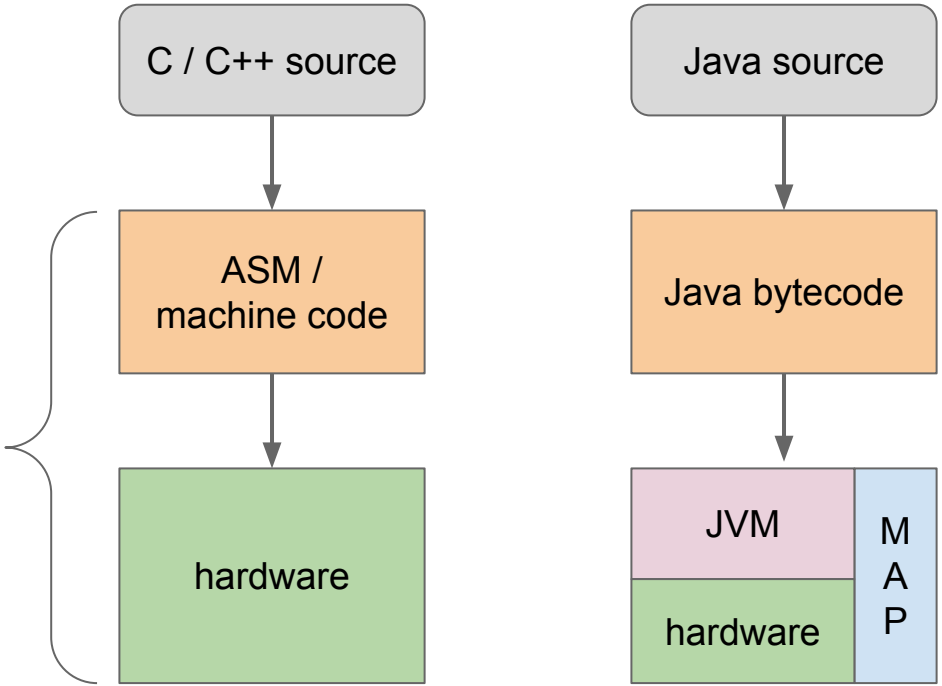


Traditional (compiled) language vs. Java



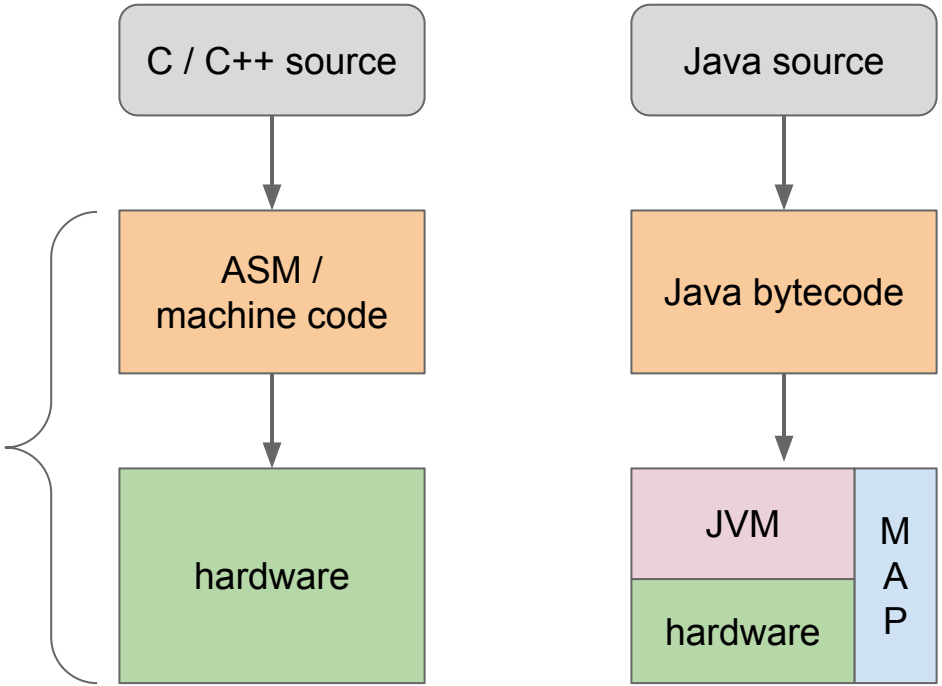
Traditional (compiled) language vs. Java

unique stack needed for each architecture/ OS combination



Traditional (compiled) language vs. Java

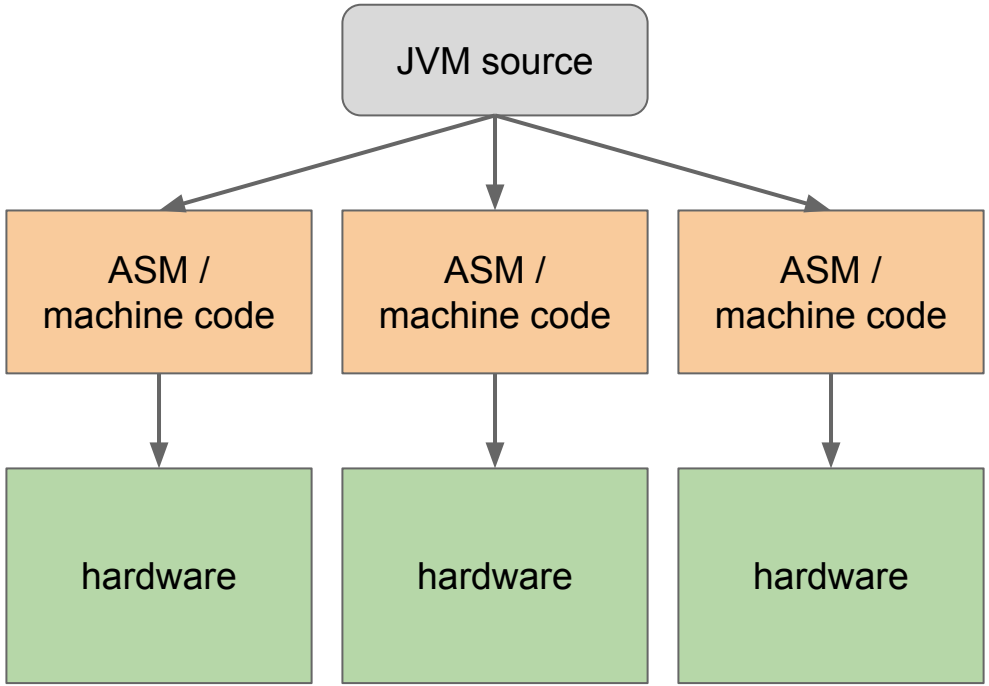
unique stack needed for each architecture/ OS combination



byte code is for a single abstract machine

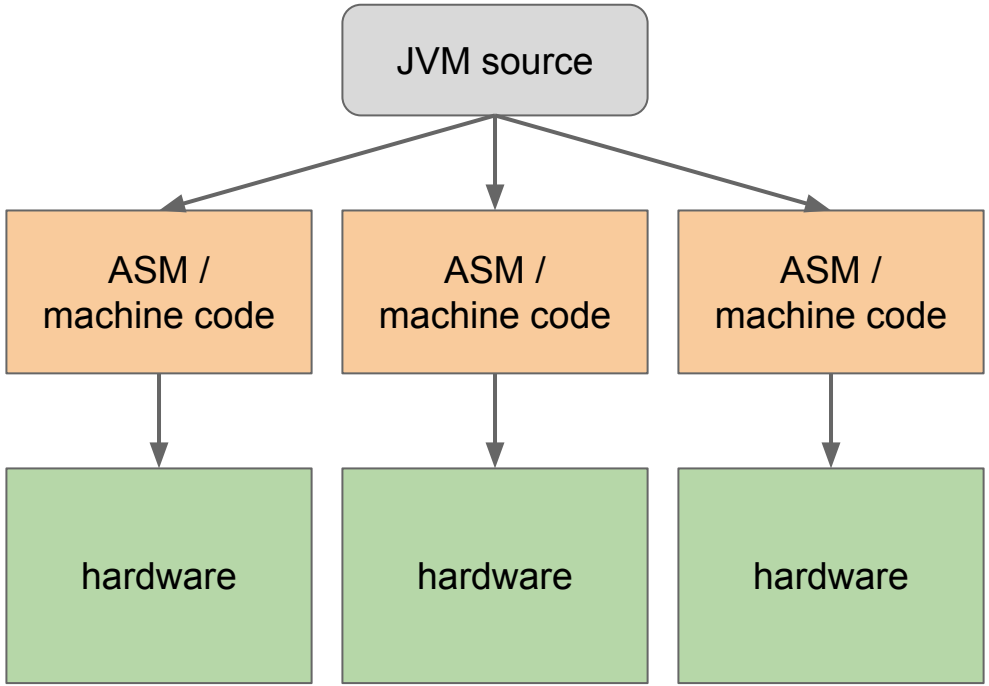
JVM simulates abstract machine on any hardware

