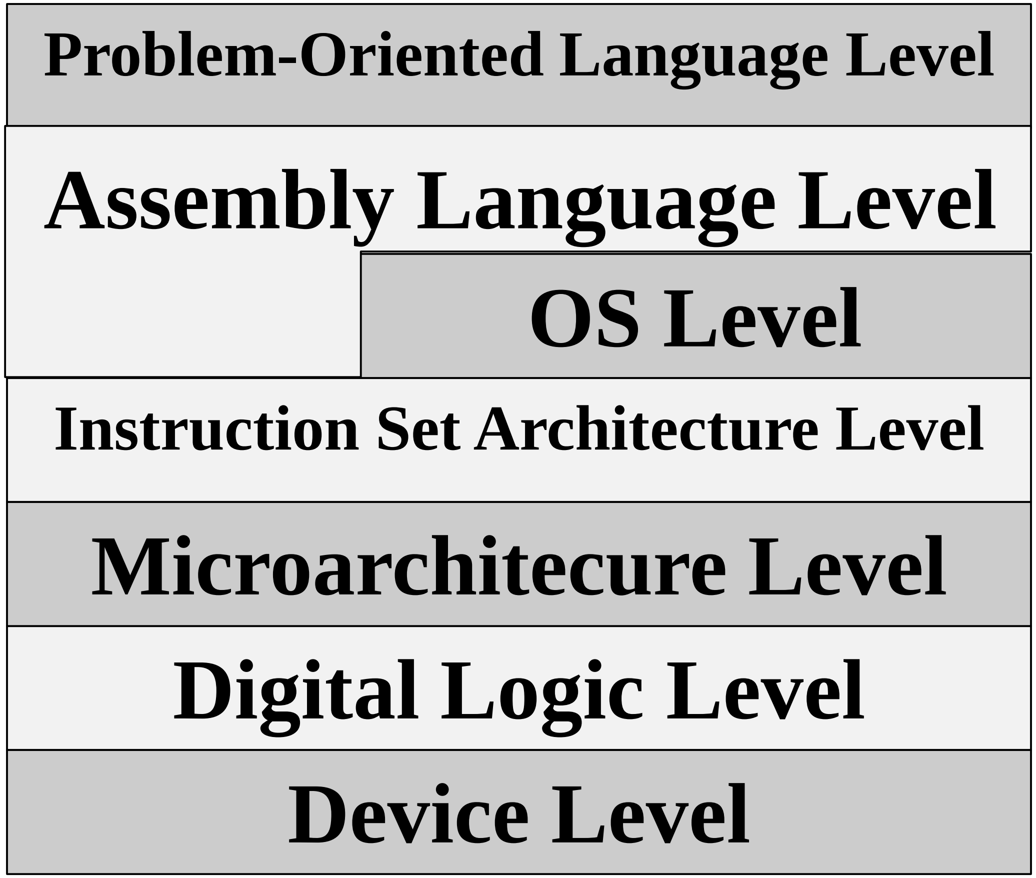
Microarchitecture
Design & Operation

Suggested Reading: Chapter 4 from Tanenbaum, Structured Computer Organization, 5/6ed, Pearson

Layered Computer Organization



Problem-Oriented Language Level

Assembly Language Level

OS Level

Instruction Set Architecture Level

Microarchitecure Level

Digital Logic Level

Device Level

IJVM

INSTRUCTION SET ARCHITECTURE

Objectives

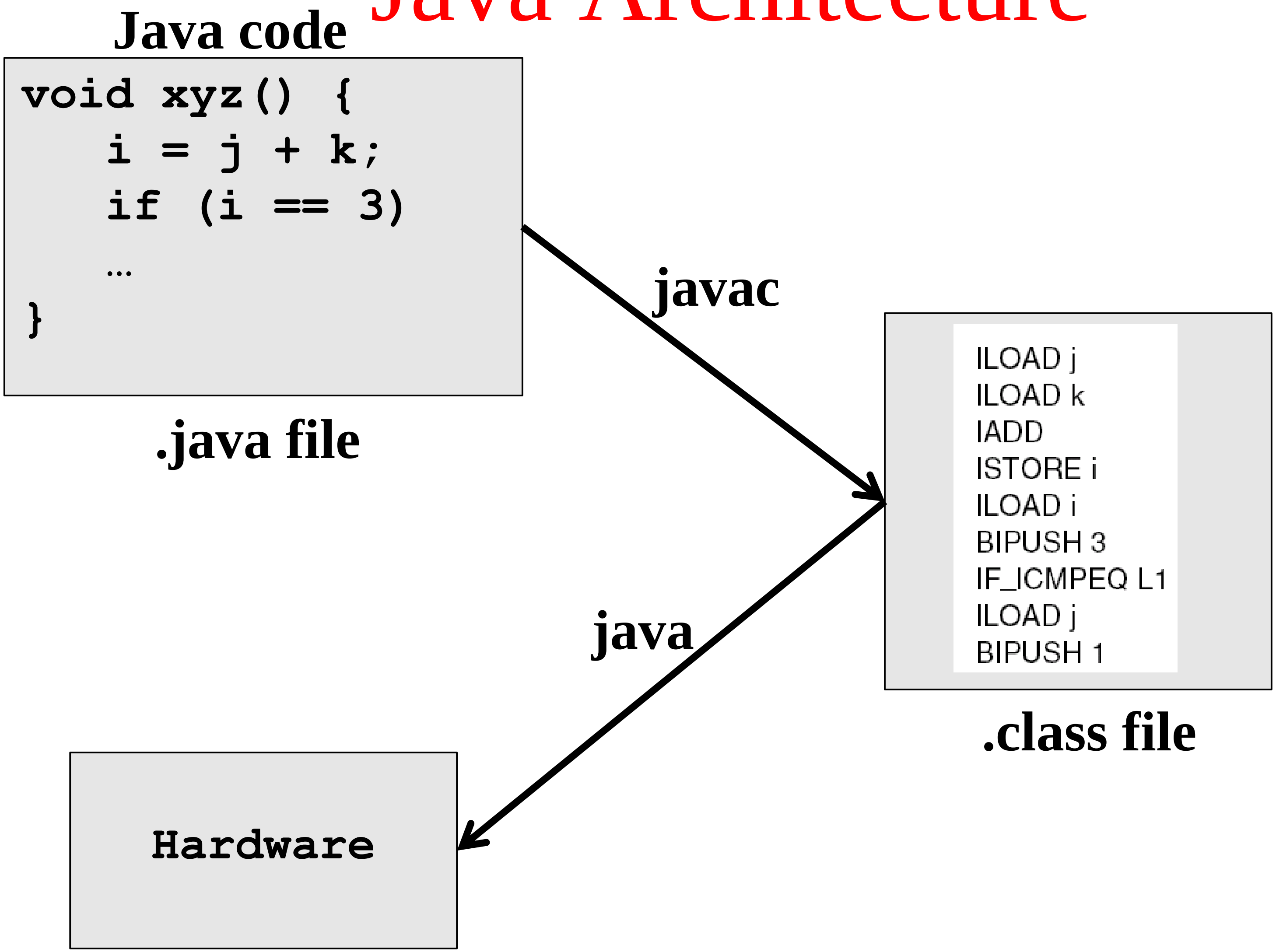
By the end of this section, you will be able to:

1. List and work with the IJVM ISA instructions
2. Translate a subset of Java programs to IJVM ISA
3. Understand stack machines

JVM

- **javac**: Java compiler
  - Compiles HL Java to *Bytecode*
  - *Bytecode = Java Assembly (JAS) Language*
- **JVM:** Java Virtual Machine
  - Invoked by the **java** command
  - An interpreter of Java Bytecode

