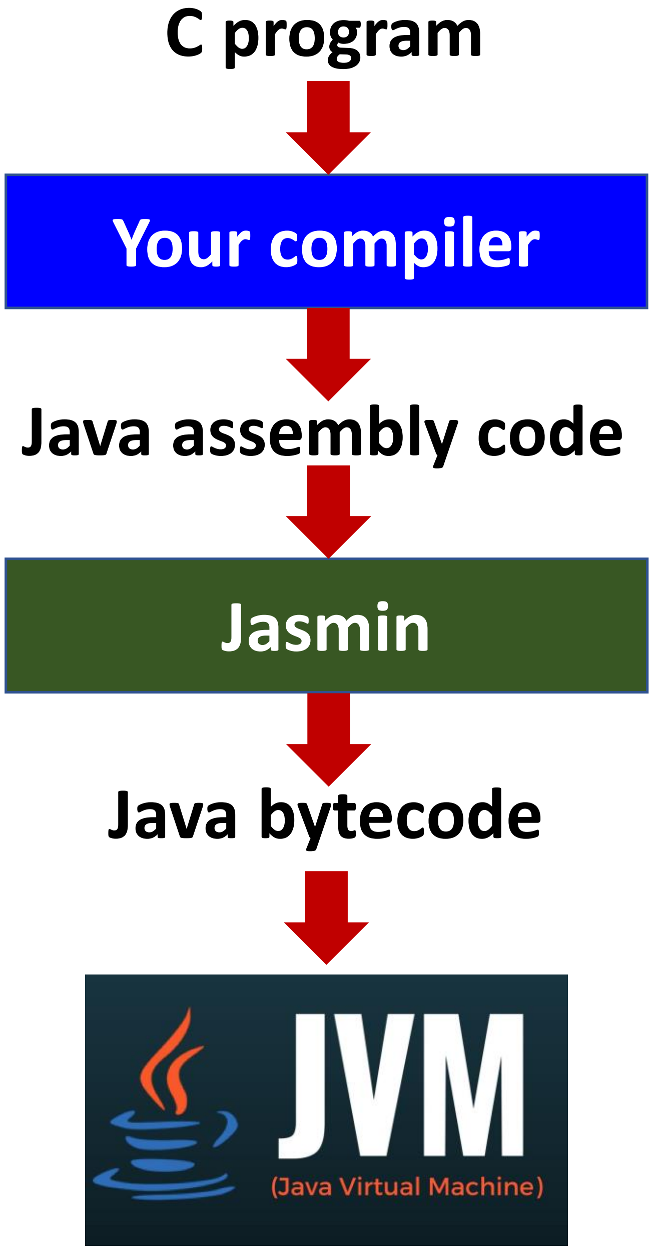
Jasmin

Assembler for Java Virtual Machine

Final Project



C program

Your compiler

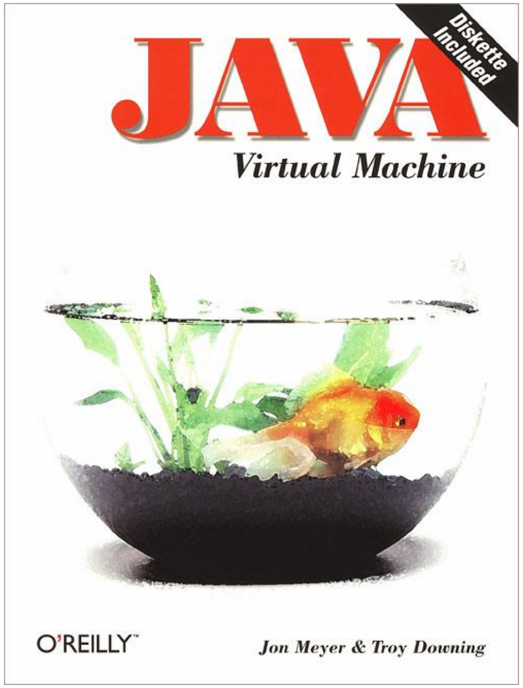
Java assembly code

Jasmin

Java bytecode

Jasmin

- Jasmin is an assembler for the Java Virtual Machine.
- It takes ASCII descriptions of Java classes, written in a simple assembler-like syntax using the Java Virtual Machine instruction set.
- Webpage
  - http://jasmin.sourceforge.net/
- Contributors
  - Jon Meyer - Original Author
    Daniel Reynaud
    Iouri Kharon



Jasmin

```
$ java –jar jasmin.jar  xxx.j
```

(Generate **xxx.class)**

```
$ java xxx
(Execution)
```

Java 指令

Intel 80386	Java 虛擬機器
• mov EAX, 10	bipush 10 istore_1
• mov EAX, 5 mov EBX, 10 add EAX, EBX	bipush 5 bipush 10 iadd istore_1

Java 指令種類

數學運算	24	iadd, lsub, frem
邏輯操作	12	iand, lor, ishl
數字轉換	15	i2s, f2l, d2i
堆入常數	20	bipush, sipush, ldc, iconst_0, fconst_1
堆疊處理	9	pop, pop2, dup, dup2
流程控制	28	goto, ifne, ifge, if_null, jsr, ret
區域變數處理	52	astore, istore, aload, iload, aload_0
陣列處理	17	aastore, bastore, aaload, baload
建立物件和陣列	4	new, newarray, anewarray, multianewarray
物件處理	6	getfield, putfield, getstatic, putstatic
method 呼叫及返回	10	invokevirtual, invokestatic, areturn
其他	5	throw, monitorenter, breakpoint, nop

Java 虛擬機器的缺點

- 8 位元 opcode
- 某些 type 就被貶為次要地位 (short, byte, char)
- 不易擴充指令集

為何要研究 Java 虛擬機器？

- 直接產生 Java bytecode
- 擴充語言、自訂新語言
- 深入瞭解並掌握 Java 系統

JVM Stack

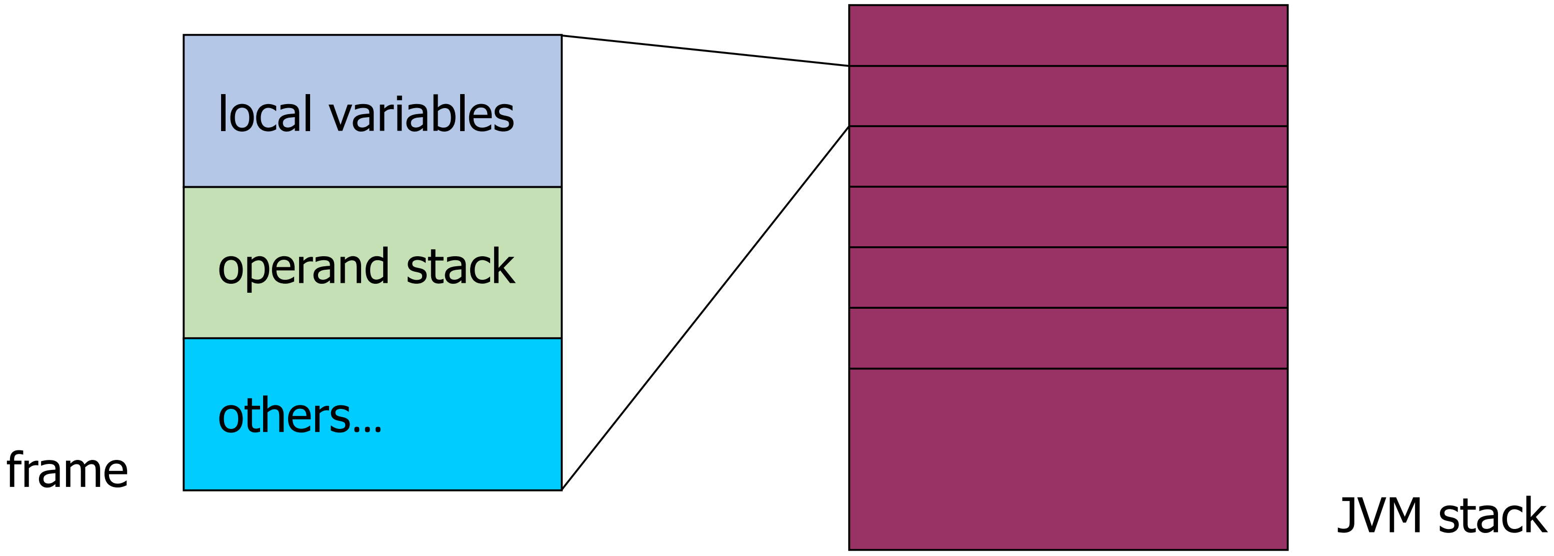
- Java stack stores frames.
- Each thread has a private JVM stack, created at the same time as thread.
- A JVM stack is analogous to the stack of a conventional language such as C: it holds local variables and partial results, and plays a part in method invocation and return.