JVM



- JVM is a program
- Written in C
- Compiled for each platform

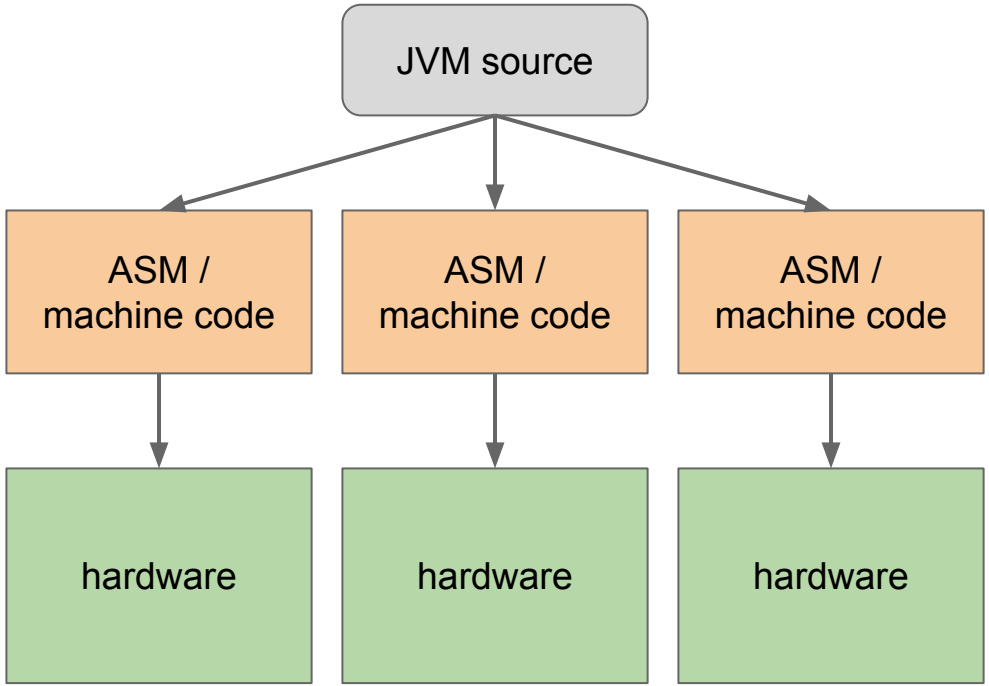
JVM



- JVM is a program
- Written in C
- Compiled for each platform

- Isn’t that the same problem?

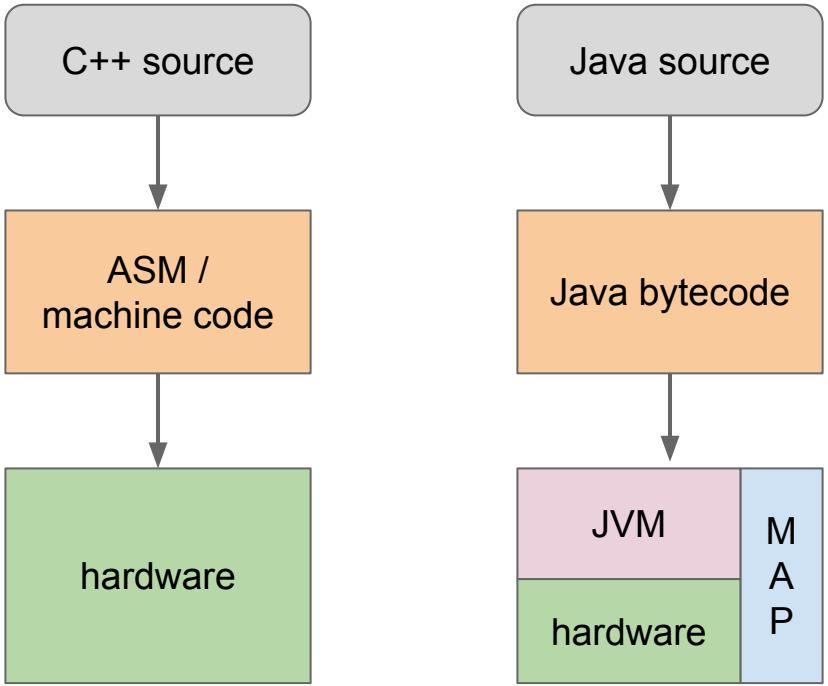
JVM



- JVM is a program
- Written in C
- Compiled for each platform

- Isn't that the same problem?
  - Not really
  - One JVM source
  - C compilers exist for target hardware and OS.
  - We just have to build the JVM once for each
  - Then, **all** java programs can be run on that platform.

Traditional (compiled) language vs. Java



unique stack needed for each architecture/ OS combination

byte code is for a single abstract machine

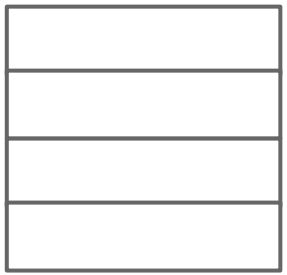
JVM simulates abstract machine on any hardware

MAP specific to arch/OS

Stack machine

- No registers
- Operands
  - Source: pop from stack
  - Destination: push onto stack
- Example
  - `C = A + B`

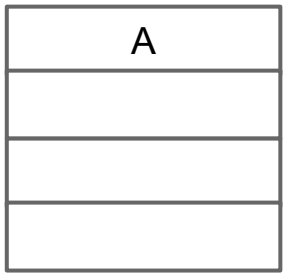
```
push A
push B
add
pop C
```



Stack machine

- No registers
- Operands
  - Source: pop from stack
  - Destination: push onto stack
- Example
  - `C = A + B`

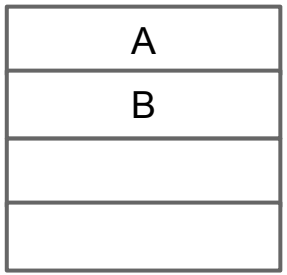
```
push A
push B
add
pop C
```



Stack machine

- No registers
- Operands
  - Source: pop from stack
  - Destination: push onto stack
- Example
  - `C = A + B`

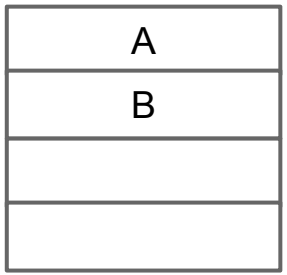
```
push A
push B
add
pop C
```



Stack machine

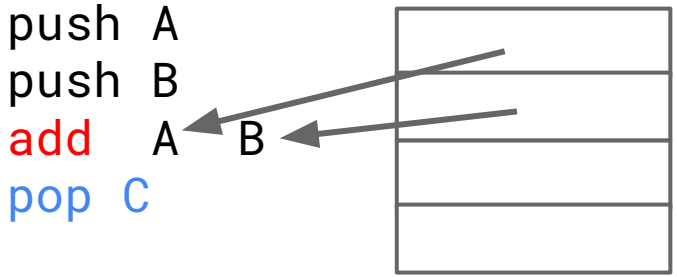
- No registers
- Operands
  - Source: pop from stack
  - Destination: push onto stack
- Example
  - `C = A + B`

```
push A
push B
add
pop C
```



Stack machine

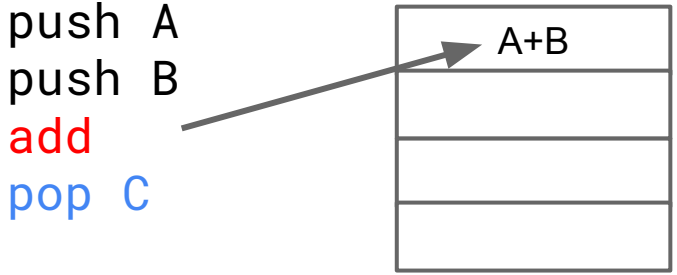
- No registers
- Operands
  - Source: pop from stack
  - Destination: push onto stack
- Example
  - `C = A + B`



Stack machine

- No registers
- Operands
  - Source: pop from stack
  - Destination: push onto stack
- Example
  - `C = A + B`

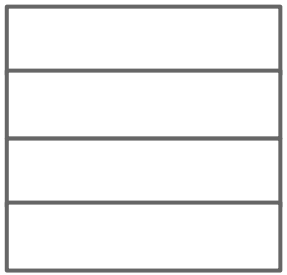
```
push A
push B
add
pop C
```



Stack machine

- No registers
- Operands
  - Source: pop from stack
  - Destination: push onto stack
- Example
  - `C = A + B`

```
push A
push B
add
pop C
```



Why a stack machine?

- Compact object code
  - Operands are often implicit (from the stack)
  - So, “add”
    rather than “add x y”
- Less complex
  - No complex addressing modes
  - So, “inc”
    rather than “inc [si+bx-7]”

Why a stack machine?

- Compact object code
  - Operands are often implicit (from the stack)
  - So, “add”
    rather than “add x y”
- Less complex
  - No complex addressing modes
  - So, “inc”
    rather than “inc [si+bx-7]”
- The primary reason: portability!

Architectures differ wildly in register design

- x86
  - 8 16-bit registers
  - 8 8-bit registers
  - 4 segment registers
  - Instruction-specific registers
    - Implied operands
    - Restrictions
- ARM
  - 16 32-bit registers
  - 13 are general purpose
    - Technically you can do what you want with any of them
  - 3 have specific purpose

Suppose JVM used registers?