Java Architecture



Java code

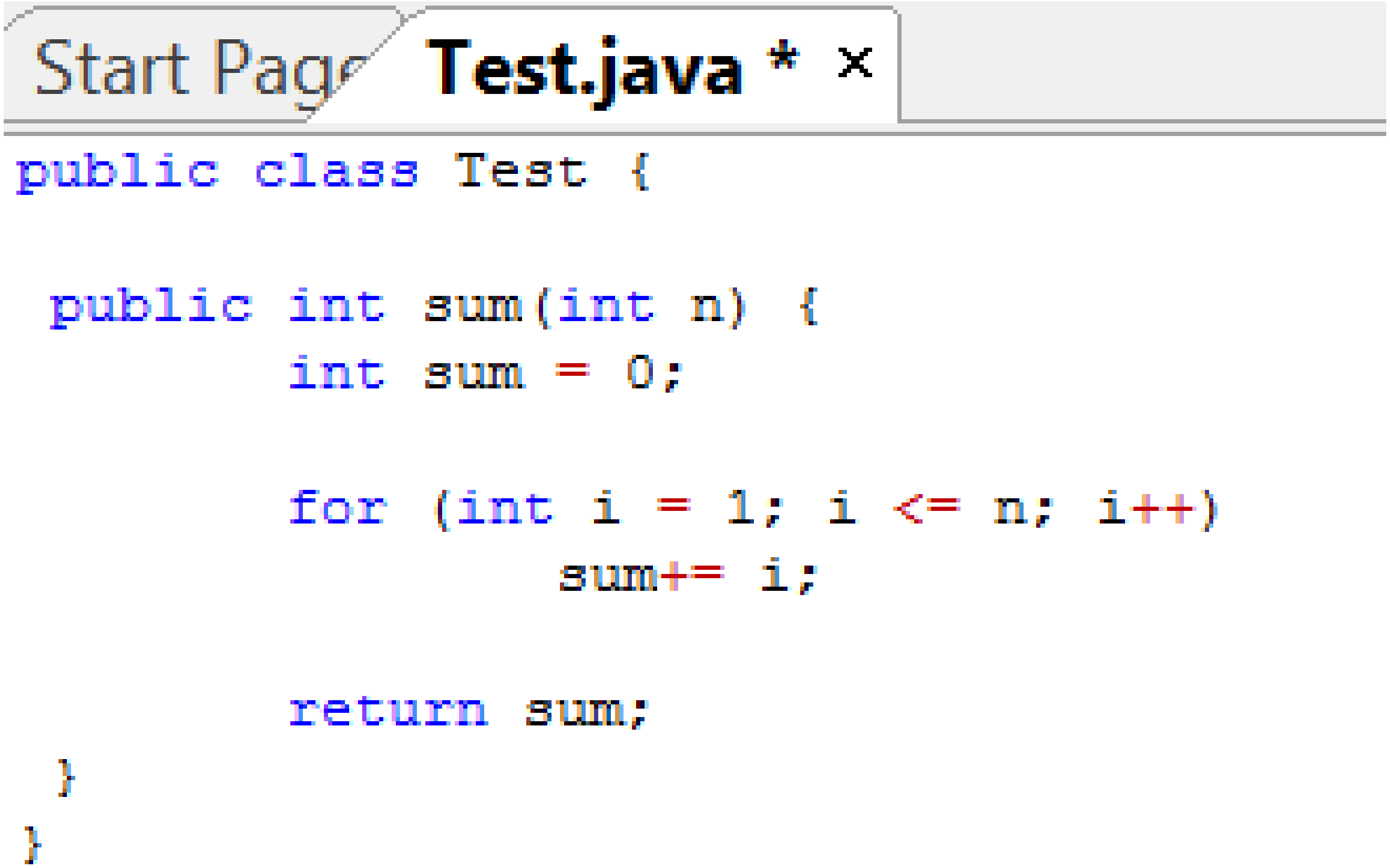
```
void xyz() {
    i = j + k;
    if (i == 3)
    ...
}
```

.java file

```
ILOAD j
ILOAD k
IADD
ISTORE i
ILOAD i
BIPUSH 3
IF_ICMPEQ L1
ILOAD j
BIPUSH 1
```

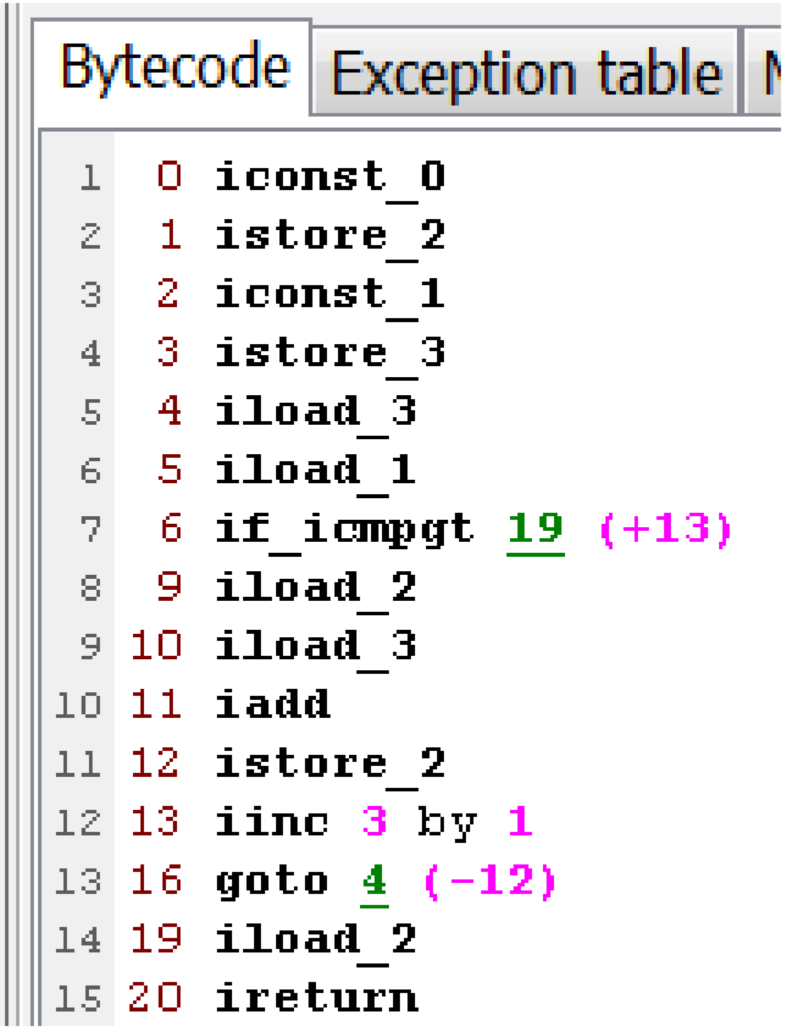
.class file

Example .java code



```
public class Test {

 public int sum(int n) {
        int sum = 0;

        for (int i = 1; i <= n; i++)
                sum+= i;

        return sum;
 }
}
```

Example .class file



```
 0 iconst_0
 1 istore_2
 2 iconst_1
 3 istore_3
 4 iload_3
 5 iload_1
 6 if_icmpgt 19 (+13)
 9 iload_2
10 iload_3
11 iadd
12 istore_2
13 iinc 3 by 1
16 goto 4 (-12)
19 iload_2
20 ireturn
```

IJVM

- **IJVM:** subset of JVM that deals with integers
    - No floating point instructions

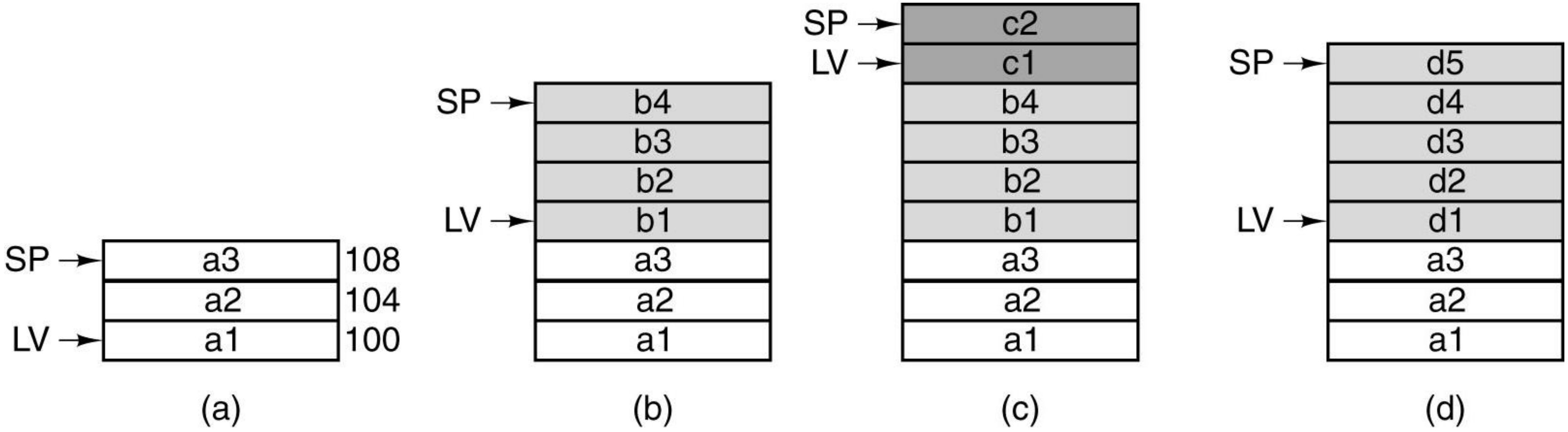
- All integer JAS opcodes are 1-byte long
    - Simpler to deal with

Stacks

- JVM is a stack machine

- **Stacks** are used to push local procedure variables
  - **Local variable frame**: data structure between LV and SP

Local Variable Frame



Use of a stack for storing local variables.

a) While *A* is active.

b) After *A* calls *B*.