JVM Stack : frame (1)

- A new frame is created each time a method is invoked.
- A frame is destroyed when its method invocation completes.
- A frame is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exception.

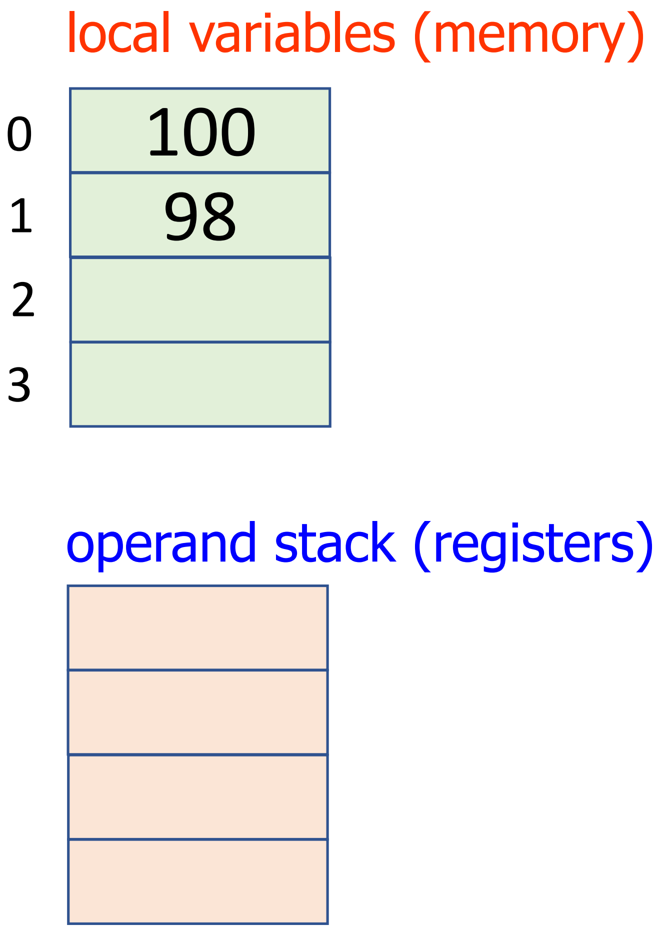
JVM Stack : frame (2)

- Each frame has its own array of local variables, its own operand stack, and a reference to the runtime constant pool of the class of the current method.



JVM Stack : frame (3)

```
iload_0

iload_1

iadd

istore_2
```

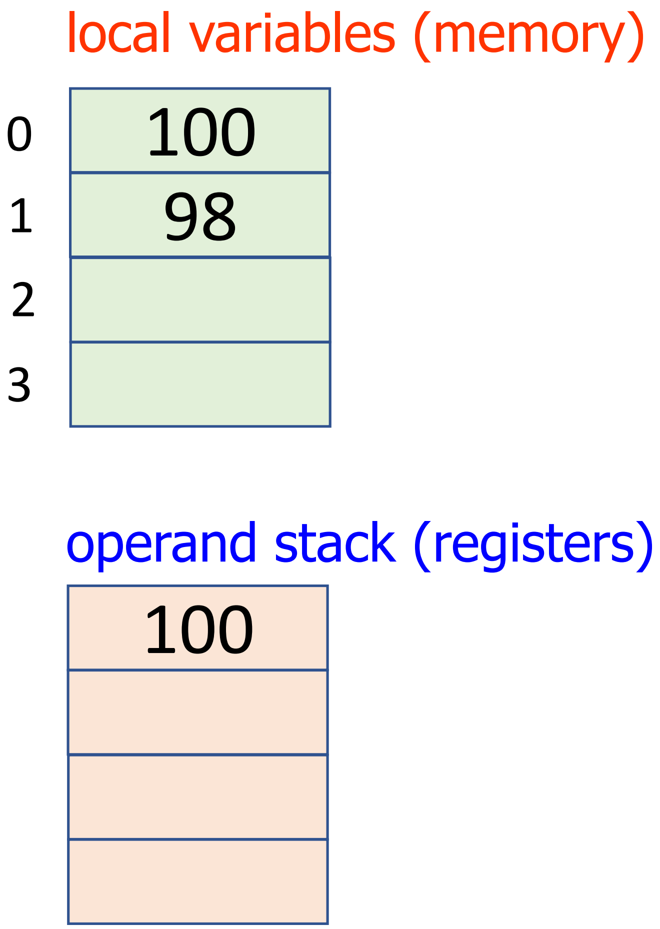


local variables (memory)

0	100
1	98
2	
3	

operand stack (registers)

JVM Stack : frame (3)



local variables (memory)

0	100
1	98
2	
3	

operand stack (registers)

100

```
iload_0

iload_1

iadd

istore_2
```

JVM Stack : frame (3)

local variables (memory)

0	100
1	98
2	
3	

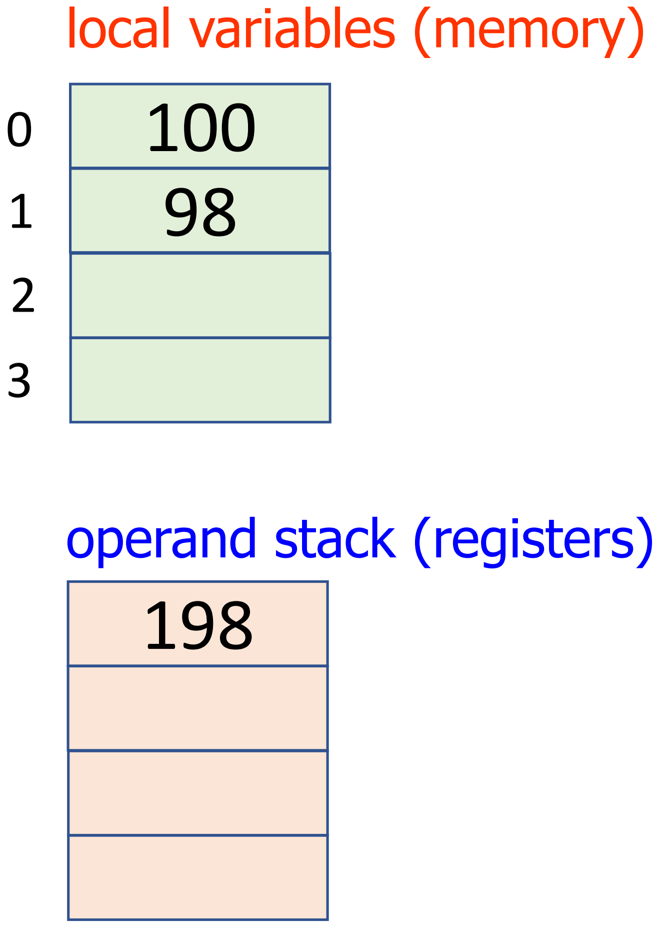
operand stack (registers)

98
100

```
iload_0
iload_1
iadd
istore_2
```

JVM Stack : frame (3)

```
iload_0

iload_1

iadd

istore_2
```



local variables (memory)

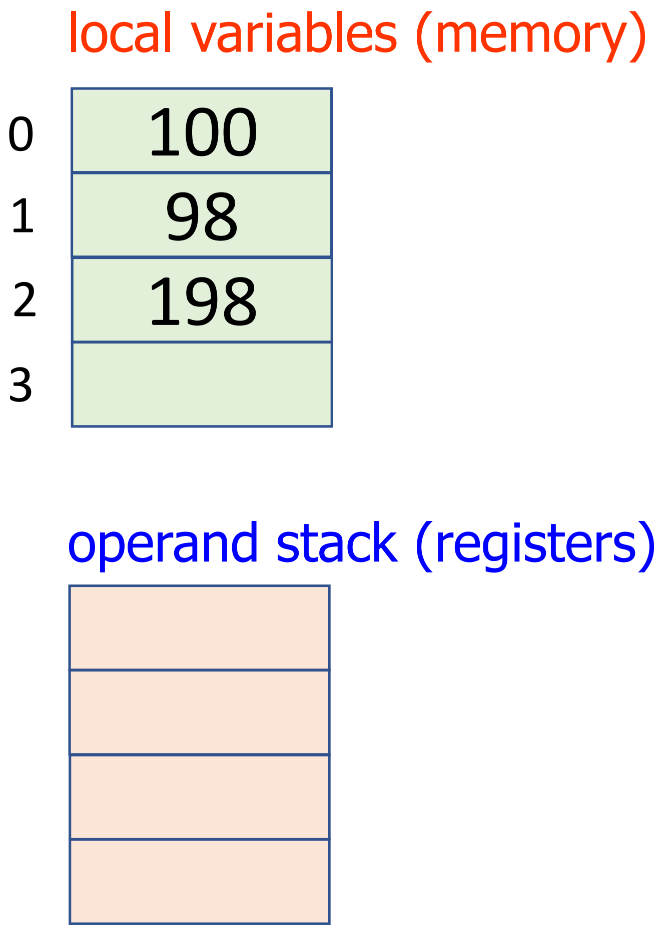
0	100
1	98
2	
3	

operand stack (registers)