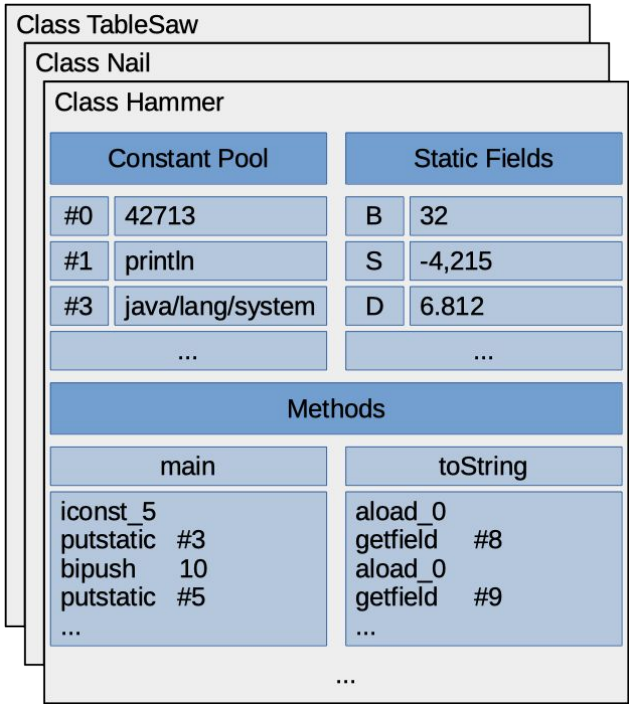
- **Which one architecture?**
- **What if more registers in JVM than architecture? Fewer?**
- **How to handle register sizes?**
- **How to manage special registers?**

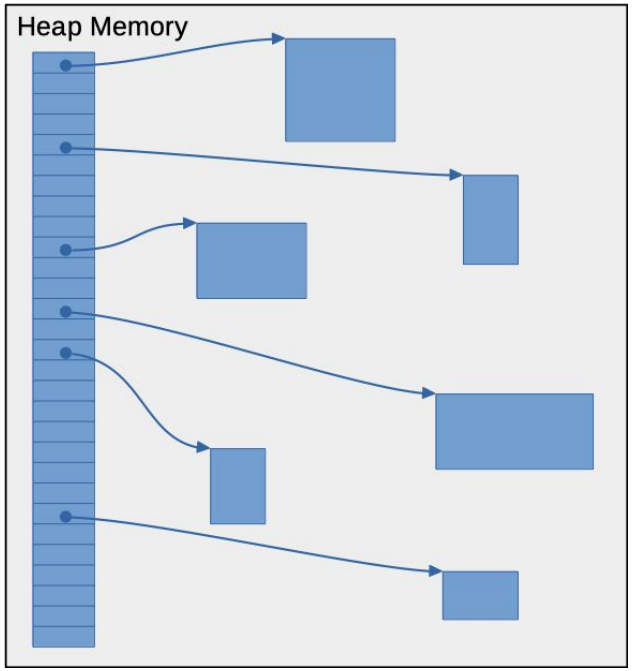
Cost

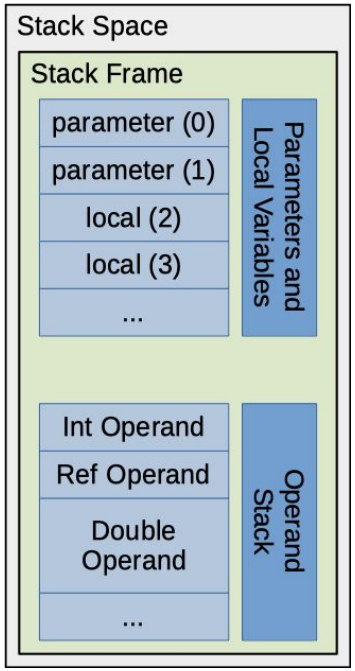
- All operands are in memory
  - Memory is slow
  - Registers are fast
- Registers don’t require caching
  - Register can be temporary storage
  - Fast to access
  - No extra instructions to load/store

```
A = A+B   mov ax,[B]   push A
C = C+B   add [A],ax   push B
D = D+B   add [C],ax   add
          add [D],ax   pop A
                       push C
                       push B
                       add
                       pop C
                       push D
                       push B
                       add
                       pop D
```