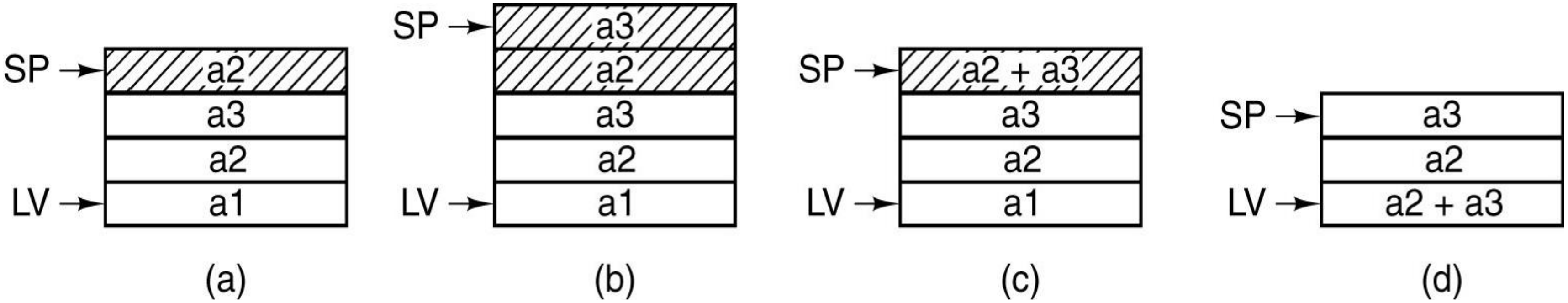
c) After *B* calls *C*.

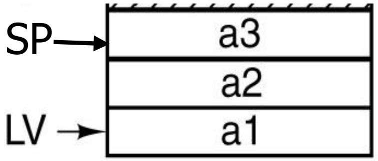
d) After *C* and *B* return and *A* calls *D*.

Operand Stacks

- JVM is an **operand stack** machine
- Stacks are used to push operands during the computation of arithmetic expressions

Operand Stack

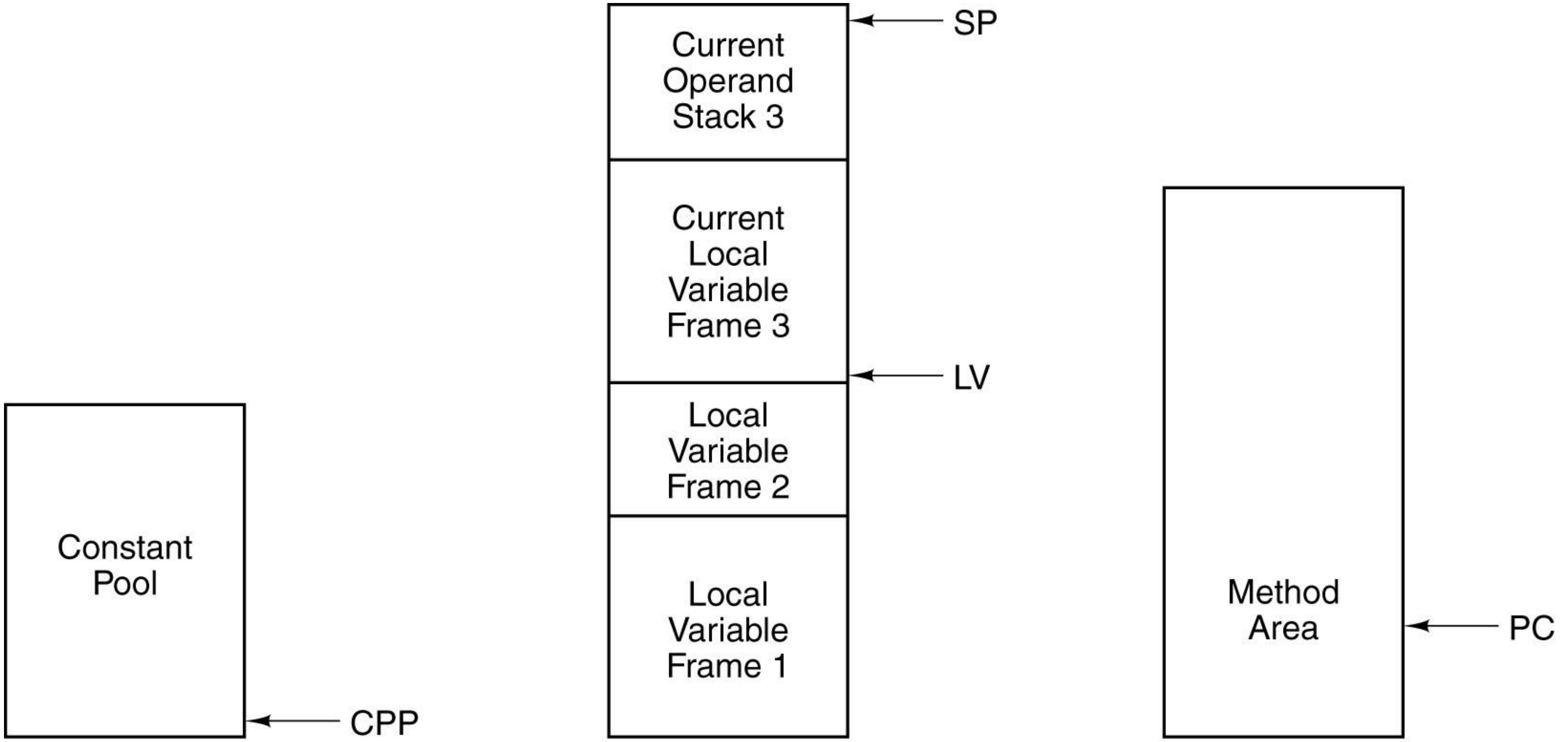




Use of an operand stack for doing an arithmetic computation

(a1 = a2+a3)

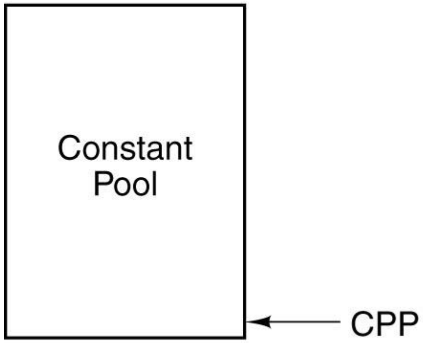
The IJVM Memory Model



The various parts of the IJVM memory.

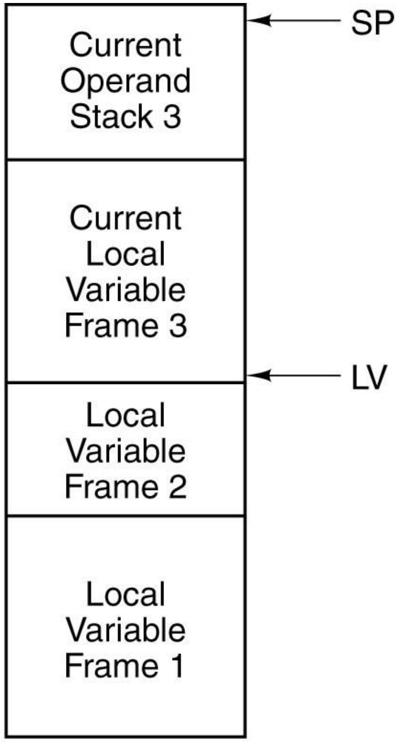
Constant Pool

- Constants, strings, & pointers to other areas of memory
- Does not change after loading (cannot be written by IJVM program)
- CPP contains the address of the first word



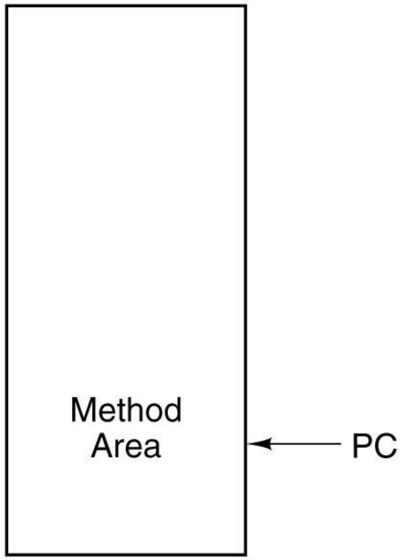
Local Variable Frame / Operand Stack

- allocated to variables stored for the lifetime of a procedure
- at the beginning of the frame: parameters
- then, local variables
- then operand stack
    - not true in general
- LV points to the start of LVF
- TOS = word at top of stack