198

JVM Stack : frame (3)

```
iload_0

iload_1

iadd

istore_2
```



local variables (memory)

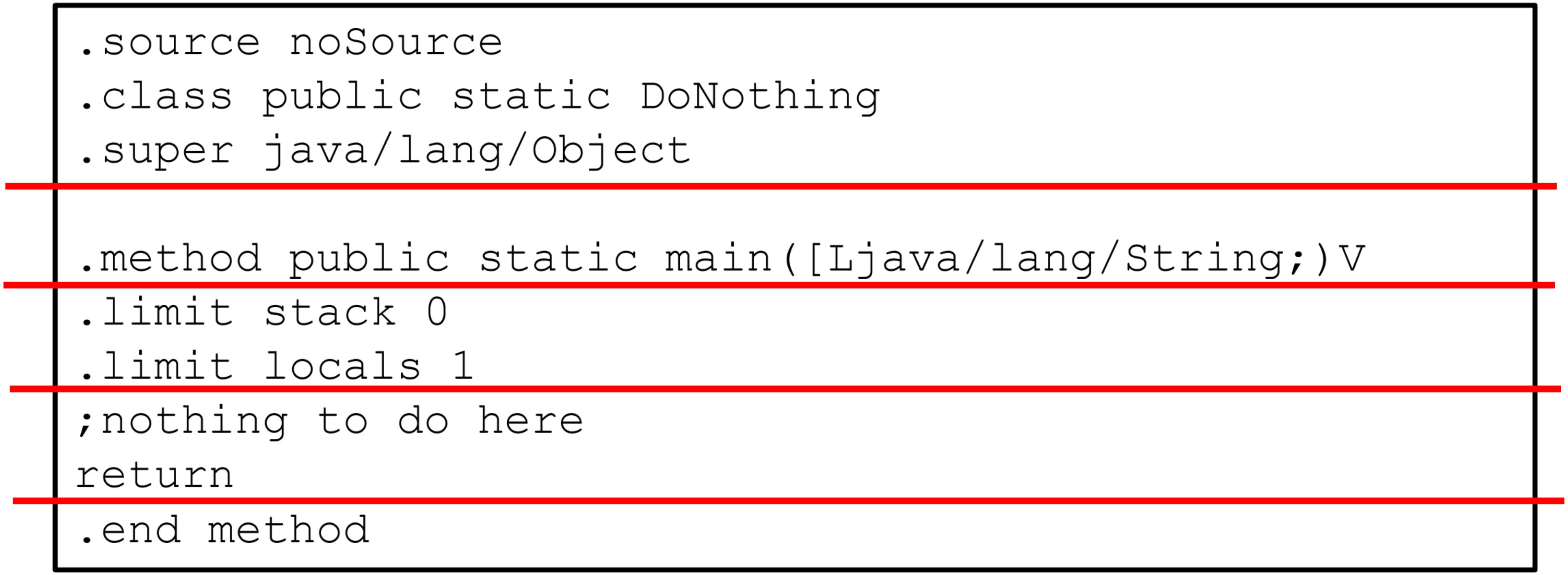
0	100
1	98
2	198
3	

operand stack (registers)

Java Assembly Language

Example

File: DoNothing.j



```
.source noSource
.class public static DoNothing
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit stack 0
.limit locals 1
;nothing to do here
return
.end method
```

Reference from http://www.ist.tugraz.at/_attach/Publish/Cb14/Jasmin.pdf

Jasmin Syntax (1)

- One statement per line
- Inline comments, initiated by **;**
- **.source**: Source of assembly
  - e.g.: .source MyCompiler.j
- **.class**: Resulting java class description
  - e.g.: .class public static MyClass
- **.super**: Superclass of resulting java class
  - always: .super java/lang/Object

Jasmin Syntax (2)

- **.method** <method signature>
  - Ex: .method public static main([Ljava/lang/String;)V
- **.limit stack n** => Sets the maximum size of the operand stack required by the method.
- **.limit locals n** => Sets the number of local variables required by the method. (ex: n = #parameters + #local_vars + #temp_vars)
- return => requires matching type on top-of-stack for non-void returns
  - Ex: ireturn
- **.end method**

Primitive Data Type in JVM

- byte: 8-bit signed two's-complement integers (default value is zero)
- short: 16-bit signed two's-complement integers (default value is zero)
- int: 32-bit signed two's-complement integers (default value is zero)
- long: 64-bit signed two's-complement integers (default value is zero)
- char: 16-bit unsigned integers
- float:,32 bits and default value is positive zero
- double: 64 bits and default value is positive zero

Class Data Type

- **Class Type:**
  - `L<fullclassname>;`

  - **Ex1**: `Ljava/lang/String;`
  - **Ex2**: `Ljava/io/PrintStream;`

Array Data Type

- Array Type:
  - `[<type>`

  - Ex1: `[C` → `char[ ]`
  - Ex2: `[[F` → `float[ ][ ]`
  - Ex3: `[Ljava/lang/Thread;` → `Thread[]`

Type Descriptor: Method

- Method:
  - (<argument_types>)<return_type>

  - “V” indicates “void” (used in return type)

  - Ex1: `()V`

➔ `void xyz() {…}`

- Ex2: `(SF[Ljava/lang/Thread;)I`

➔ `int xyz(short x, float y, Thread[] z) {…}`

Data Management (1)

- Each method has its own operand stack
  - Most instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack.
  - Each entry on the operand stack can hold a value of any Java Virtual Machine type, including a value of type long or type double.

Data Management (2)

- Each method has its own local variables (an array of variables)
  - Local variables are addressed by indexing. The index of the first local variable is zero.
  - A single local variable can hold a value of type boolean, byte, char, short, int, float, reference, or returnAddress.
  - A pair of local variables can hold a value of type long or double.
  - A value of type **long** or type **double** occupies two consecutive local variables.
  - Such a value may only be addressed using the lesser index.

Instructions

Please reference the following links:

- https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html#jvms-6.5.ldc
- http://jasmin.sourceforge.net/instructions.html

Instruction: Handling Local Variables

- **iload n**: push integer, stored in index n of local variables, onto stack
- **istore n**: pop integer from stack and stores it into index n of local variables
- **aload n**: push object, stored in index n of local variables, onto stack
- **astore n**: pop object from stack and stores it into index n of local variables

Instruction: Constant

- Push integer constant n onto the operand stack
  - sipush n => push short
  - bipush n => push byte

- ldc “<string>“
  - Push string constant <string> onto the operand stack
  - Ex: ldc “Hello World"

Instruction: Arithmetic Operators

- ineg: toggles sign of int on top of stack
- iadd: add two integers
- imul: multiply two integers
- idiv: divide two integers
- irem: modulo division of two integers

Instruction: Logic Operators

- iand bitwise and of two integers
- ior bitwise or of two integers

Observation (1)

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello! World!");
    }
}
```

Observation (2)

```
.source noSource
.class public HelloWorld
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 0
.limit locals 1


;nothing to do here


return
.end method
```

Observation (3)

```
public final class System {
    …
    public final static PrintStream out = null;
    …
}
```

```
.source noSource
.class public HelloWorld
.super java/lang/Object
.method public static main([Ljava/lang/String;)V
.limit stack 10

.limit locals 10

getstatic java/lang/System/out Ljava/io/PrintStream;

ldc "Hello World!"

invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V


return

.end method
```

out是一個類別，它的type是 PrintStream

- getstatic: field manipulation instruction
  (put “static field” into the operand stack)