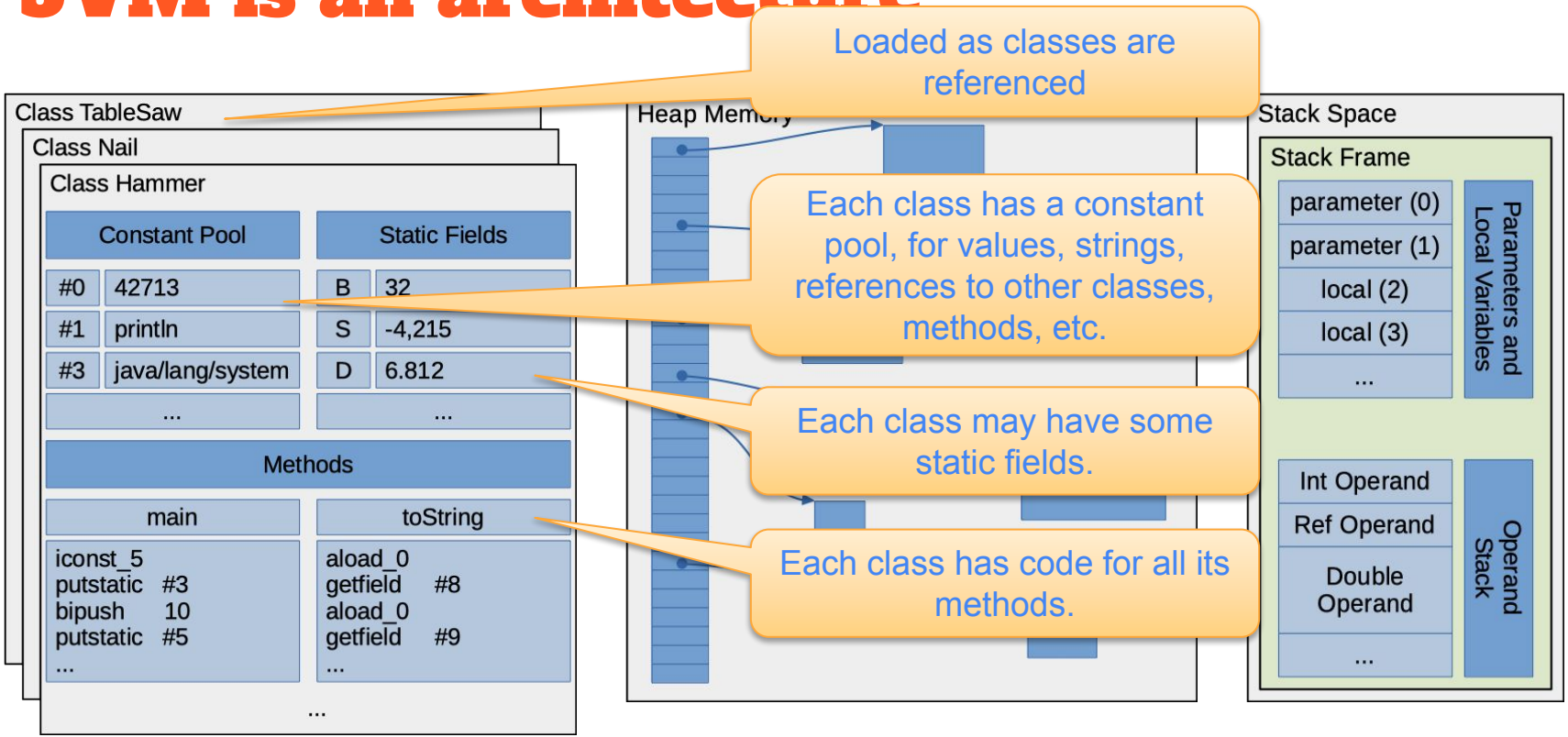
JVM is an architecture



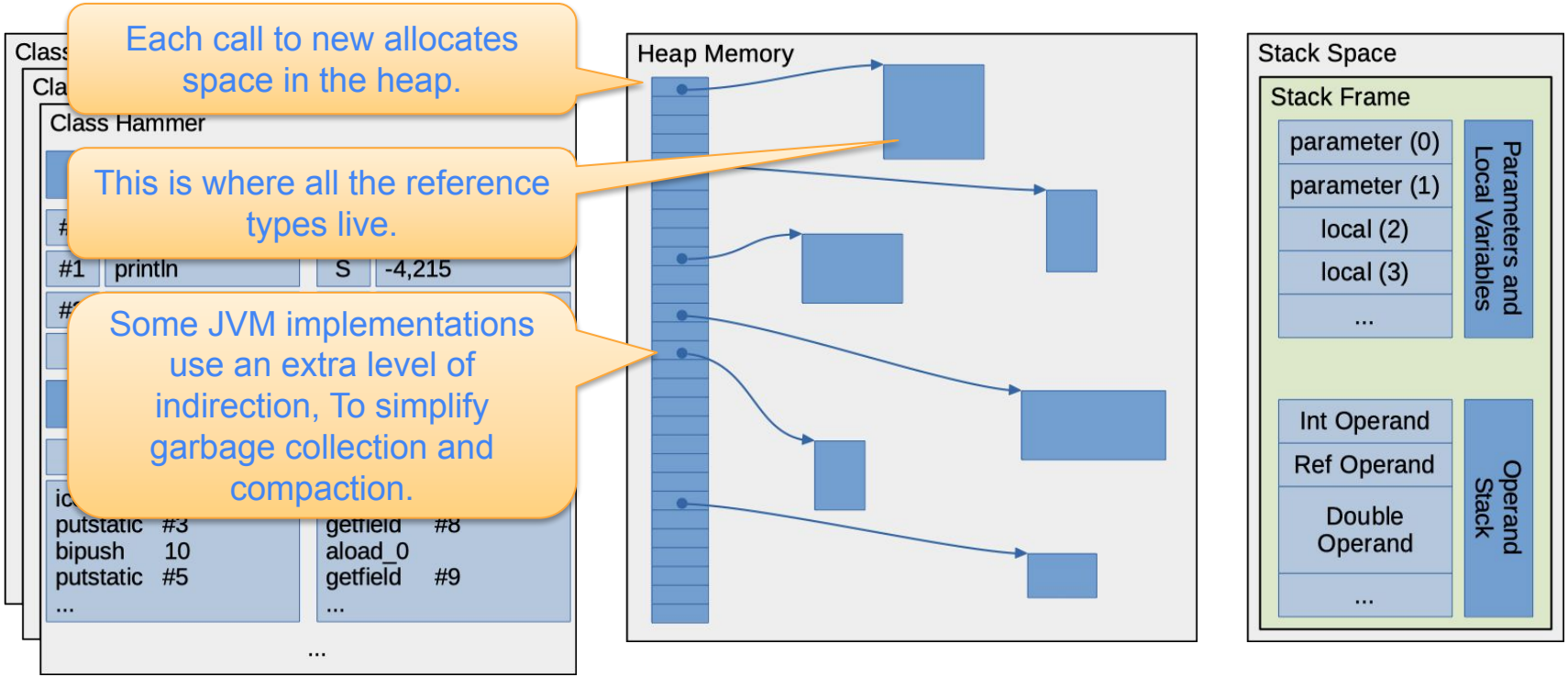




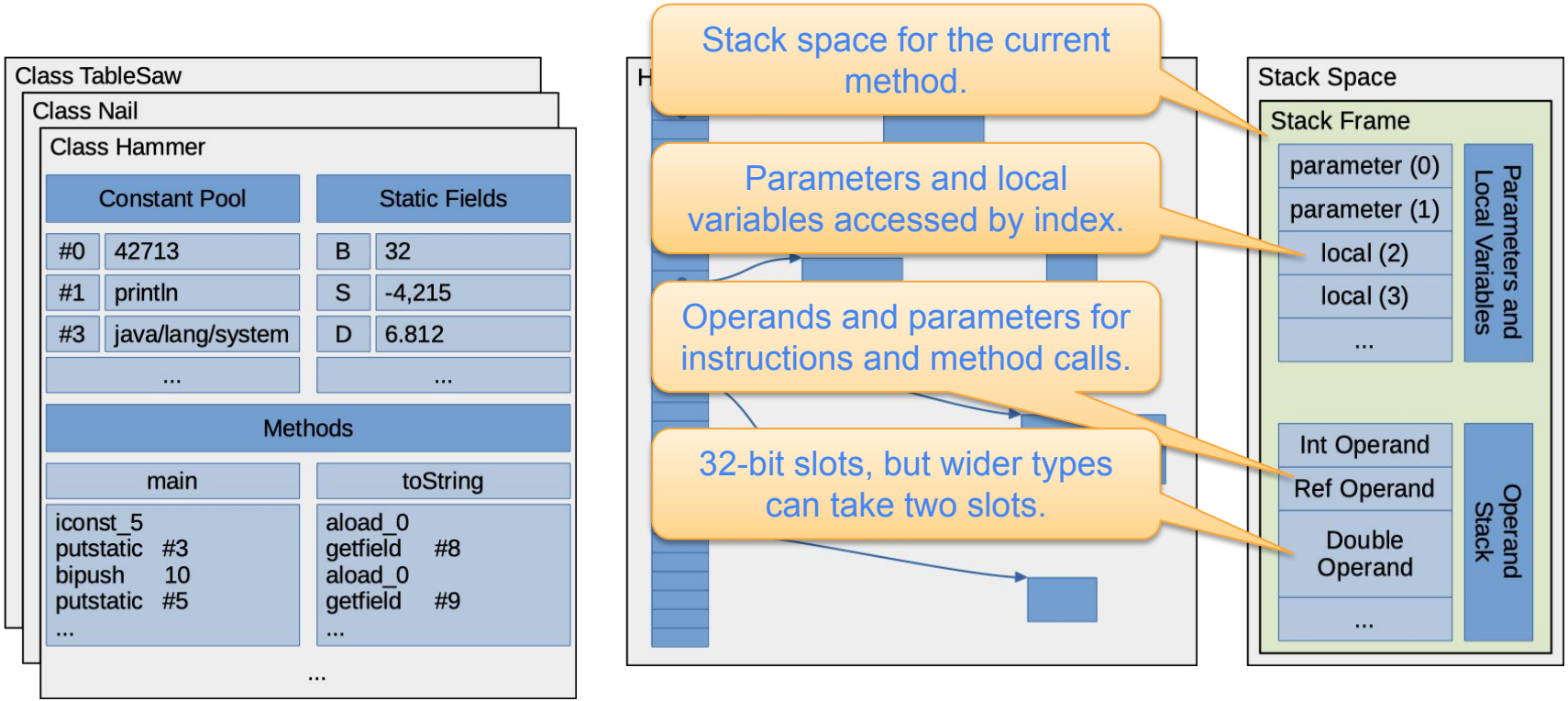
JVM is an architecture



JVM is an architecture

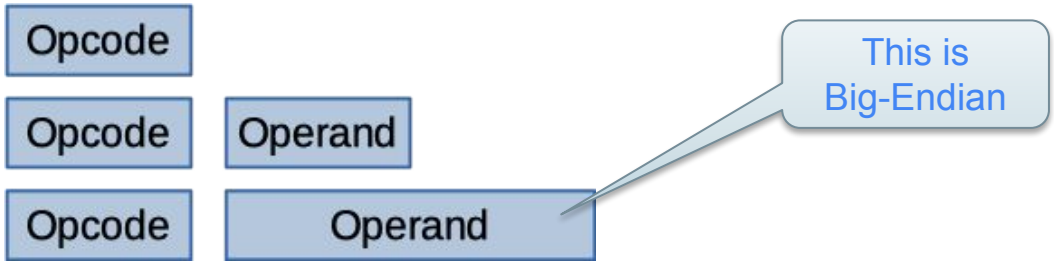


JVM is an architecture



Instructions

- One-byte opcode, up to 256 instructions
  - 202 defined so far, room for some extension
  - We'll learn more than 80 of them … really fast.
- Variable length — most are 1 or 2 bytes
- Shortest instructions to access first 4 variables
- Short instructions to access up to 256 variables, constants, fields
- Longer, general instruction can access 65,536 constants, fields



Meet Some Bytecode

Instruction	Operand (size)	Description
iload_0		Push integer value of local variable 0 onto stack.
iload_1		Push integer value of local variable 1 onto stack.
iload_2		Push integer value of local variable 2 onto stack.
iload_3		Push integer value of local variable 3 onto stack.
iload	*Idx (8)*	Push value of local variable *idx* onto stack.

The ‘i’ means integer.

Loading Literal Values

Instruction	Operand (size)	Description
iconst_m1		Push -1 onto the stack
iconst_0		Push 0 onto the stack.
iconst_1		Push 1 onto the stack.
iconst_2 … iconst_5		Push 2 onto the stack. … Push 5 onto the stack.
bipush	*val (8)*	Push byte *val* as an int
bspush	*val (16)*	Push short *val* as an int
ldc	*idx (8)*	Load the value at index *idx* from the constant pool.

Instructions for Integer Math

Instruction	Operand (size)	Description
iadd		Pop integers B then A, push A + B
isub		Pop integers B then A, push A - B
imul		Pop integers B then A, push A * B
idiv		Pop integers B then A, push A / B
irem		Pop integers B then A, push A % B
iand,ior, ixor		Pop integers B then A, push A & B, A | B or A ^ B
ishl, ishr		Pop integers B then A, push A << B or A >> B

Saving Results

Instruction	Operand (size)	Description
istore_0		Pop integer value and store in local variable 0.
istore_1		Pop integer value and store in local variable 1.
istore_2		Pop integer value and store in local variable 2.
istore_3		Pop integer value and store in local variable 3.
ilstore	*idx*	Pop integer value and store in local variable at *idx*.

Operands for Different Types

Integer	Long	Float	Double
iload_0 .. 3	lload_0 .. 3	fload_0 .. 3	dload_0 .. 3
iload	lload	fload	dload
iconst_m1 .. 5	lconst_0 .. 1	fconst_0 .. 2	dconst_0 .. 1
istore_0 .. 3	lstore_0 .. 3	fstore_0 .. 3	dstore_0 .. 3
istore	lstore	fstore	dstore
iadd, isub …	*ladd, lsub …*	fadd, fsub …	dadd, dsub …

- Where are the instructions for bool, byte or short?
  - Local operands and local variables are really ints.
  - Array elements and fields can be these smaller types
  - .. but they're converted to int when we operate on them.

More Instructions

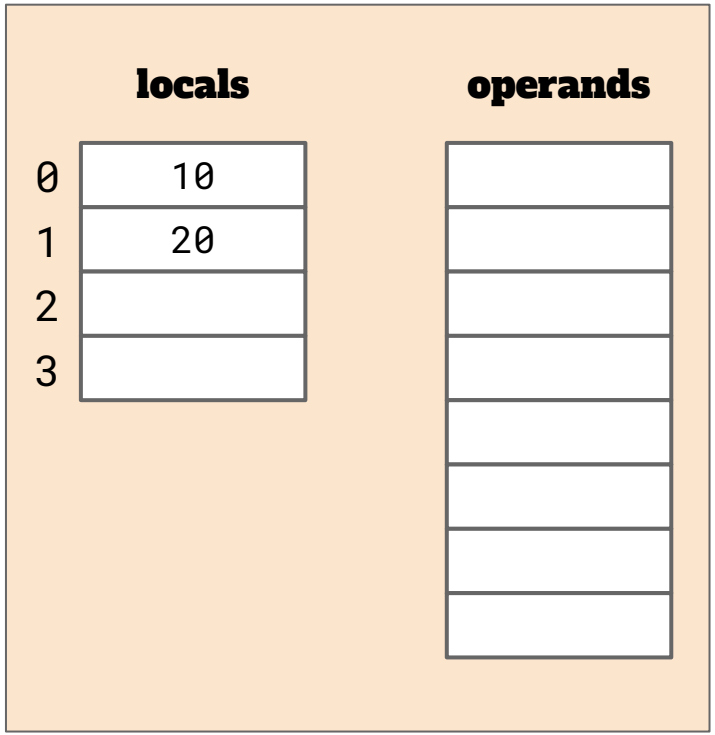
- We’re skipping lots of instructions
  - Stack manipulation
  - Type conversion
  - Field and array access
  - Conditionals, branching and jumping
  - Calling methods and returning