Method Area

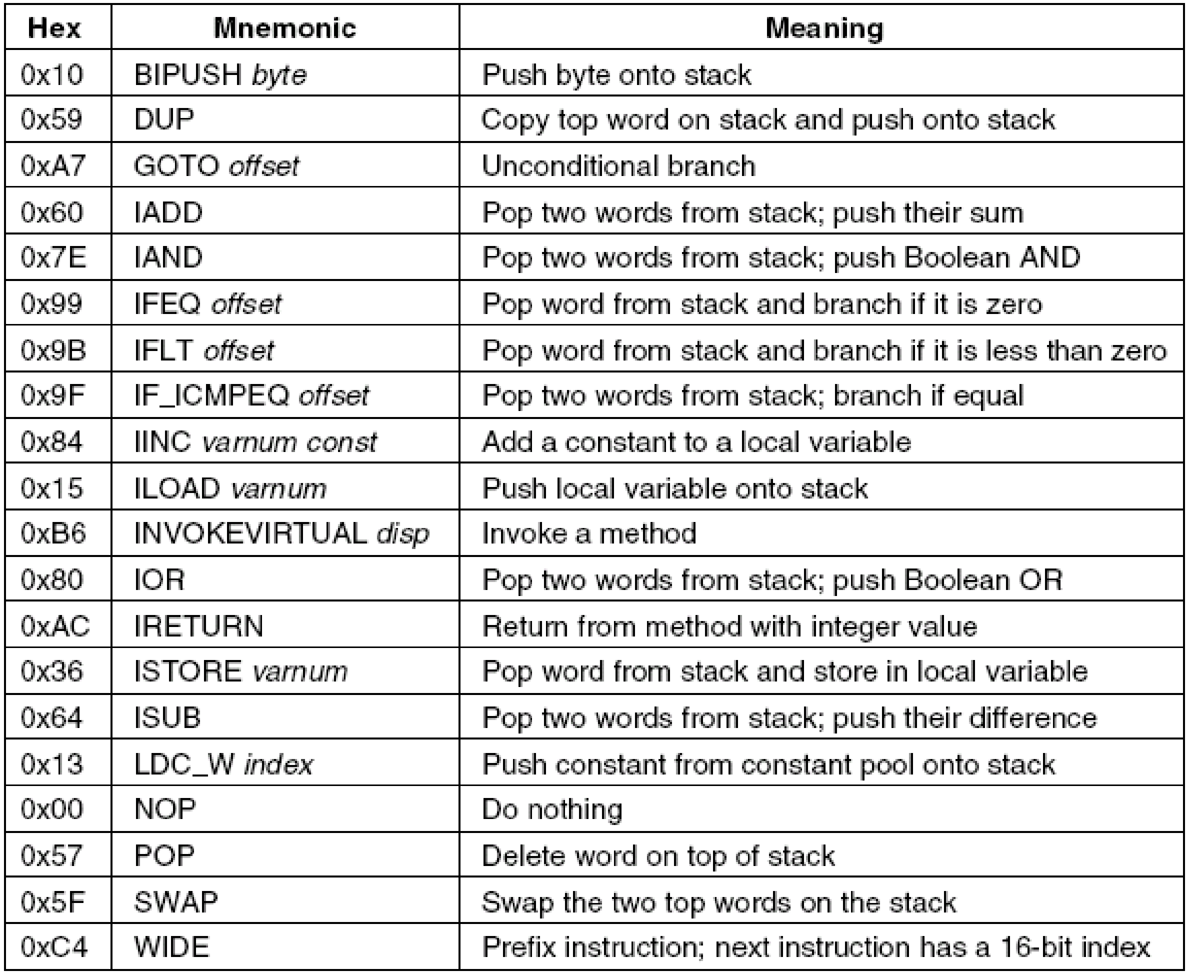
- Program text
- PC points to the instruction to be fetched next



Offsets

- CPP, LV, & SP : word pointers
  - Offset is a word
  - LV+1: second word in LFV
  - SP: word on top of stack, SP – 1, next to top word
- PC: byte address
  - Offset is a byte
  - PC+1: next **byte** is to be fetched

The IJVM Instruction Set

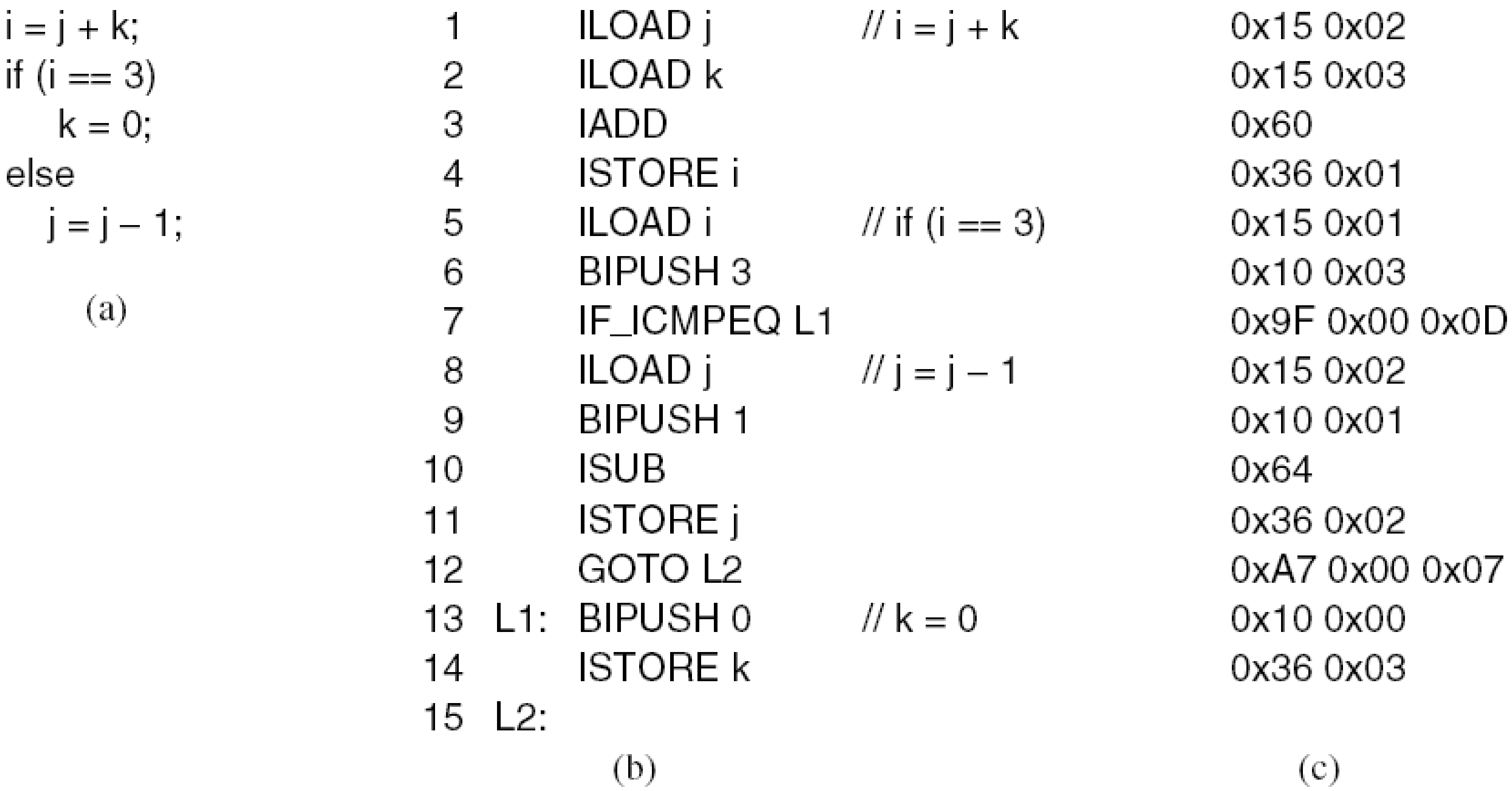


Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

The IJVM instruction set. The operands *byte*, *const*, and *varnum* are 1 byte. The operands *disp*, *index*, and *offset* are 2 bytes.