```
getstatic java/lang/System/out Ljava/io/PrintStream;
```

- out 是 System 類別裡的 static field
- 資料型態是 PrintStream 類別

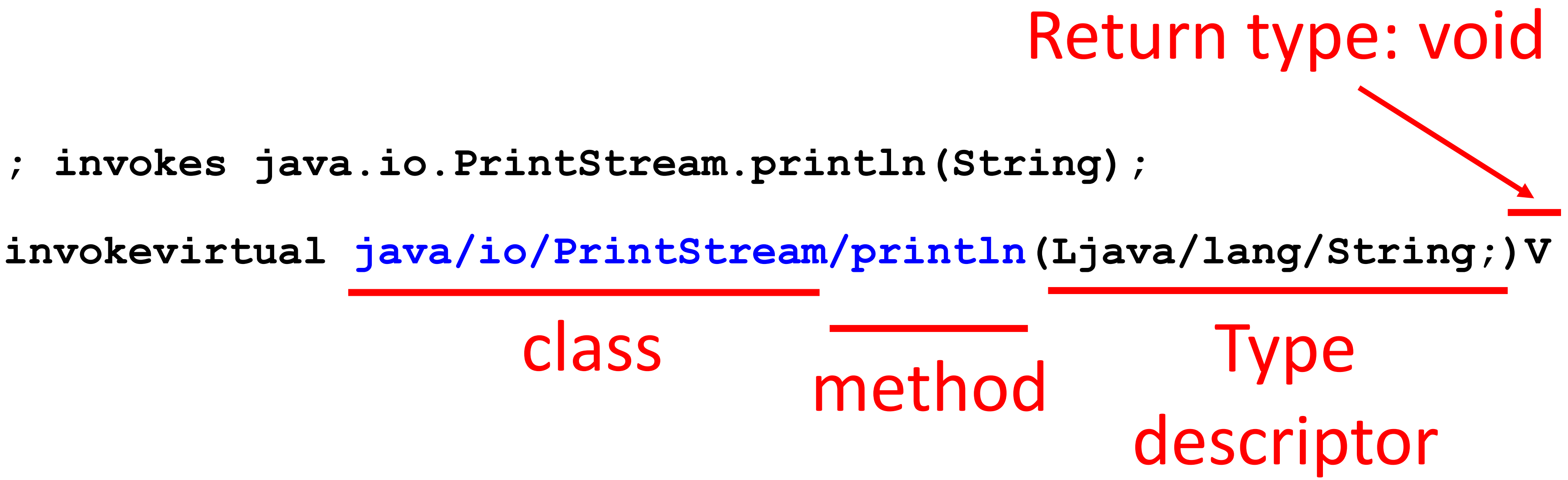
- ldc <constant>: load constant into operand stack

```
ldc 1.2                ; push a float
ldc 10                 ; push an int
ldc "Hello World"      ; push a String
```

```
ldc "Hello World!"
```

- Push “Hello World!” into the operand stack

- invokevirtual: an instruction is used to invoke methods.



Return type: void

```
; invokes java.io.PrintStream.println(String);
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

class

method

Type descriptor

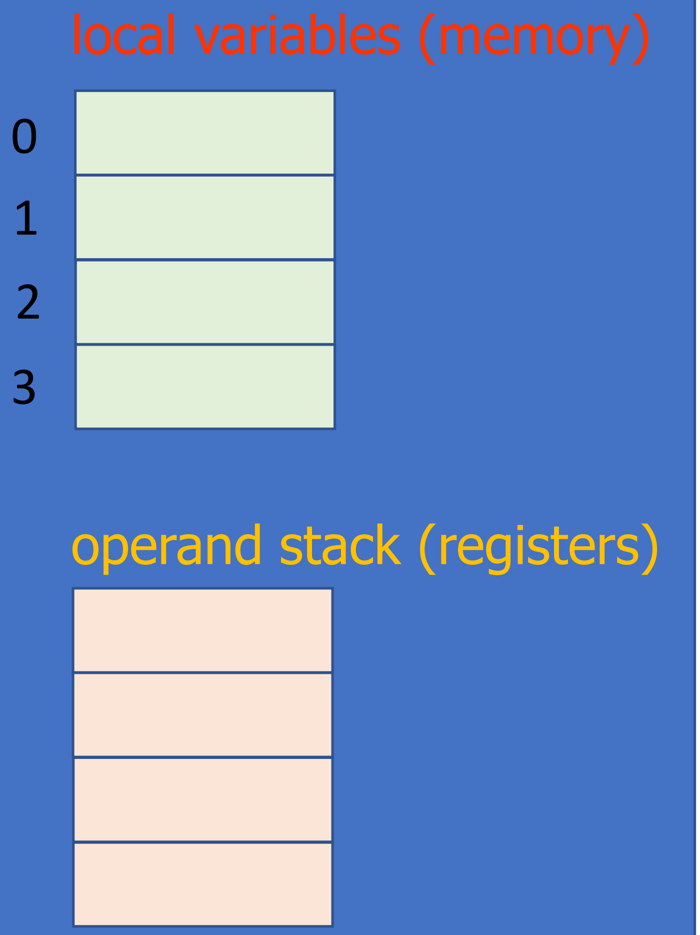
- Integer indicated by letter I
- Void indicated by letter V
- Float indicated by letter F
- Double indicated by letter D

Example: compute1.java

```
public class compute1{
    public static void main(String[] args)
{
        int num1;
        float b;

        b = 22.0f;
        num1 = (int) (b * 3.14f);
    }
}
```

Example: compute1.j (Jasmin version)

```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```

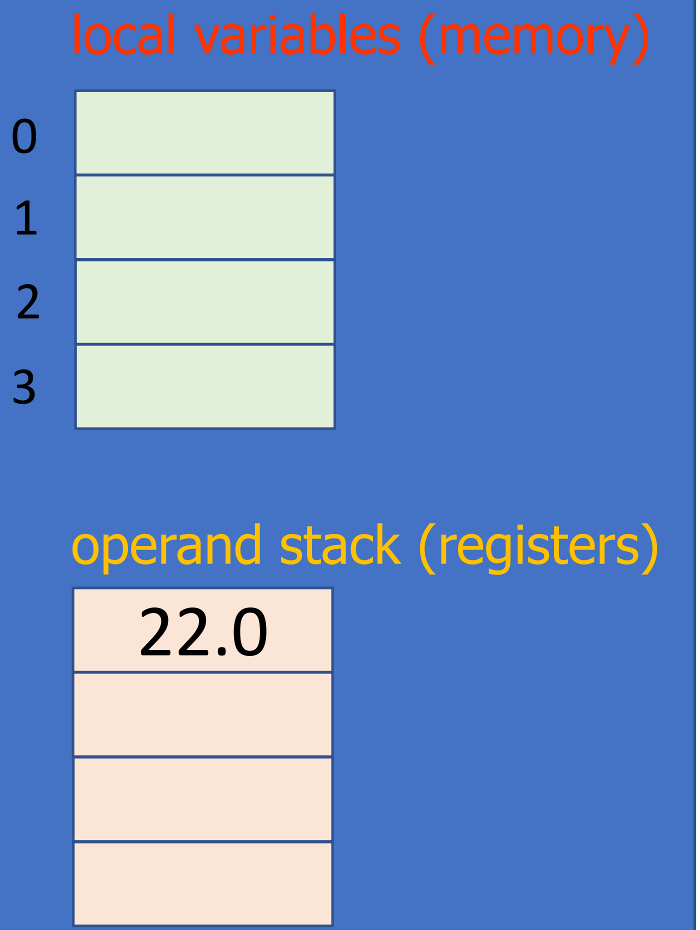


local variables (memory)

operand stack (registers)

Example: compute1.j

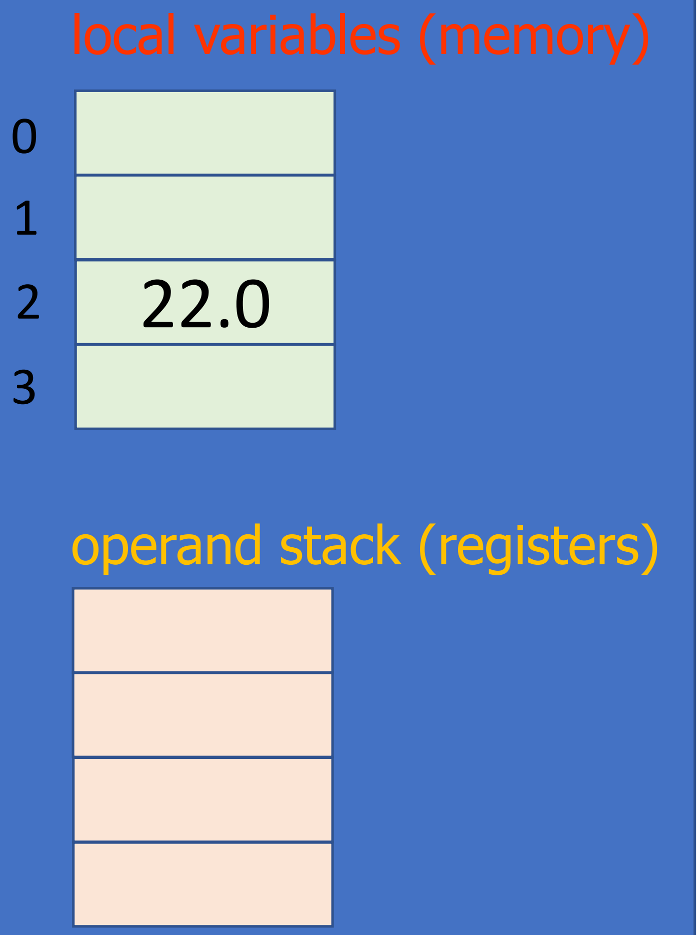
```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```


local variables (memory)

operand stack (registers)