Instruction type	%
Local-variable push	34.5
Local-variable pop	7.0
Memory push	20.2
Memory pop	4.0
Constant push	6.8
Compute	9.2
Branches	7.9
Calls/returns	7.3
Misc stack ops	2.1
New objects	0.4
All others	0.6

Survey of Bytecodes

Note : More pushes than pops

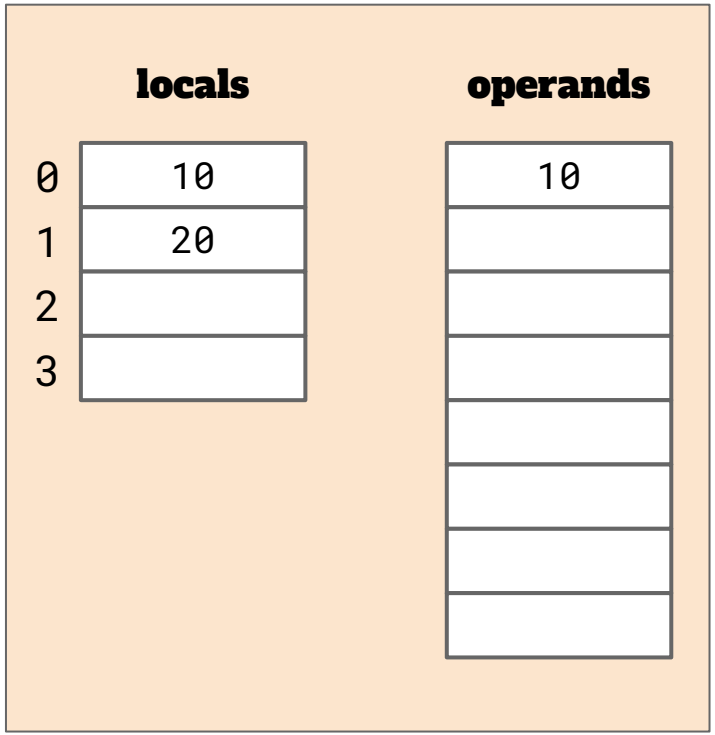
C = A + B
A is local 0
B is local 1
C is local 2