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index

Compiling Java to IJVM



```
i = j + k;
if (i == 3)
    k = 0;
else
   j = j – 1;
```

(a)

```
        ILOAD j         // i = j + k          0x15 0x02
        ILOAD k                               0x15 0x03
        IADD                                  0x60
        ISTORE i                              0x36 0x01
        ILOAD i         // if (i == 3)        0x15 0x01
        BIPUSH 3                              0x10 0x03
        IF_ICMPEQ L1                          0x9F 0x00 0x0D
        ILOAD j         // j = j – 1          0x15 0x02
        BIPUSH 1                              0x10 0x01
        ISUB                                  0x64
        ISTORE j                              0x36 0x02
        GOTO L2                               0xA7 0x00 0x07
   L1:  BIPUSH 0        // k = 0              0x10 0x00
        ISTORE k                              0x36 0x03
   L2:
```

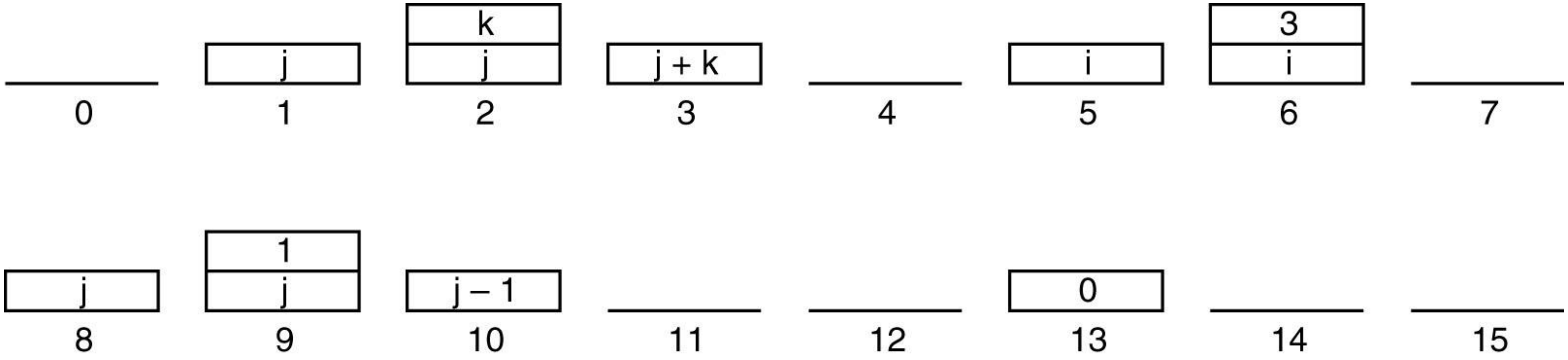
(b) (c)

a) A Java fragment.

b) The corresponding Java assembly language.

c) The IJVM program in hexadecimal.

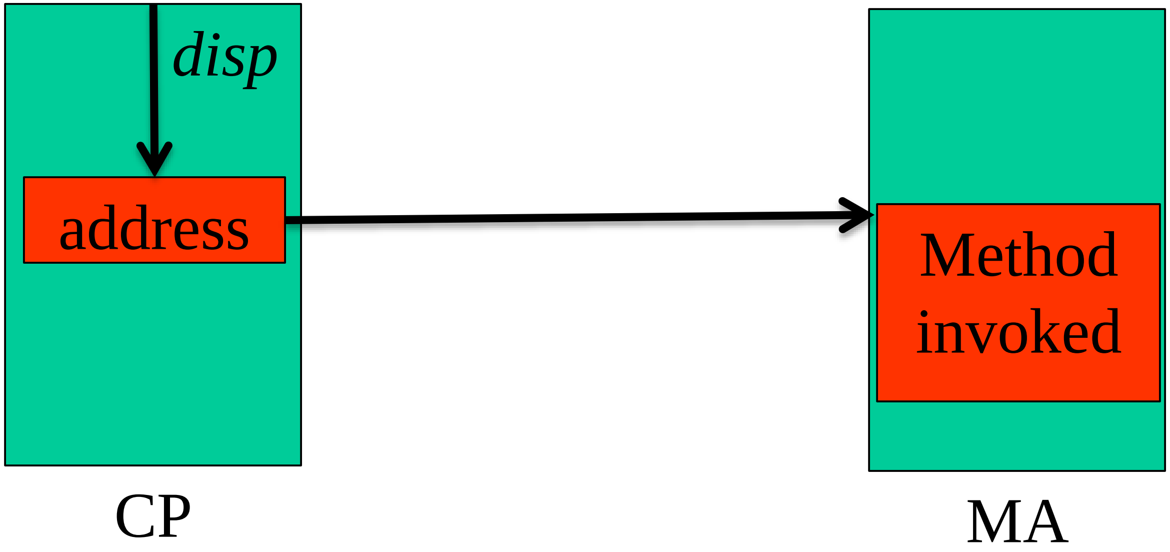
Stack When Executing Bytecode



The stack after each instruction of Fig. 4-14(b).

IVOKEVIRTUAL *disp*

- Invokes another method
- *disp* (16 bit) = position in constant pool that contains the address in method area where method starts

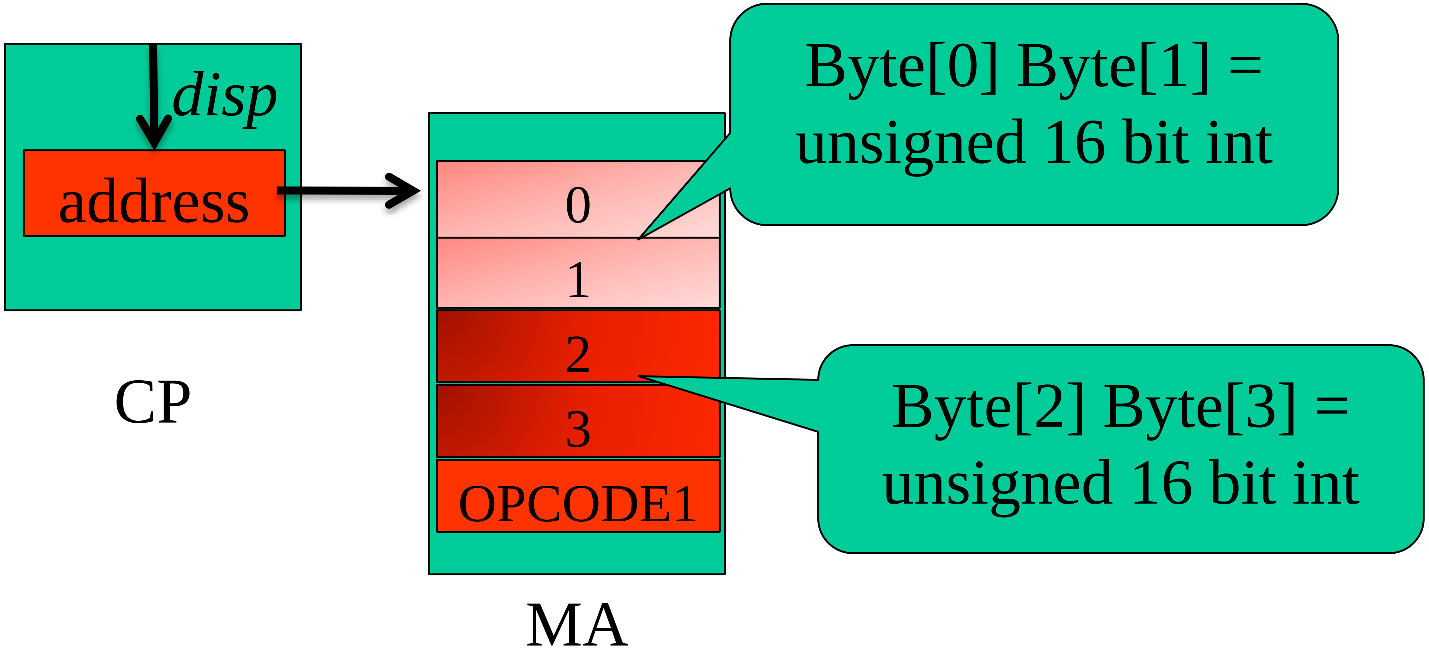


IVOKEVIRTUAL *disp*

- SIMPLIFIED: cannot call methods except in same object
  - *No Object-Orientation!*
- Caller:
  - Pushes OBJREF being called onto stack
    - Not needed for IJVM
  - Pushes method parameters
    - First parameter is param 1