Example: compute1.j

```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```



local variables (memory)

0

1

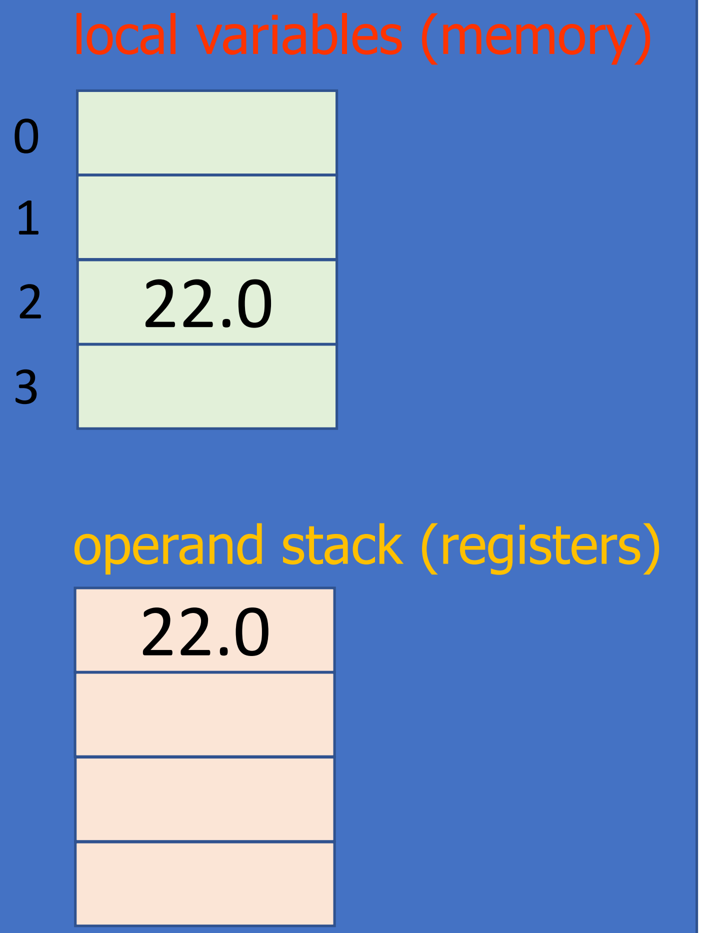
2 22.0

3

operand stack (registers)

Example: compute1.j

```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```



local variables (memory)

0	
1	
2	22.0
3	

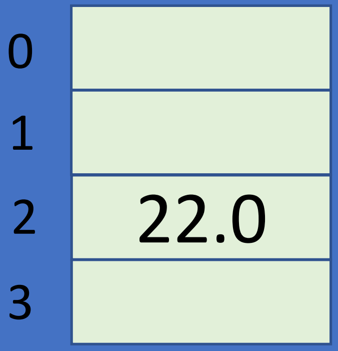
operand stack (registers)

22.0

Example: compute1.j

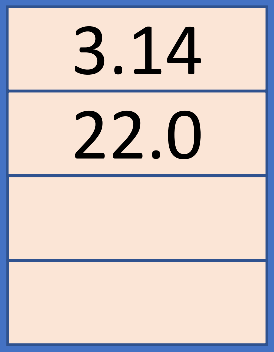
```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```

local variables (memory)



0	
1	
2	22.0
3	

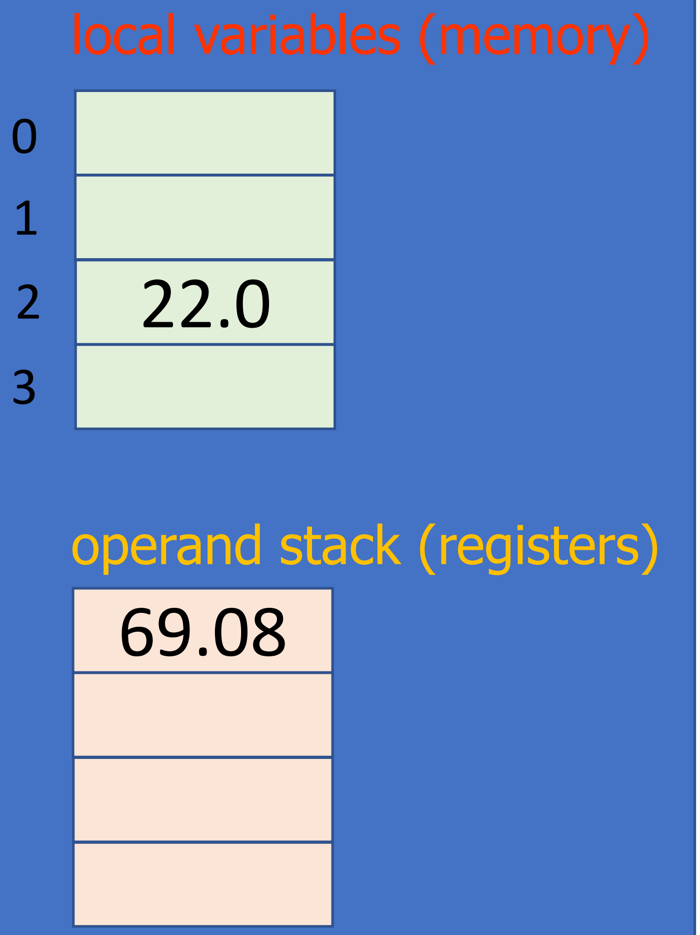
operand stack (registers)



3.14
22.0

Example: compute1.j

```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```



local variables (memory)

0	
1	
2	22.0
3	

operand stack (registers)

69.08

Example: compute1.j

```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```

local variables (memory)

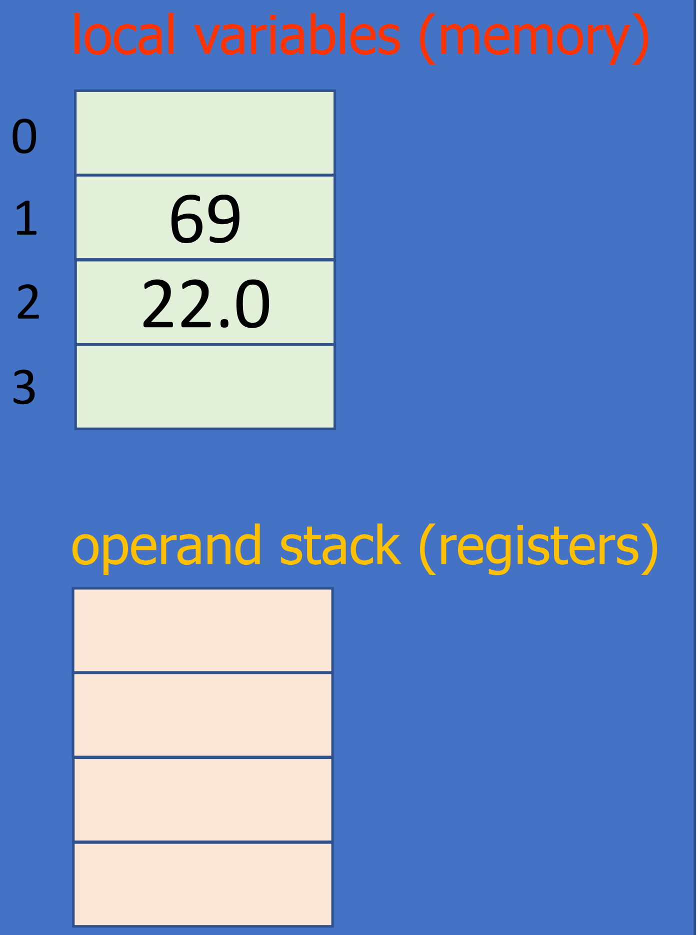
0	
1	
2	22.0
3	

operand stack (registers)

69

Example: compute1.j

```
.class public static compute1
.super java/lang/Object

.method public static
main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10

   ldc 22.0
   fstore 2
   fload 2
   ldc 3.14
   fmul
   f2i
   istore 1

   return
.end method
```



local variables (memory)

0	
1	69
2	22.0
3	

operand stack (registers)

Example: compute3.java