frame

C = A + B
A is local 0
B is local 1
C is local 2

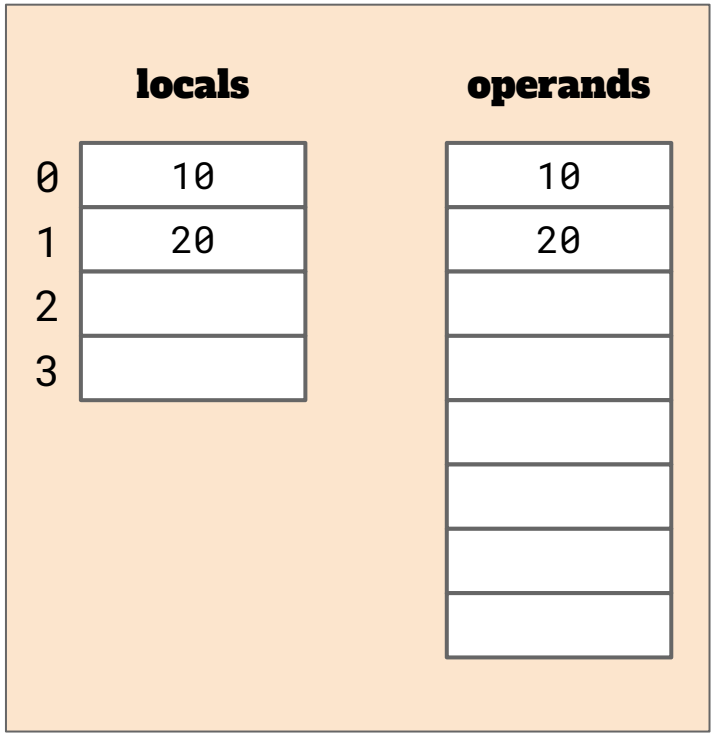
iload	0



frame

C = A + B
A is local 0
B is local 1
C is local 2

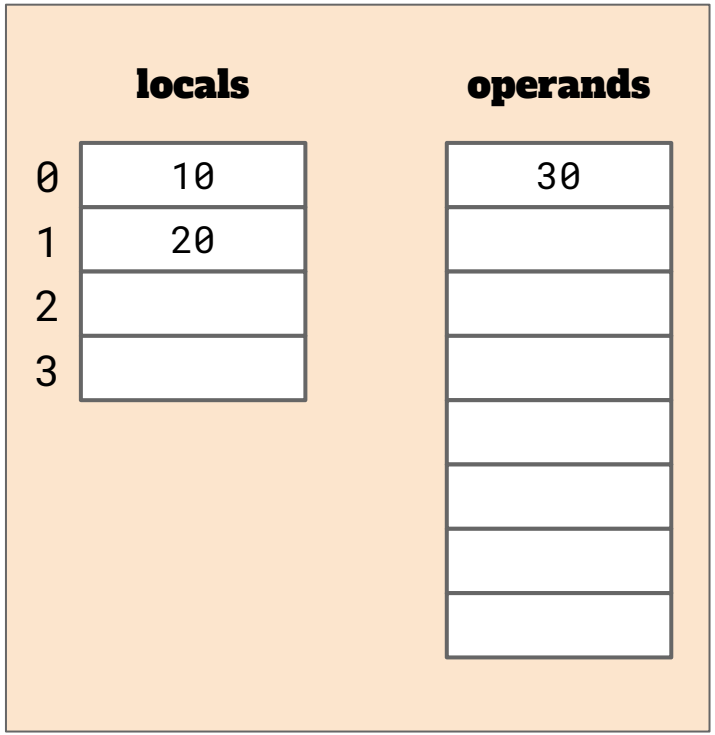
iload	0
iload	1



frame

C = A + B
A is local 0
B is local 1
C is local 2

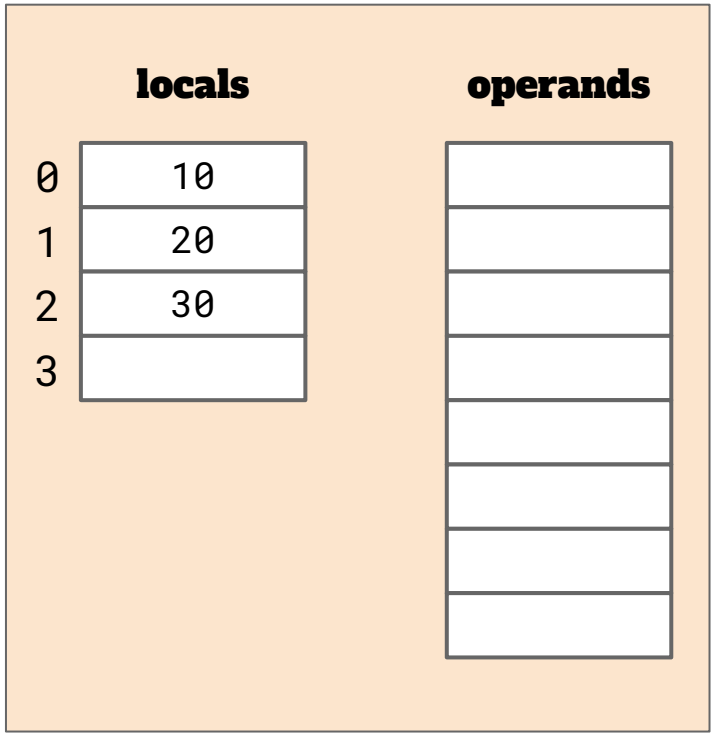
iload	0
iload	1
iadd	



frame

C = A + B
A is local 0
B is local 1
C is local 2

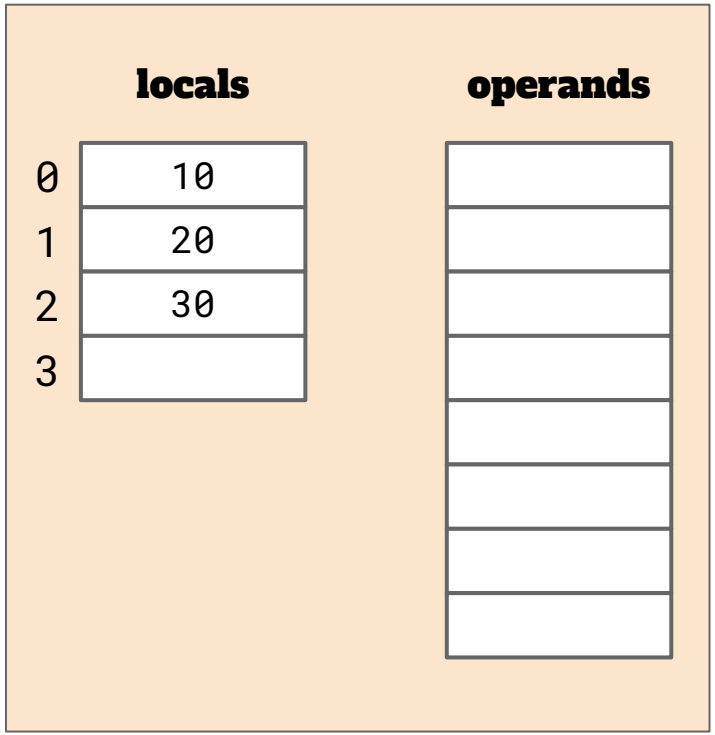
iload	0
iload	1
iadd	
istore	2



frame

C = A + B
A is local 0
B is local 1
C is local 2

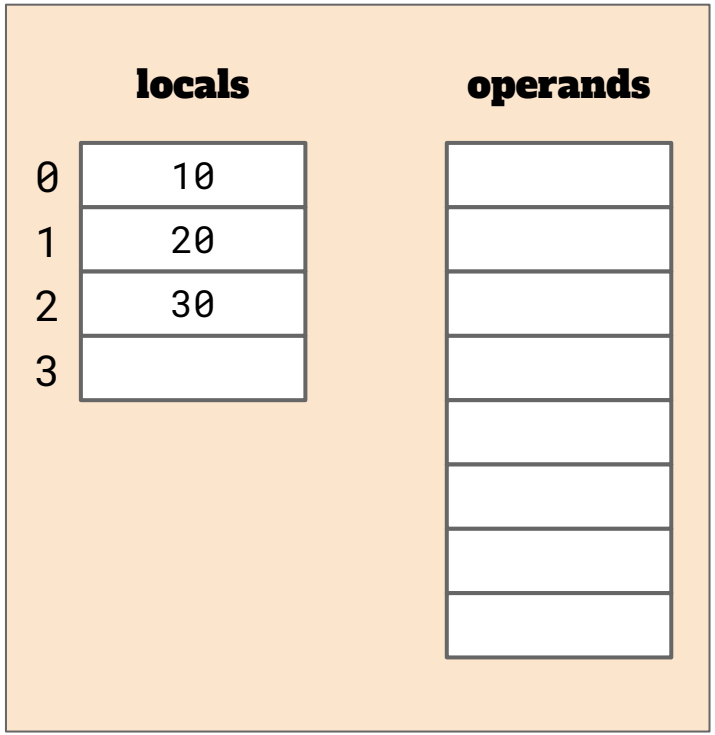
iload	0
iload	1
iadd	
istore	2



iload	0	iload	1	iadd	istore	2

C = A + B
A is local 0
B is local 1
C is local 2

iload_0	
iload_1	
iadd	
istore_2	

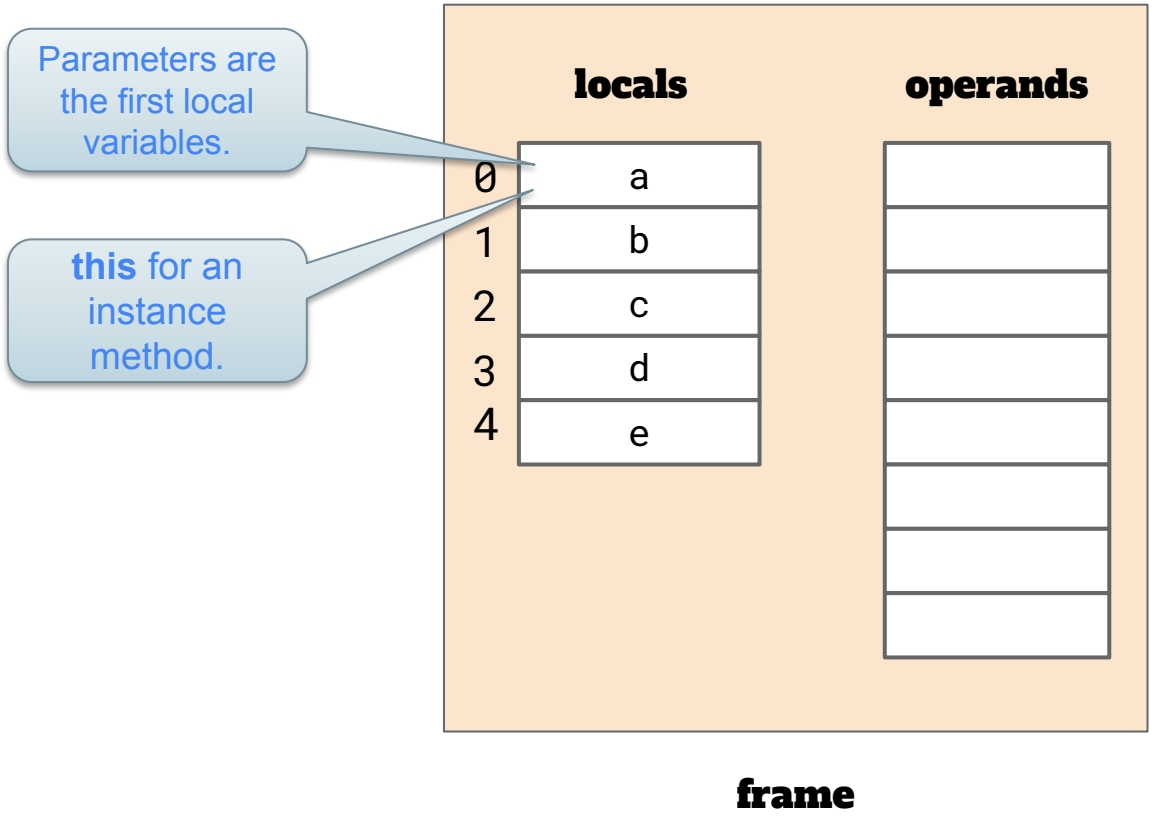


A shorter encoding.

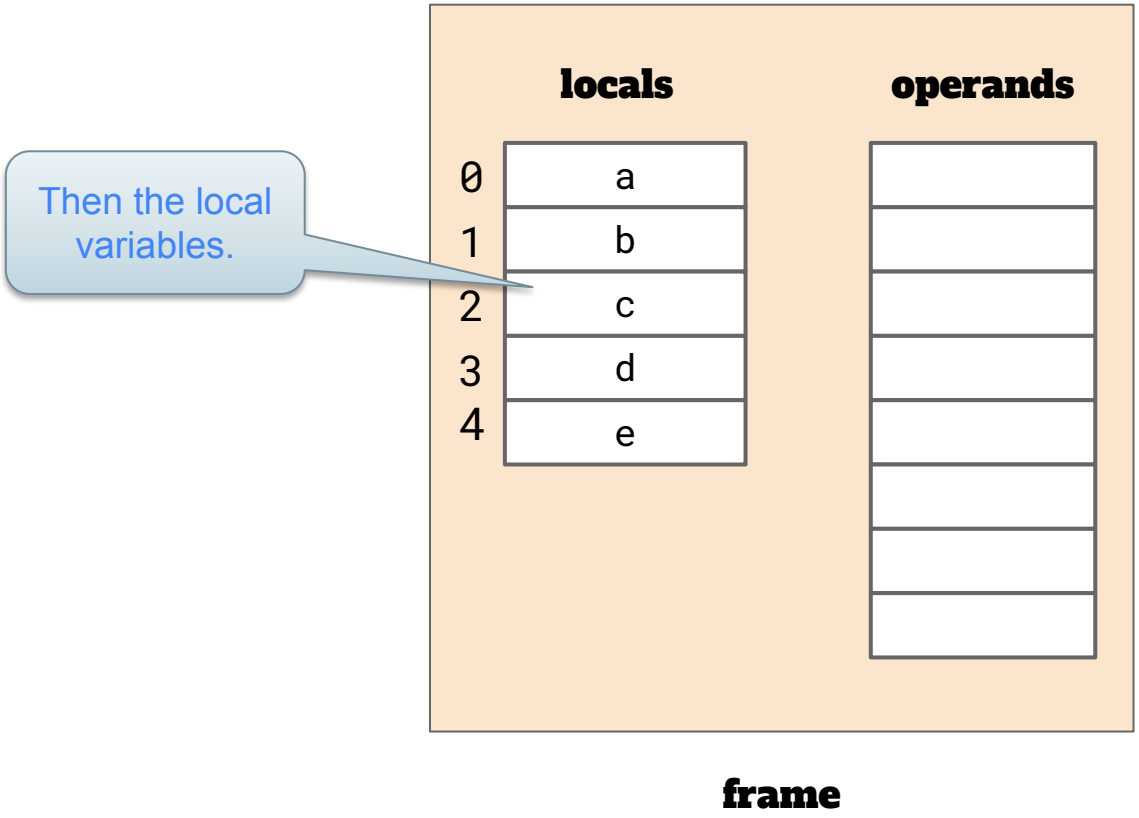
Less important with Just-In-Time Compilation

iload_0	iload_1	iadd	istore_2

```
static void f( int a ) {
  int b = ?;
  int c = ?;
  int d = ?;
  int e;

  e = a + b * c;
}
```

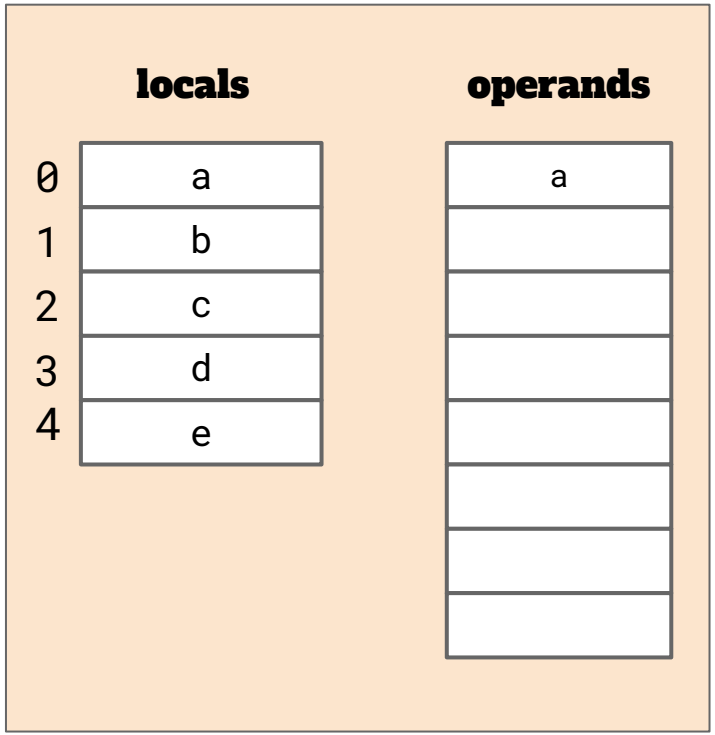


```
static void f( int a ) {
  int b = ?;
  int c = ?;
  int d = ?;
  int e;

  e = a + b * c;
}
```



```
static void f( int a ) {
  int b = ?;
  int c = ?;
  int d = ?;
  int e;

  e = a + b * c;
}
```