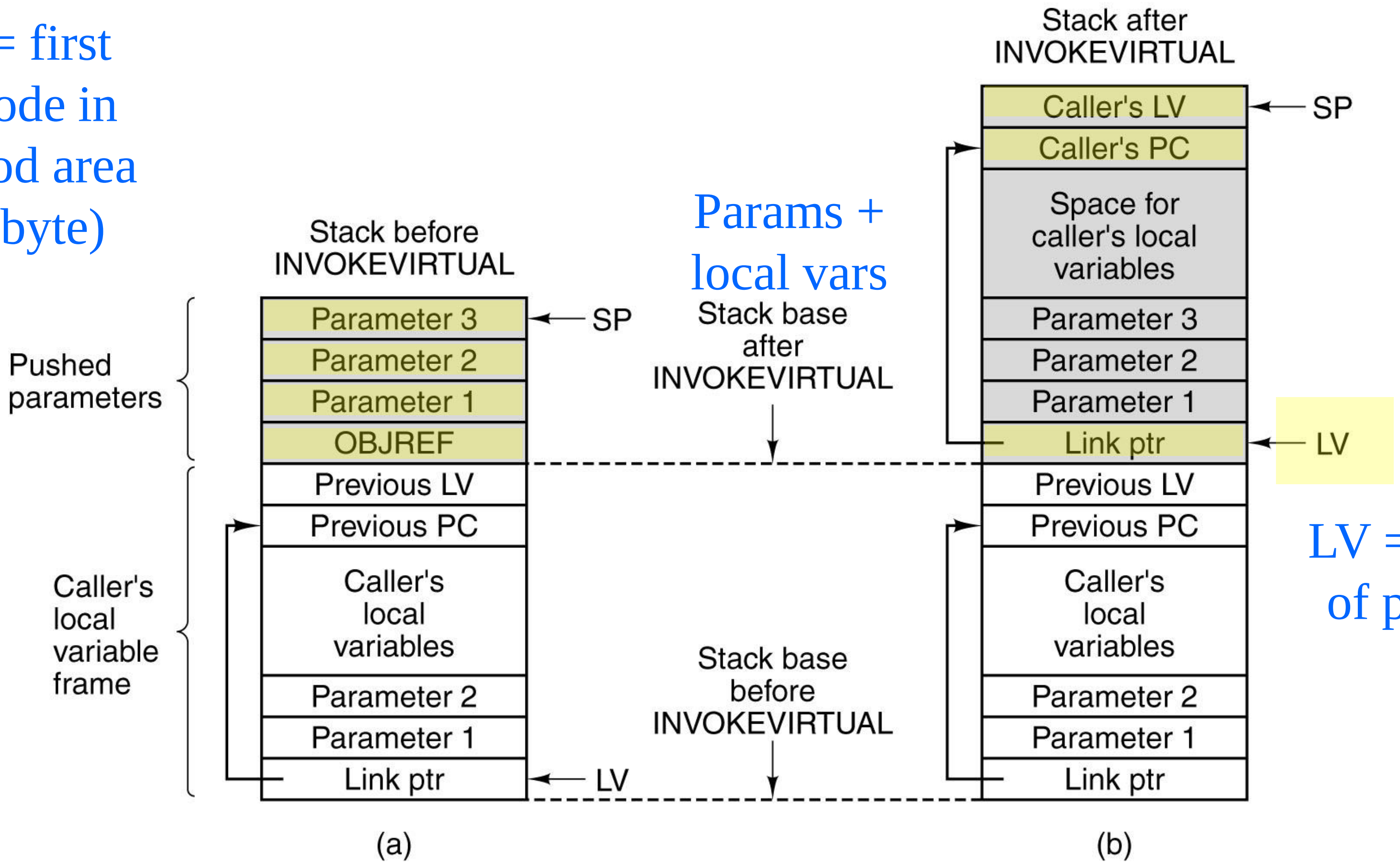
IVOKEVIRTUAL *disp*

- First 4 bytes of a method
  - First 2: number of parameters, including OBJREF (param 0)
  - Second 2: size of local variable area


disp
address
CP
0
1
2
3
OPCODE1
MA
Byte[0] Byte[1] = unsigned 16 bit int
Byte[2] Byte[3] = unsigned 16 bit int

INVOKEVIRTUAL

PC = first Opcode in method area (5th byte)



Params + local vars

LV = SP - # of params

a) Memory before executing INVOKEVIRTUAL.

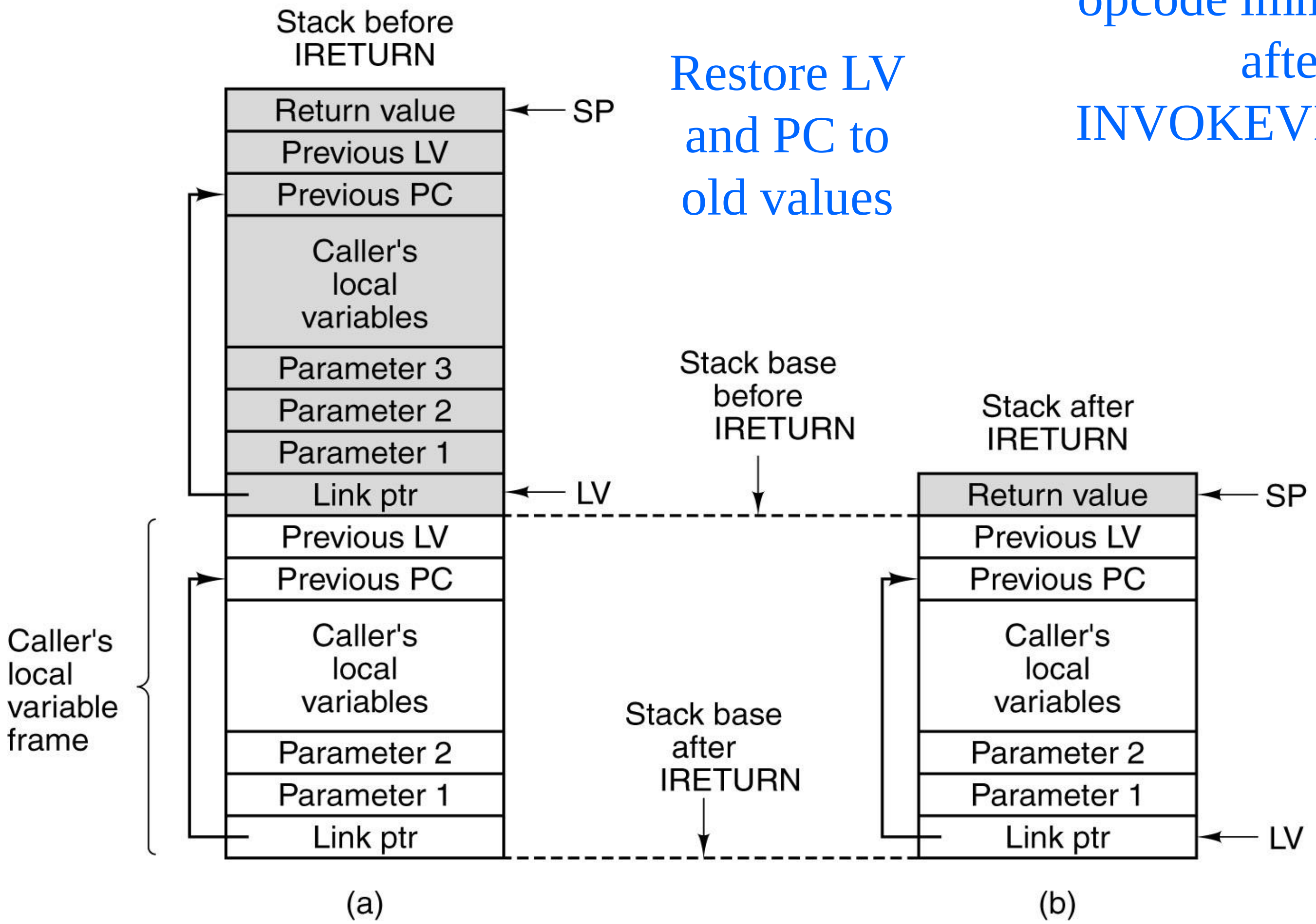
b) After executing it.

Tanenbaum, Structured Computer Organization, Fifth Edition, (c) 2006 Pearson Education, Inc. All rights reserved. 0-13-148521-0

IRETURN

Control is returned to opcode immediately after INVOKEVIRTUAL

Restore LV and PC to old values



a) Memory before executing IRETURN.

b) After executing it.

Tanenbaum, Structured Computer Organization, Fifth Edition, (c) 2006 Pearson Education, Inc. All rights reserved. 0-13-148521-0

Exercise

- Convert the following HL Java code to IJVM

```
int j = 100
int s = 0
for (int i = 0; i < j; i++)
    s += i
```

```
.main   // main program
.var    // local variables for main
  i
  j
  s
.end-var
```

```
start:
   BIPUSH 0x64         // 100
   ISTORE j
   BIPUSH 0x0
   DUP
   ISTORE s
   ISTORE i
```

```
check:
   ILOAD i
   ILOAD j
   ISUB // i - j
   IFLT for_body // i < j
   GOTO end
for_body:
   // s += i
   ILOAD i
   ILOAD s
   IADD
   ISTORE s
   // i++
   ILOAD i
   BIPUSH 0x1
   IADD
   ISTORE i
   GOTO check
end:

.end-main
```

Input and Output

Author: Dan Stone

```
.main

L1:   IN           // request character input from memory
      DUP          // duplicate top of stack (input char) for comparing
      BIPUSH 0x0   // push 0x0 for comparison
      IF_ICMPEQ L2 // if no characters are available for input, loop
      OUT          // else, print character
      GOTO L1      // loop back to beginning of program


L2:   POP          // No key has been pushed, so clear the stack,
      GOTO L1      // and start over
.end-main
```

Defining Methods in JAS

typically after main

```
.method methodName()
.var
variables
.end-var
body
…
IRETURN
.end-method
```

Calling Methods in JAS

```
LDC_W OBJREF
INVOKEVIRTUAL methodName
```

Before `.main`, you need:

```
.constant
   OBJREF 0x40
.end-constant
```

Calling Methods with Parameters

```
LDC_W OBJREF
ILOAD param1
ILOAD param2
INVOKEVIRTUAL methodName
```