```java
public class compute3{
    public static void main(String[] args)
{
        float a, b;

        b = 22.0f;
        if (b > 0)
            a = 10;
        else
            a = b;
        System.out.println(a);
    }
}
```

Example: compute3.j (Jasmin version)

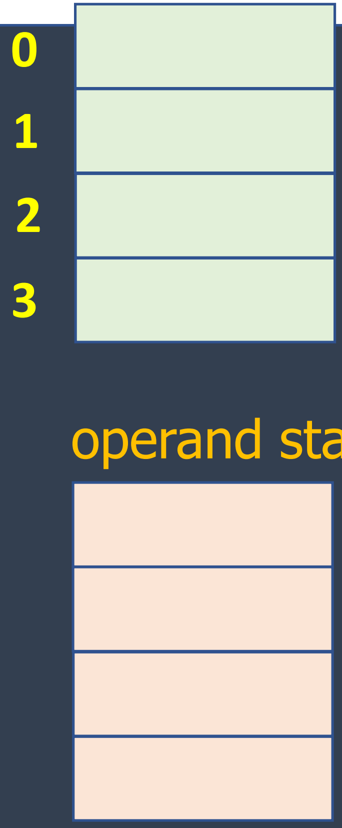
```
.class public static compute3
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
.limit stack 10
.limit locals 10
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1   ; a = 10
    goto END
ELSE:
    fload 2    ; load 22.0 into operand stack.
    fstore 1   ; a = b
END:
    ; print the value.
    getstatic java/lang/System/out Ljava/io/PrintStream;
    fload 1
    invokevirtual java/io/PrintStream/println(F)V

    return
return
.end method
```

Example: compute3.j

```
   ldc 22.0f
   fstore 2
   fload 2
   ldc 0.0f
   fcmpl
   ifle ELSE
   ldc 10.0f
   fstore 1    ; a = 10
   goto END
ELSE:
    fload 2     ; load 22.0 into operand stack.
    fstore 1    ; a = b
END:
```

local variables (memory)

operand stack (registers)

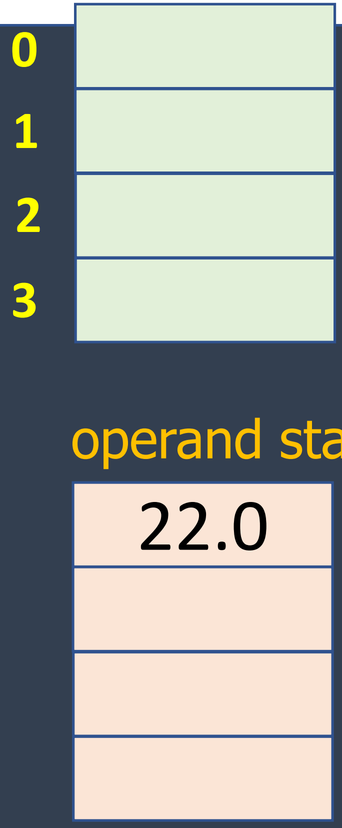


Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1     ; a = 10
    goto END
ELSE:
     fload 2      ; load 22.0 into operand stack.
     fstore 1     ; a = b
END:
```

local variables (memory)

operand stack (registers)



Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1    ; a = 10
    goto END
ELSE:
     fload 2     ; load 22.0 into operand stack.
     fstore 1    ; a = b
END:
```

local variables (memory)

0	
1	
2	22.0
3	

operand stack (registers)

Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1     ; a = 10
    goto END
ELSE:
     fload 2     ; load 22.0 into operand stack.
     fstore 1    ; a = b
END:
```

```
if (b > 0)
    a = 10;
else
    a = b;
```

local variables (memory)

0	
1	
2	22.0
3	

operand stack (registers)

22.0

Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1      ; a = 10
    goto END
ELSE:
     fload 2      ; load 22.0 into operand stack.
     fstore 1     ; a = b
END:
```

```
if (b > 0)
    a = 10;
else
    a = b;
```

local variables (memory)

0	
1	
2	22.0
3	

operand stack (registers)

0.0
22.0

Example: compute3.j

```
   ldc 22.0f
   fstore 2
   fload 2
   ldc 0.0f
   fcmpl
   ifle ELSE
   ldc 10.0f
   fstore 1     ; a = 10
   goto END
ELSE:
    fload 2      ; load 22.0 into operand stack.
    fstore 1     ; a = b
END:
```

```
if (b > 0)
   a = 10;
else
   a = b;
```

local variables (memory)

0	
1	
2	22.0
3	

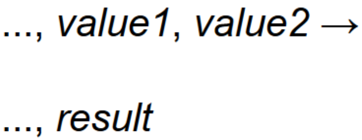
operand stack (registers)

1

**Please reference the following link for the description of all instructions.
https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html**

Instruction: fcmpl

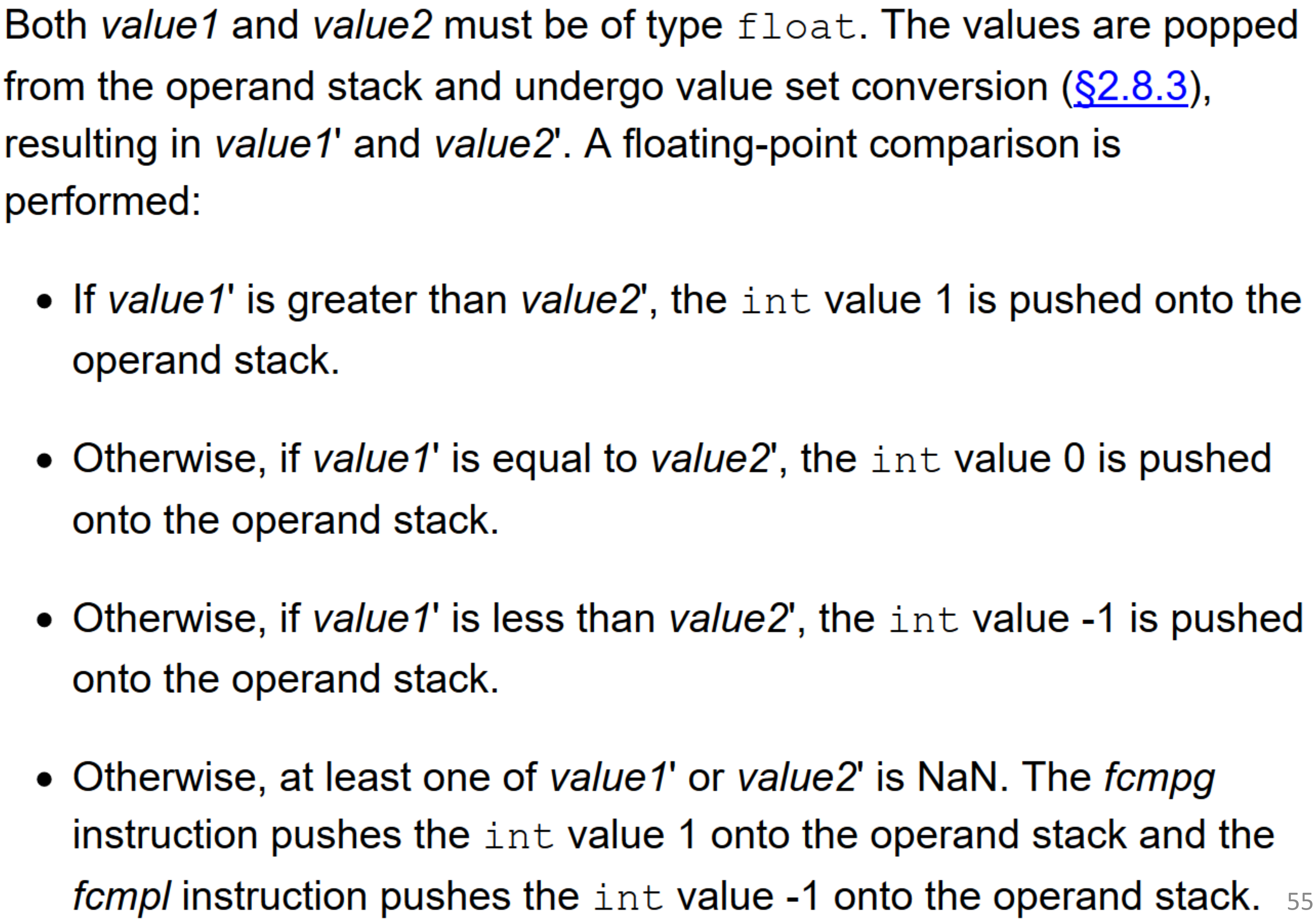
Operand Stack



..., *value1*, *value2* →

..., *result*

Description



Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and undergo value set conversion (§2.8.3), resulting in *value1*' and *value2*'. A floating-point comparison is performed:

- If *value1*' is greater than *value2*', the `int` value 1 is pushed onto the operand stack.
- Otherwise, if *value1*' is equal to *value2*', the `int` value 0 is pushed onto the operand stack.
- Otherwise, if *value1*' is less than *value2*', the `int` value -1 is pushed onto the operand stack.
- Otherwise, at least one of *value1*' or *value2*' is NaN. The *fcmpg* instruction pushes the `int` value 1 onto the operand stack and the *fcmpl* instruction pushes the `int` value -1 onto the operand stack.

Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1      ; a = 10
    goto END
ELSE:
     fload 2      ; load 22.0 into operand stack.
     fstore 1     ; a = b
END:
```

```
if (b > 0)
     a = 10;
else
     a = b;
```

local variables (memory)