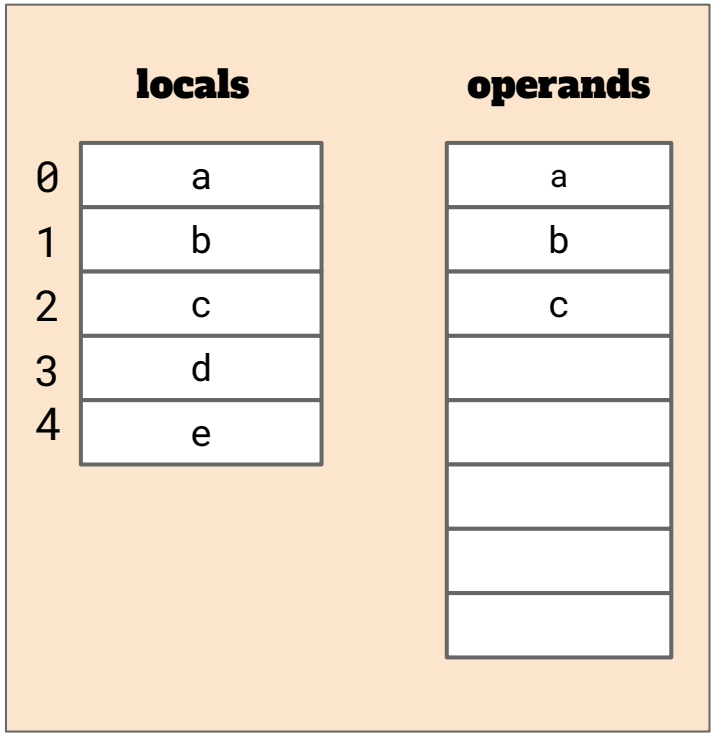
iload_0	



frame

```
static void f( int a ) {
  int b = ?;
  int c = ?;
  int d = ?;
  int e;

  e = a + b * c;
}
```

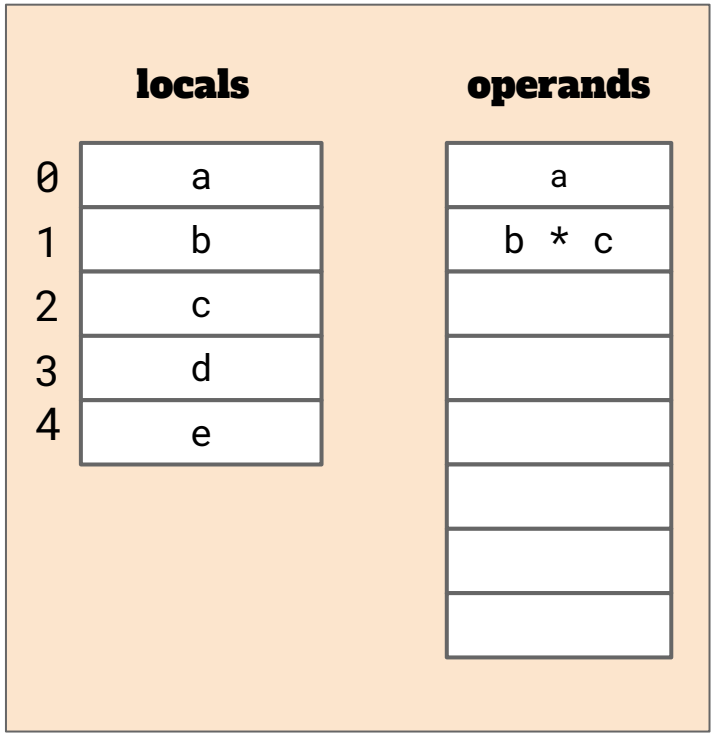
iload_0	
iload_1	
iload_2	



frame

```
static void f( int a ) {
  int b = ?;
  int c = ?;
  int d = ?;
  int e;

  e = a + b * c;
}
```

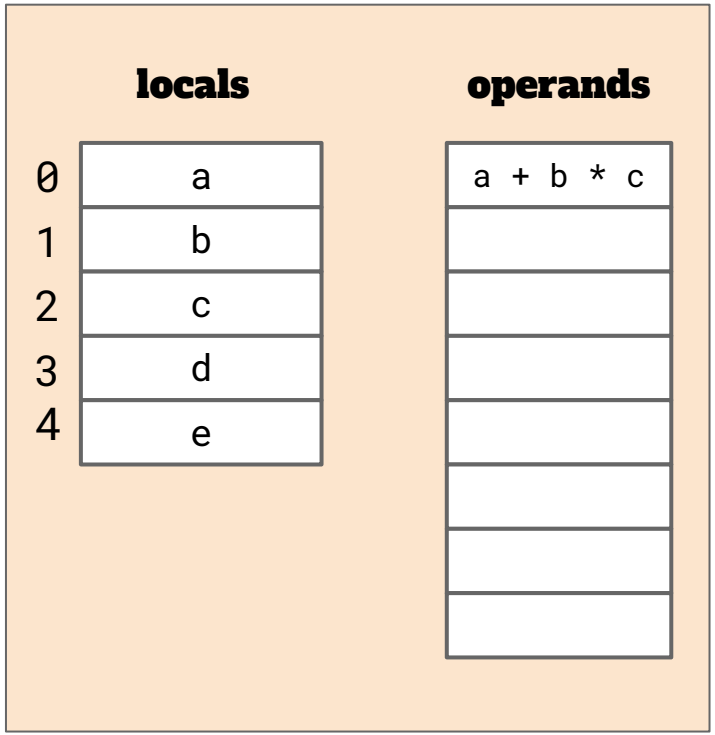
iload_0	
iload_1	
iload_2	
imul	



frame

```
static void f( int a ) {
  int b = ?;
  int c = ?;
  int d = ?;
  int e;

  e = a + b * c;
}
```

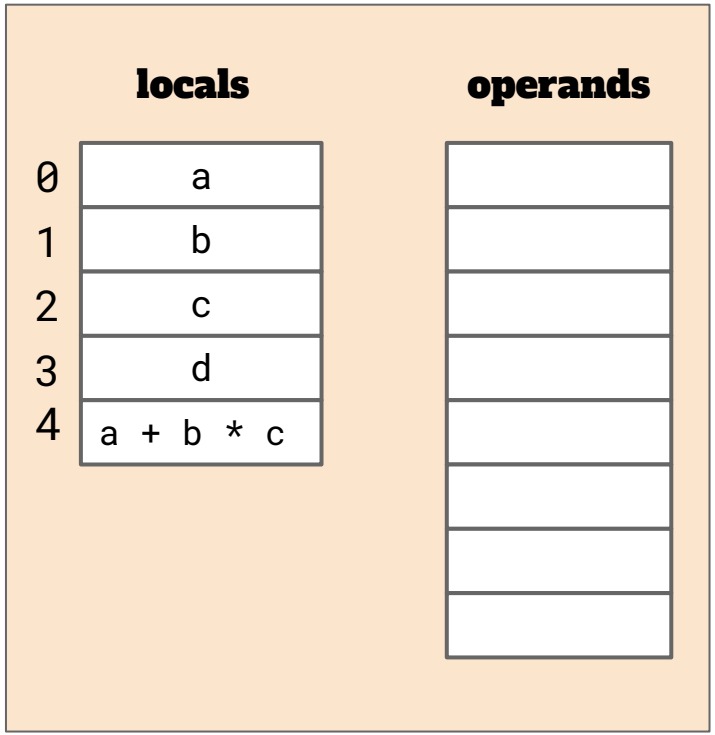
iload_0	
iload_1	
iload_2	
imul	
iadd	



frame

```
static void f( int a ) {
  int b = ?;
  int c = ?;
  int d = ?;
  int e;

  e = a + b * c;
}
```

iload_0	
iload_1	
iload_2	
imul	
iadd	
istore	4



frame

Parameter passing

```
int calc(int a, int b)
```

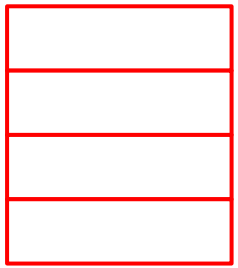
How is this done?