Using LDC_W

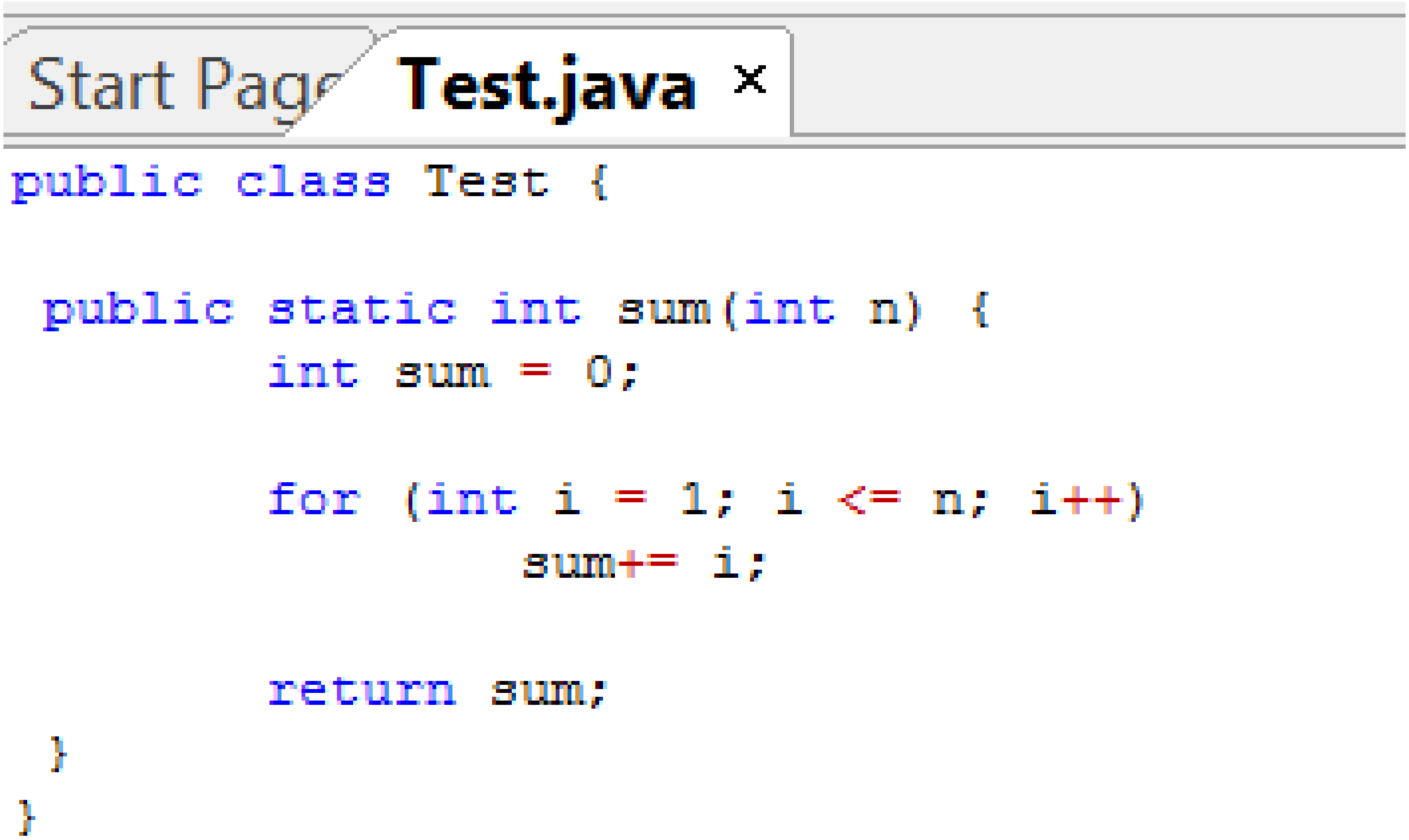
```
// this program displays all the printable ASCII values 32..126
 .constant
   one 1
   start 32
   stop 126
.end-constant
.main
   LDC_W start
   next: DUP
   OUT // output the current character
   DUP
   LDC_W stop
   ISUB IFEQ done // exit if we've reached the end
   LDC_W one
   IADD
   GOTO next // increment and do the next one done
   POP
   HALT
.end-main
```

Exercise

- Write a JAS method *int sum(int n)* that calculates and returns the sum of integers up to *n*,

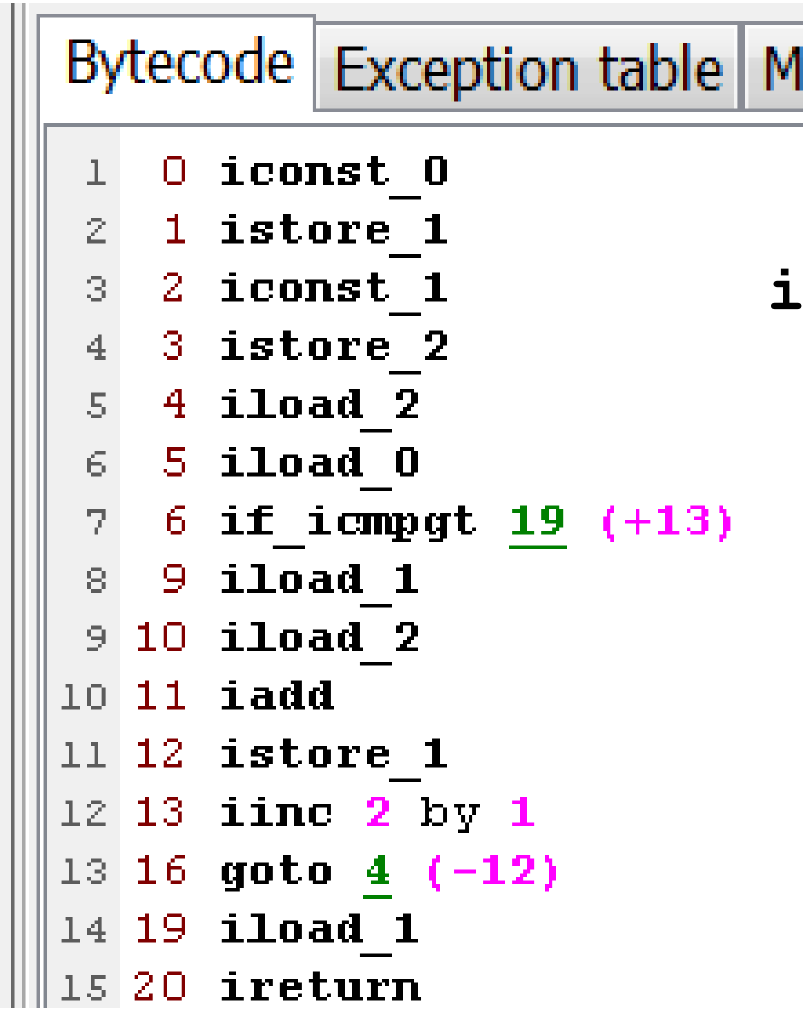
- $\sum_{i=1}^{n} i$

HL Java code (static method)



```
public class Test {

 public static int sum(int n) {
        int sum = 0;

        for (int i = 1; i <= n; i++)
                sum+= i;

        return sum;
 }
}
```

JAS code



```
 0 iconst_0
 1 istore_1
 2 iconst_1
 3 istore_2
 4 iload_2
 5 iload_0
 6 if_icmpgt 19 (+13)
 9 iload_1
10 iload_2
11 iadd
12 istore_1
13 iinc 2 by 1
16 goto 4 (-12)
19 iload_1
20 ireturn
```