0	
1	
2	22.0
3	

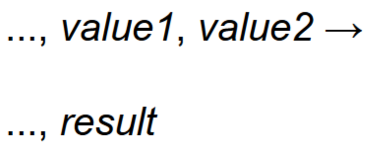
operand stack (registers)

0.0	Value 2
22	Value 1

**Please reference the following link for the description of all instructions.
https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html**

Instruction: fcmpl

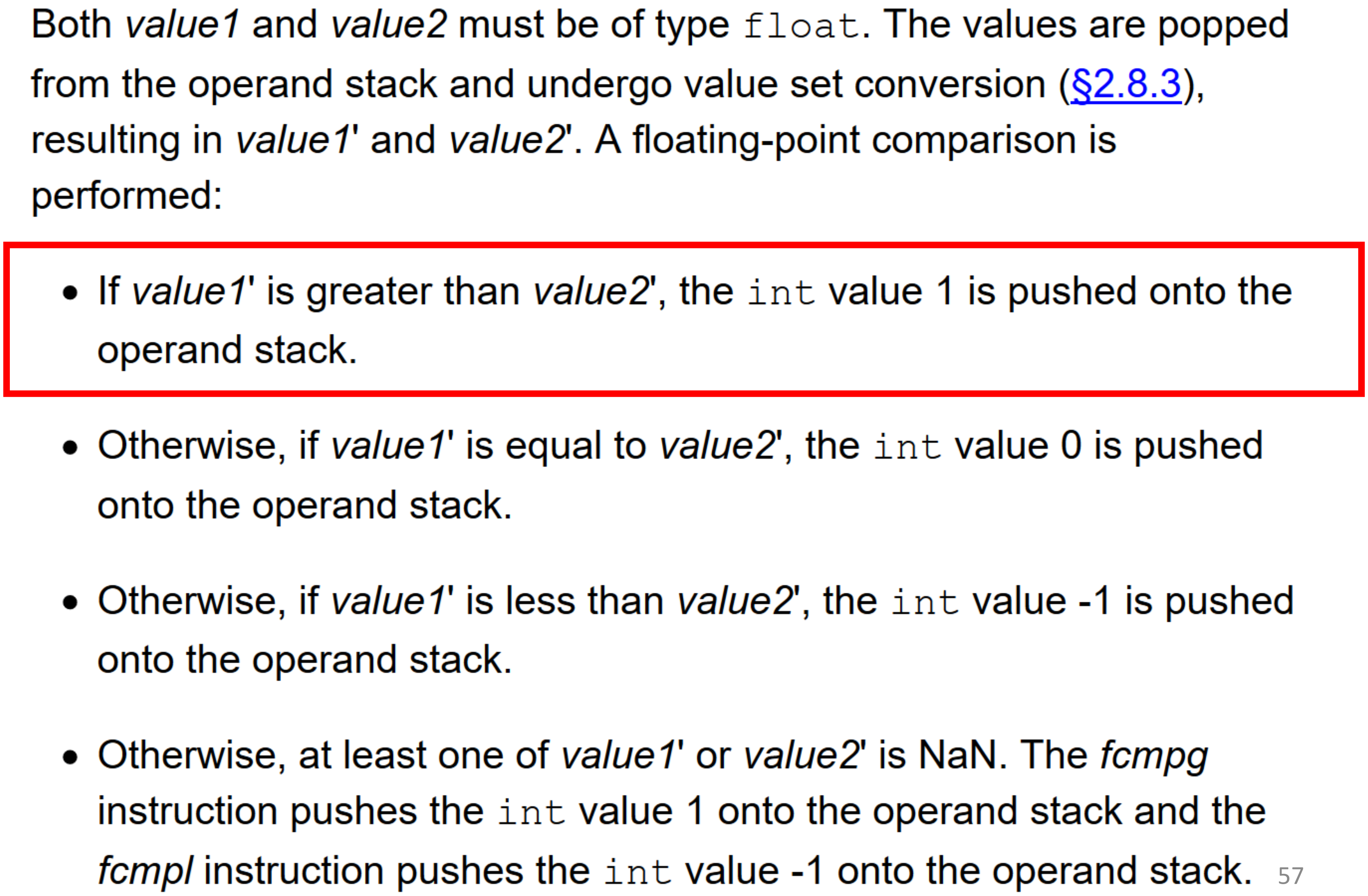
Operand Stack



..., *value1*, *value2* →

..., *result*

Description

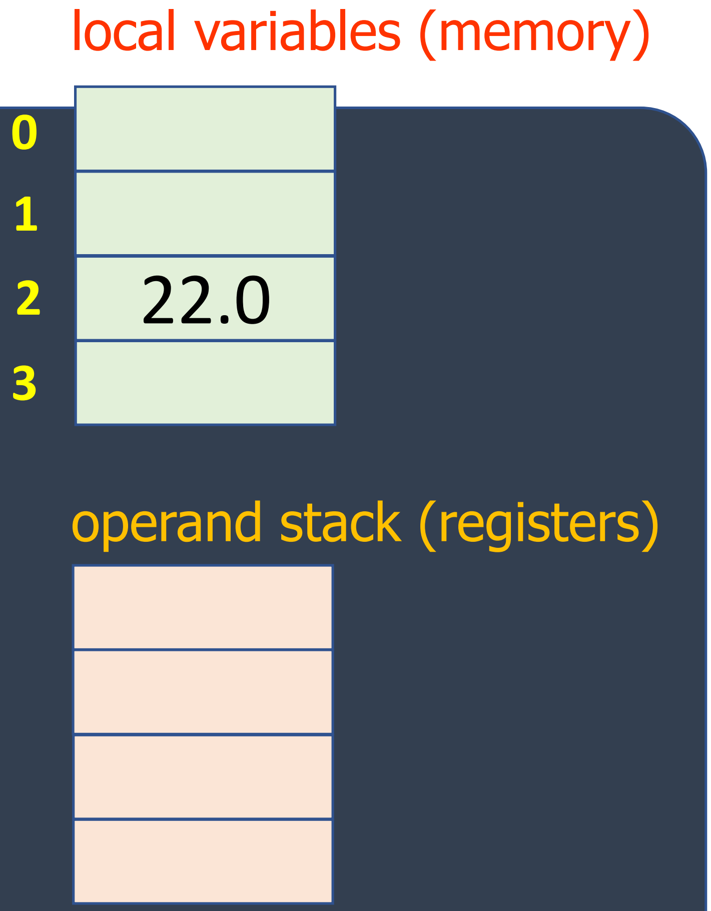


Both *value1* and *value2* must be of type `float`. The values are popped from the operand stack and undergo value set conversion (§2.8.3), resulting in *value1*' and *value2*'. A floating-point comparison is performed:

- If *value1*' is greater than *value2*', the `int` value 1 is pushed onto the operand stack.
- Otherwise, if *value1*' is equal to *value2*', the `int` value 0 is pushed onto the operand stack.
- Otherwise, if *value1*' is less than *value2*', the `int` value -1 is pushed onto the operand stack.
- Otherwise, at least one of *value1*' or *value2*' is NaN. The *fcmpg* instruction pushes the `int` value 1 onto the operand stack and the *fcmpl* instruction pushes the `int` value -1 onto the operand stack.

Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1     ; a = 10
    goto END
ELSE:
     fload 2      ; load 22.0 into operand stack.
     fstore 1     ; a = b
END:
```



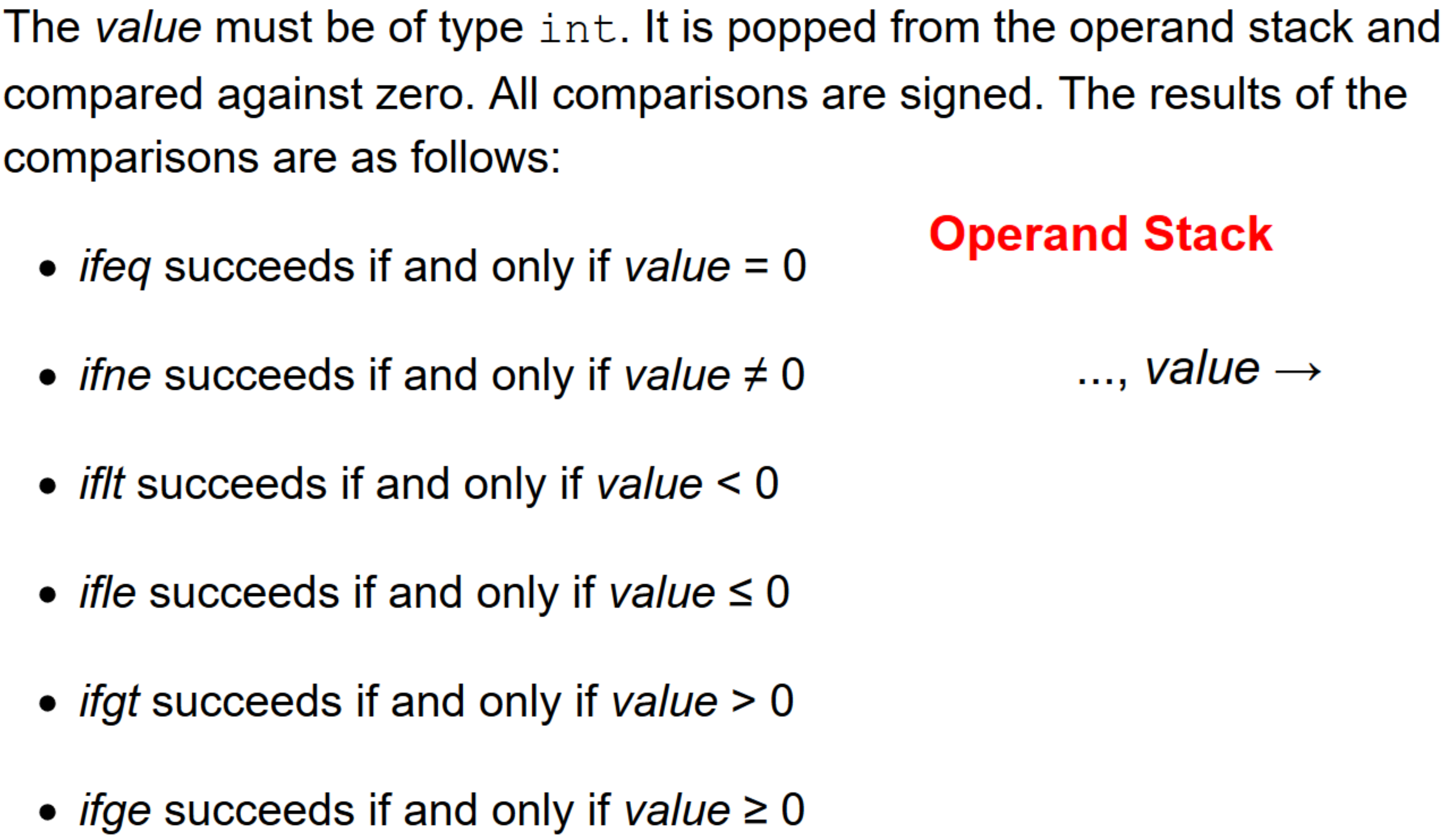
local variables (memory)

0	
1	
2	22.0
3	

operand stack (registers)

Description

Instruction: if<*cond*>   (ex: ifle)



The *value* must be of type `int`. It is popped from the operand stack and compared against zero. All comparisons are signed. The results of the comparisons are as follows:

- *ifeq* succeeds if and only if *value* = 0
- *ifne* succeeds if and only if *value* ≠ 0
- *iflt* succeeds if and only if *value* < 0
- *ifle* succeeds if and only if *value* ≤ 0
- *ifgt* succeeds if and only if *value* > 0
- *ifge* succeeds if and only if *value* ≥ 0

Operand Stack

..., *value* →

Please reference the following link for the description of all instructions.
https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1    ; a = 10
    goto END
ELSE:
     fload 2     ; load 22.0 into operand stack.
     fstore 1    ; a = b
END:
```

local variables (memory)

0	
1	
2	22.0
3	

operand stack (registers)

10.0

Example: compute3.j

```
    ldc 22.0f
    fstore 2
    fload 2
    ldc 0.0f
    fcmpl
    ifle ELSE
    ldc 10.0f
    fstore 1       ; a = 10
    goto END
ELSE:
     fload 2      ; load 22.0 into operand stack.
     fstore 1     ; a = b
END:
```

local variables (memory)

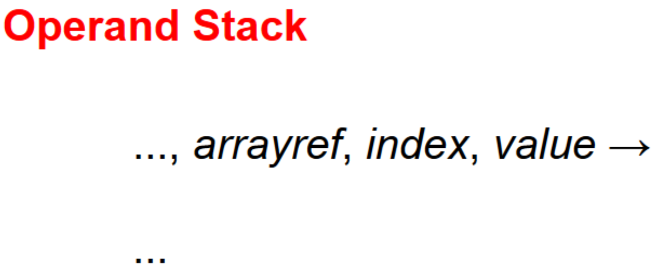
0	
1	10.0
2	22.0
3	

operand stack (registers)

Array Operation

- Construct a integer-array : **int x[] = new int[5]**
  - bipush 5 ; push 5 into stack.
  - newarry int ; construct integer array (size=5)
  - astore 6
- Read array element : **s = t[4]**
  - aload 6 ; push reference for array t.
  - bipush 4
  - iaload ; pop t and 4, and then push t[4].
  - istore 8

Array Operation

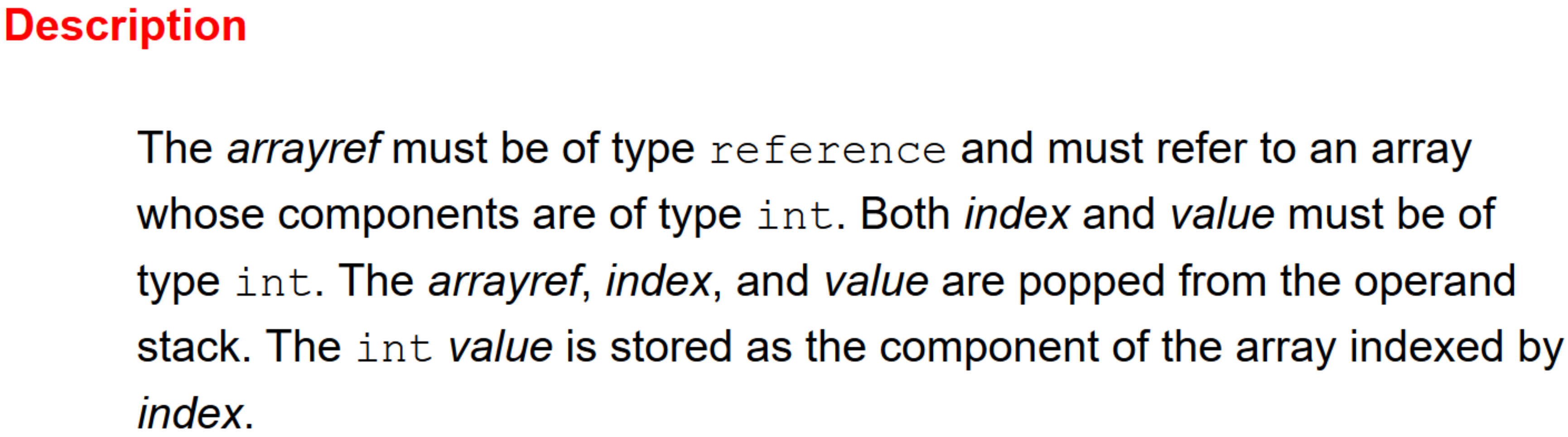


Operand Stack

..., *arrayref*, *index*, *value* →

...

- Assign array element : **s[5] = 10**
  - aload 6 ; push reference for s.
  - bipush 5 ; push 5
  - bipush 10 ; push 10
  - iastore ;



Description

The *arrayref* must be of type `reference` and must refer to an array whose components are of type `int`. Both *index* and *value* must be of type `int`. The *arrayref*, *index*, and *value* are popped from the operand stack. The `int` *value* is stored as the component of the array indexed by *index*.

透過反組譯觀察

```
$ javap -c  xxx.class
```

Reference

- https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html

- http://jasmin.sourceforge.net/

- http://jasmin.sourceforge.net/instructions.html

- http://www.ist.tugraz.at/_attach/Publish/Cb14/Jasmin.pdf

Backup

Switch-Case Construct

```
int i;
......
switch (i) {
case 1:
        return 1;
case 10:
        return 2;
case 100:
        return 3;
default:
        return 0;
}
```

```
iload i
lookupswitch
   1         : R1
   10        : R2
   100       : R3
   default   : R4
R1:
   iconst_1
   ireturn
R2:
   iconst_2
   ireturn
R3:
   iconst_3
   ireturn
R4:
   iconst_4
   ireturn
```