In ASM/C pass parameters on stack

Same in Java

Parameter passing

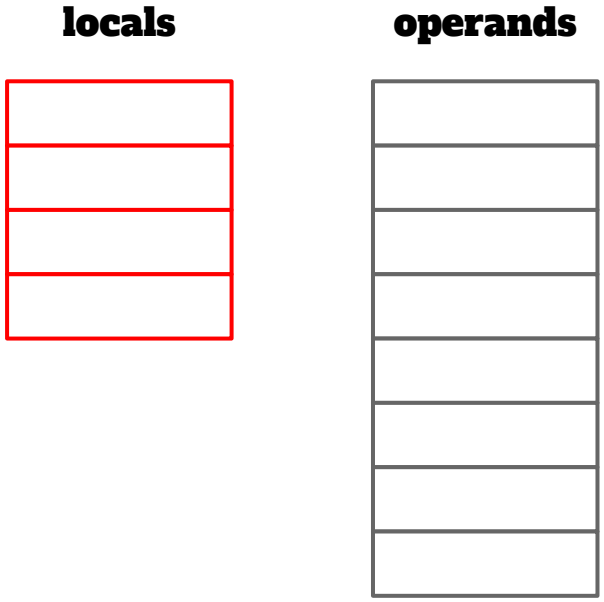
locals

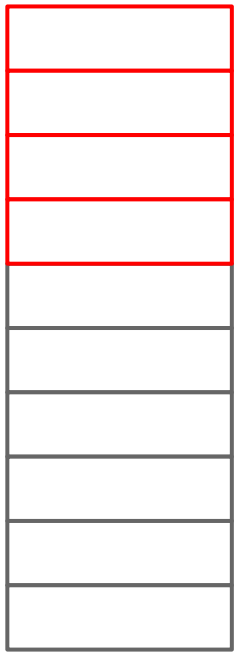


fixed, static size

operands

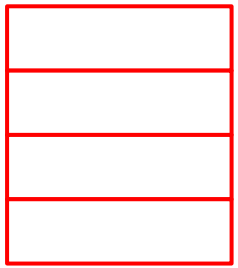
Parameter passing





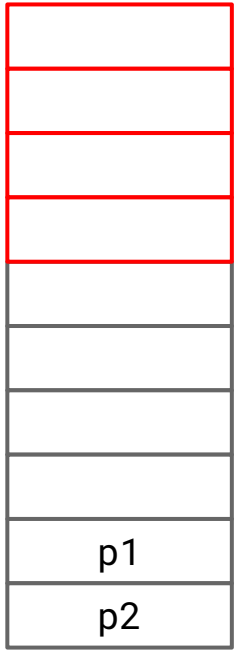
Parameter passing

locals



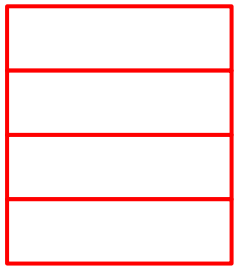
operands

push params on stack



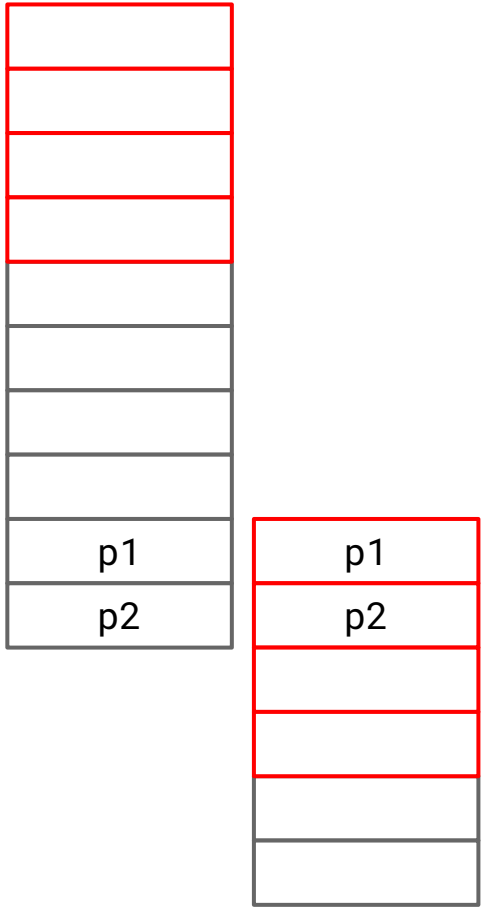
Parameter passing

locals



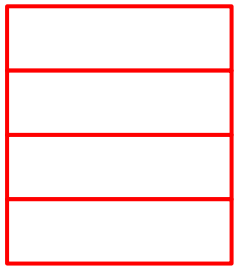
operands

call procedure



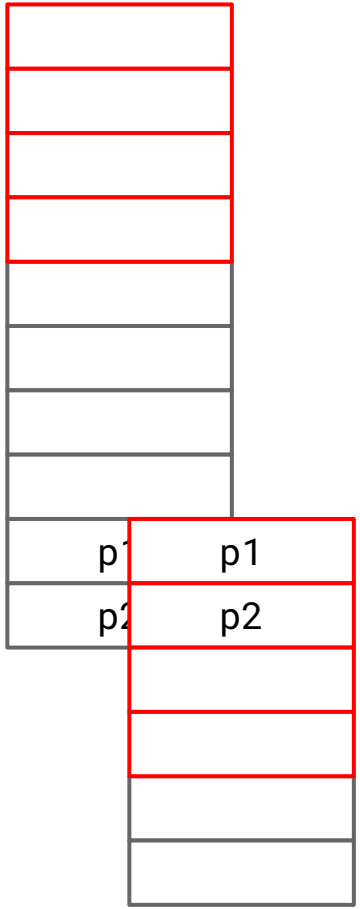
Parameter passing

locals

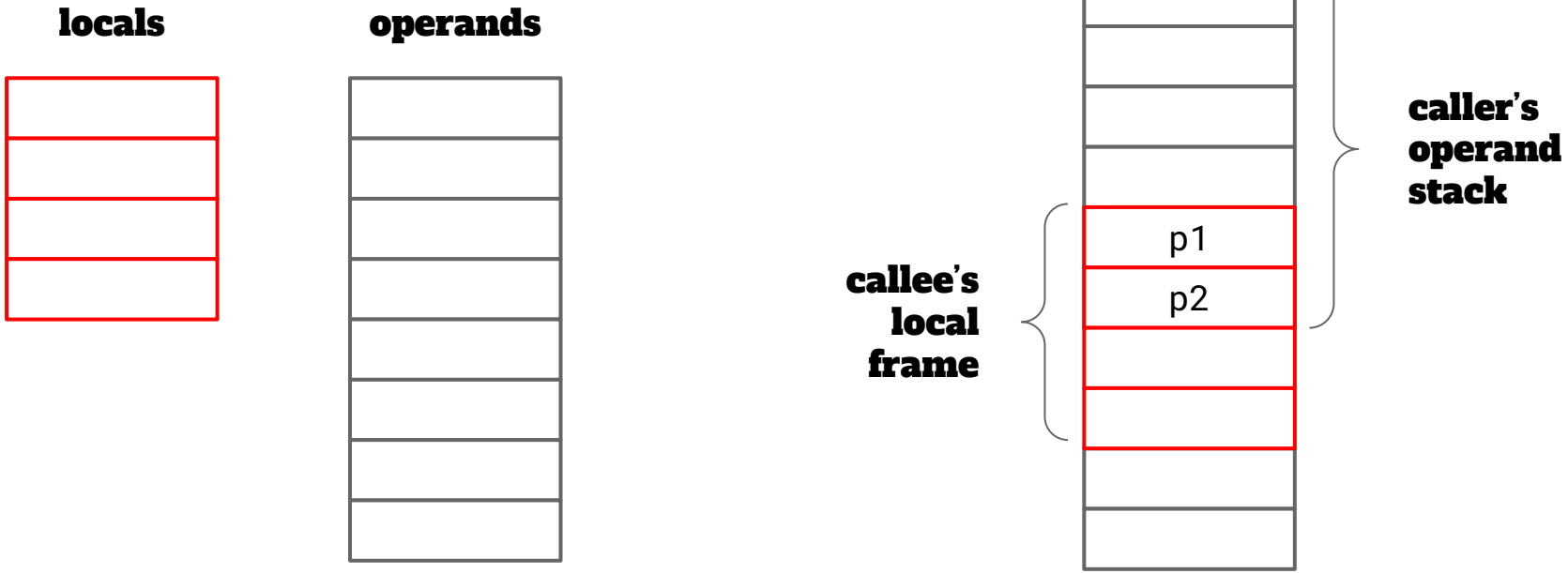


operands

call procedure



Parameter passing



Meet javap

- The java development kit includes **javap**
- … a class file disassembler.
- It can show you what's in a class file.
- With the –c option, it can show bytecode for each method:

```
$ javap -c Test.class
public class Test {
  public static void main(java.lang.String[]);
    Code:
       0: ldc           #2    // String Hello
       2: astore_1
       3: ldc           #3    // String  world.
       5: astore_2
       6: getstatic     #4    // Field …
```

That's java bytecode right there.

Performance

- Java does not run *natively*
  - All instructions are *interpreted* by JVM
- Executing an instruction
  - Fetch: read next instruction
  - Decode: determine what to do
  - Execute: do it
- For assembly, that's one machine instruction.
- How long in JVM?
  - A single byte code instruction becomes
  - 10 to 100s of machine instructions (e.g. new instruction)

Executing Java bytecode

(1) Software interpreter

- Interpret each instruction
- Perform required operations
  - Change the state of the JVM
- Short loop to execute JVM instructions one after another
  - Repeatedly fetch & interpret the next instruction
  - Same process over and over
  - … but operands and JVM state will vary from instruction to instruction.

Speedup techniques

- Cache code
  - Saves disk/file IO

Executing Java bytecode

(2) Compile to native code

- Instead of emitting bytecode
  - Emit native ASM/machine code
- Rarely done
- Why
  - Defeats portability
  - Lots of other aspects of the JVM (e.g., garbage collection) that still need to be provided.
  - Difficult to optimize

Speedup techniques

- Native code for individual JVM instruction
  - fetch/decode/execute done by the processor
  - ... not done via software.
- Still has overhead
  - Stack operations (rather than directly using registers)

Executing Java bytecode

(3) JIT (just in time) compiler

- Read class files for each new class
  - Including bytecode for each method.
- Individual methods can be compiled to sequence of native instructions.
  - If they are expensive or executed frequently.

Speedup techniques

- Native methods can run faster
- Not all methods need to be compiled to native instructions
- Can even support different implementations of the same method.
- For example:
  - double(x) ⇒ add ax,ax
  - double(2) ⇒ mov ax,4

Executing Java bytecode

(4) JVM hardware chip

- Build hardware to run bytecode natively
- Architecture is designed specifically for bytecode
  - Faster than general-purpose machine

Speedup techniques

- Very fast
  - For JVM
  - Bytecode is inefficient compared to traditional
  - Eg, no registers
- Disadvantages
  - Special-purpose hardware can be expensive.