Using *Jclasslib* byte code viewer

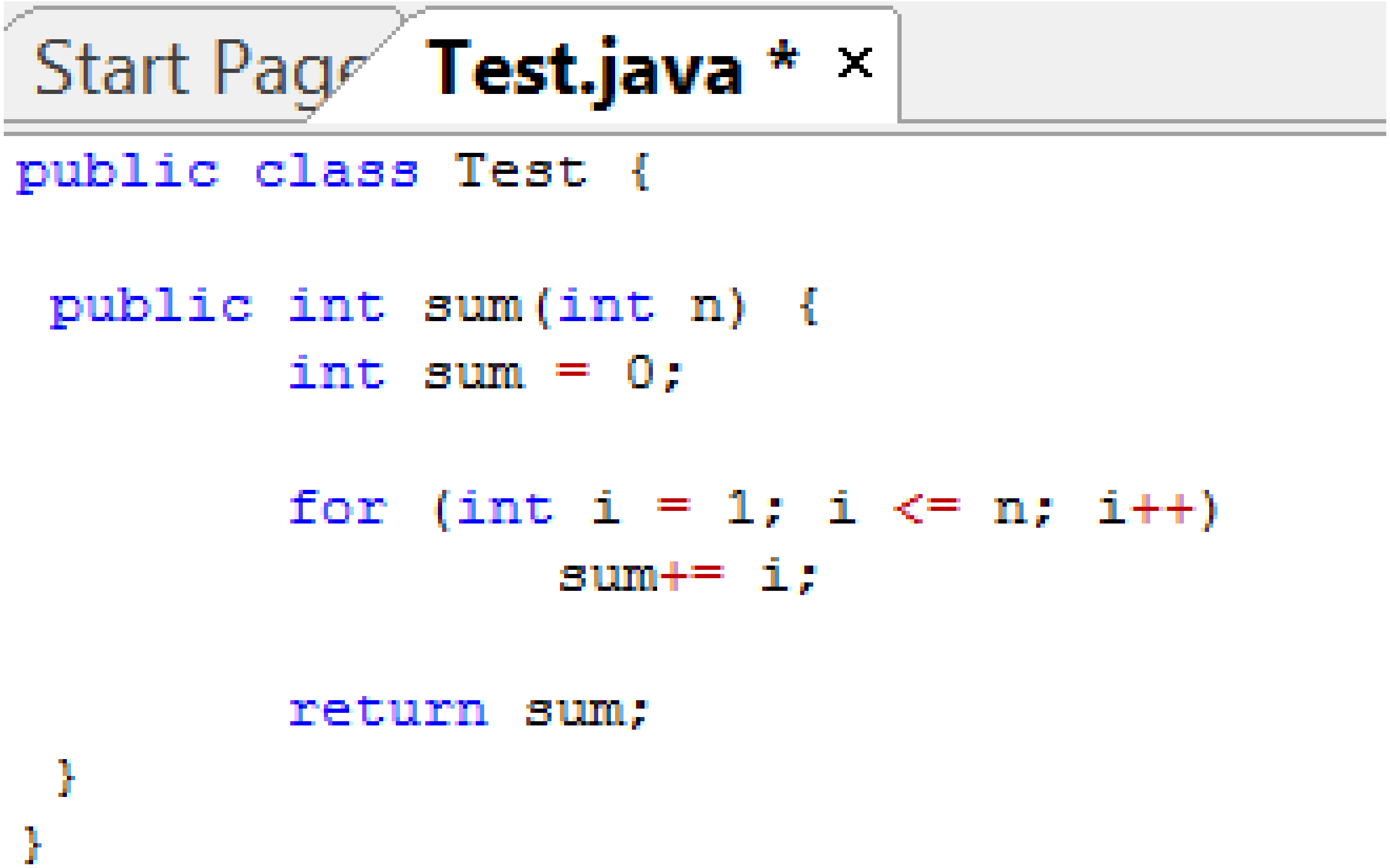
`iconst` loads a constant to the stack

`sum` is var 1

`i` is var 2

Parameter **`n`** is var 0

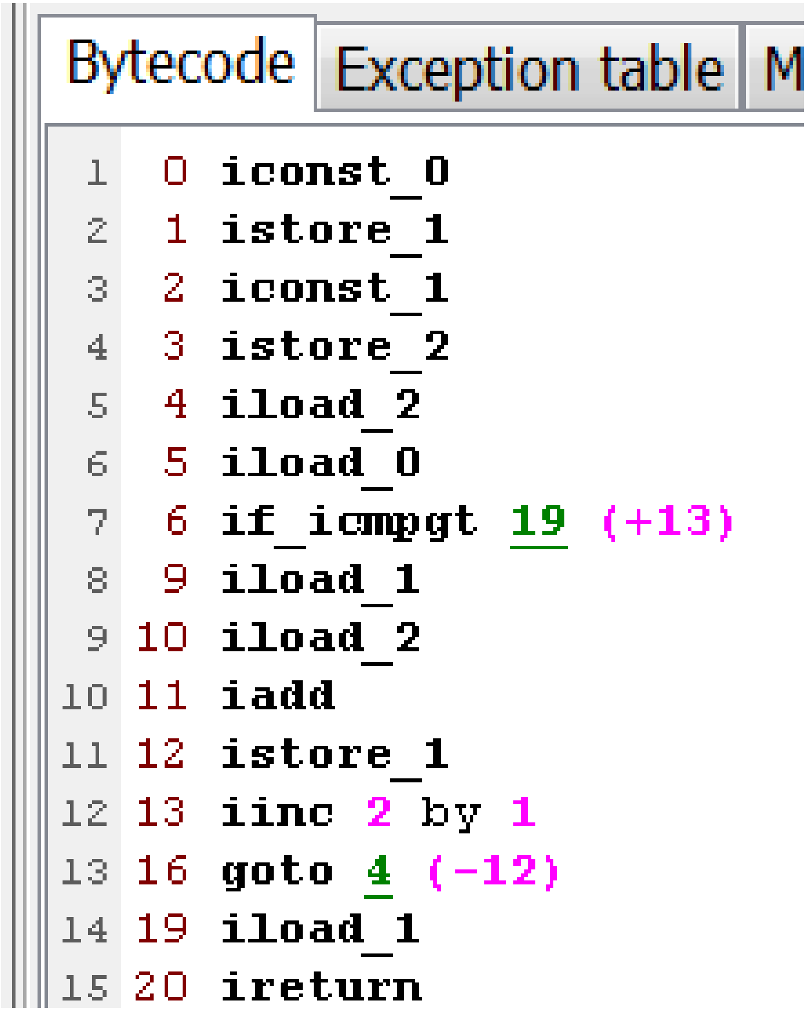
HL Java code (instance method)



```
public class Test {

 public int sum(int n)  {
        int sum = 0;

        for (int i = 1; i <= n; i++)
                sum+= i;

        return sum;
 }
}
```

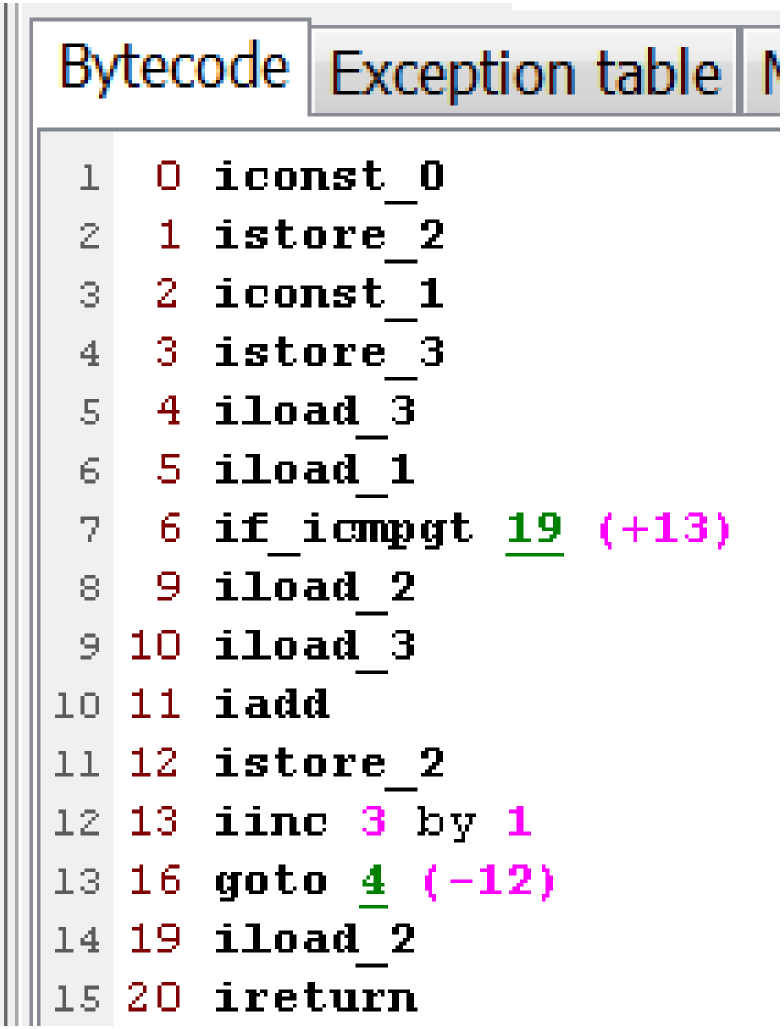
JAS code

Static



```
 0 iconst_0
 1 istore_1
 2 iconst_1
 3 istore_2
 4 iload_2
 5 iload_0
 6 if_icmpgt 19 (+13)
 9 iload_1
10 iload_2
11 iadd
12 istore_1
13 iinc 2 by 1
16 goto 4 (-12)
19 iload_1
20 ireturn
```

Instance



```
 0 iconst_0
 1 istore_2
 2 iconst_1
 3 istore_3
 4 iload_3
 5 iload_1
 6 if_icmpgt 19 (+13)
 9 iload_2
10 iload_3
11 iadd
12 istore_2
13 iinc 3 by 1
16 goto 4 (-12)
19 iload_2
20 ireturn
```

Variables are shifted by 1, since var 0 is OBJREF