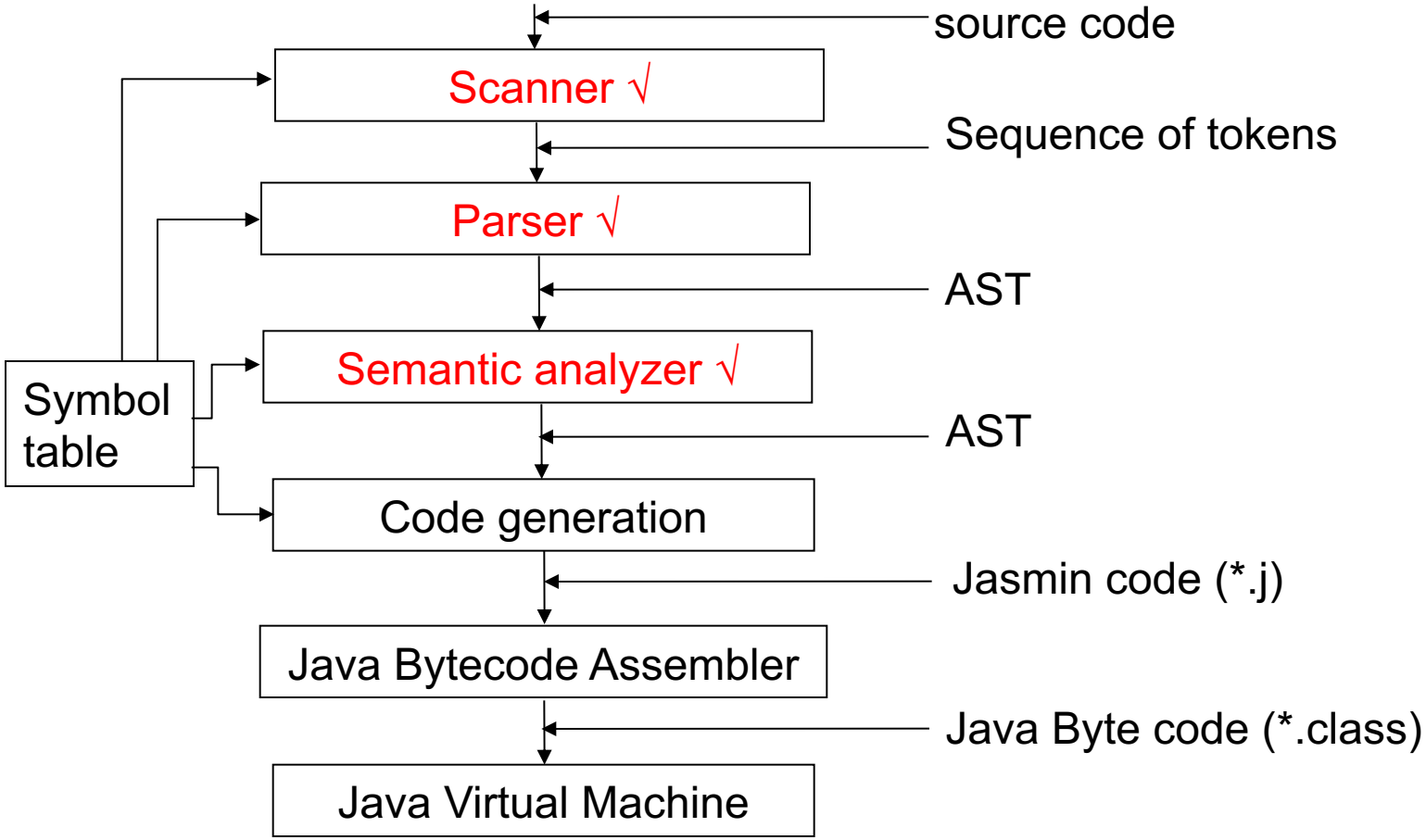
JVM and Jasmin

Dr. Nguyen Hua Phung
Faculty of CSE
HCMUT

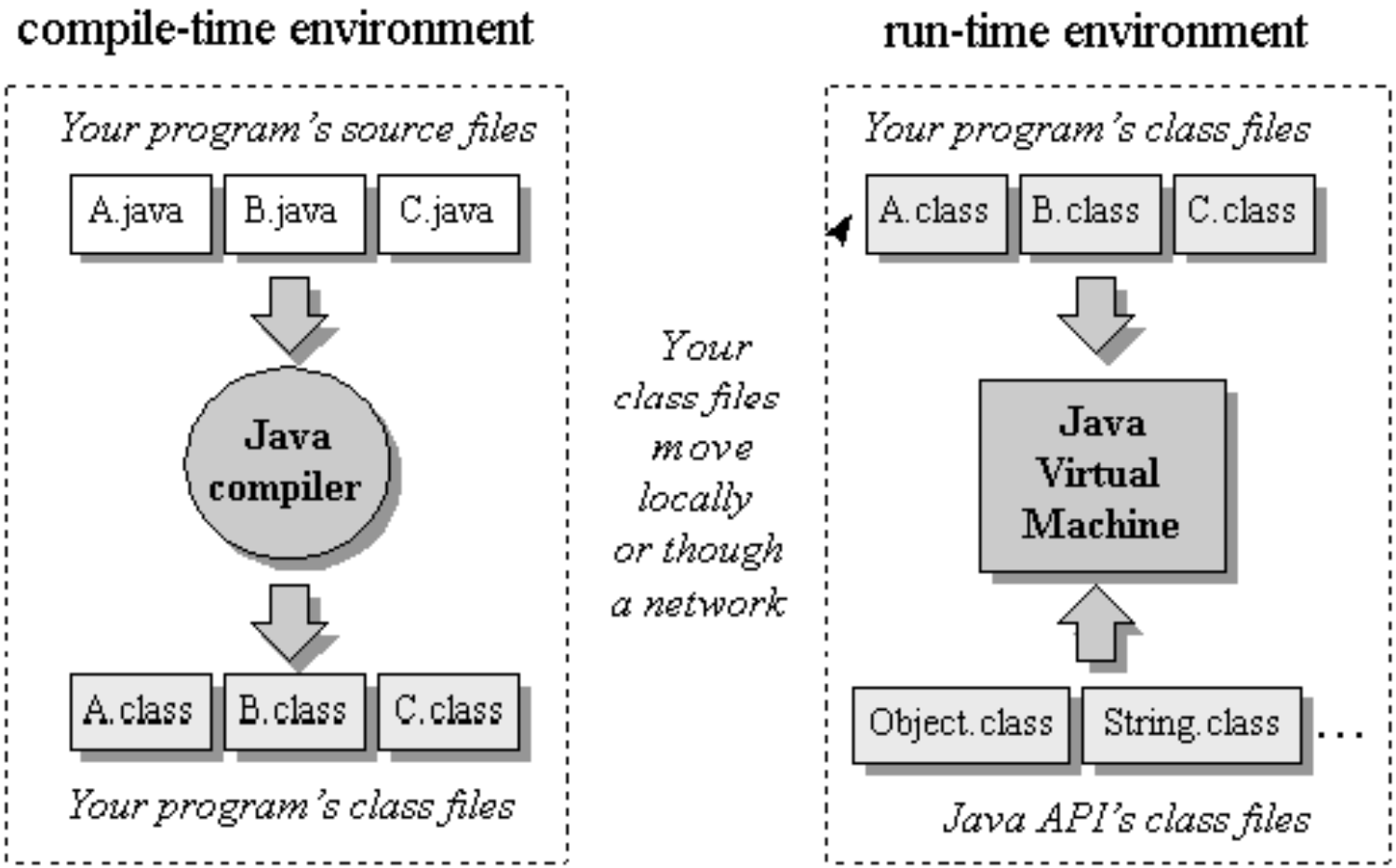
Outline

- Our compiler
- Java Virtual Machine
  - Data types
  - Operand stack
  - Local variable array
  - Instructions

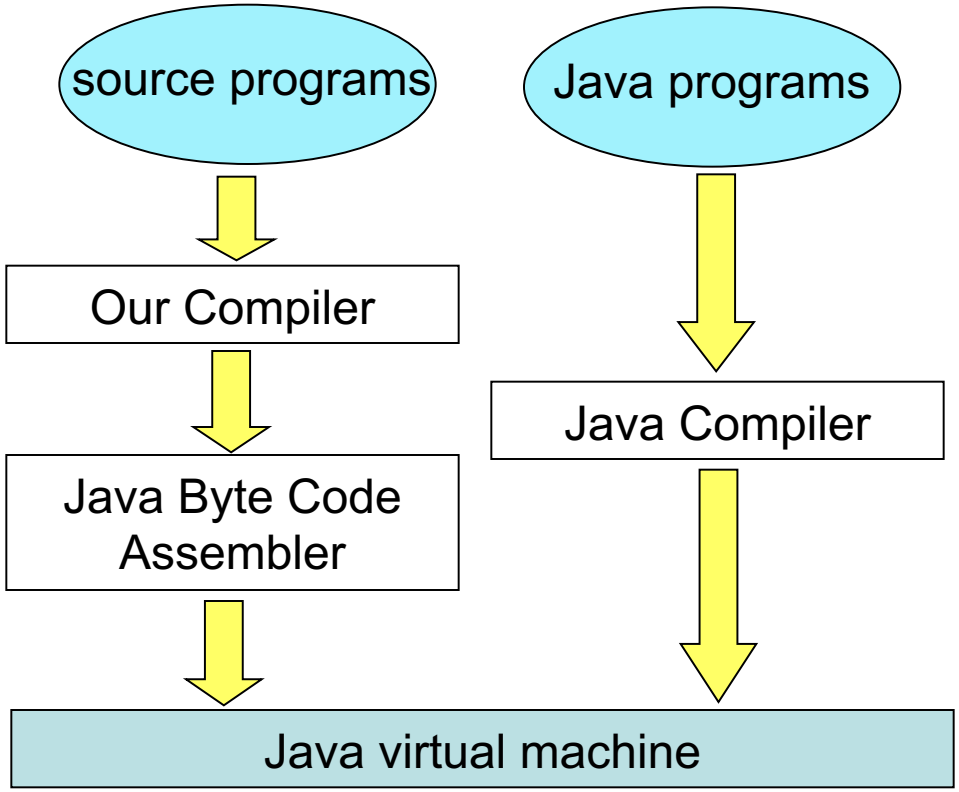
Our Compiler



Java Programming Environment



From [1]


source programs
Java programs
Our Compiler
Java Compiler
Java Byte Code Assembler
Java virtual machine

Why Jasmin ?

- Jasmin is a Java assembler
  - adopts a one-to-one mapping
  - operation codes are represented by mnemonic
  - Example:

```
public class VD {
    public void main(String[] args) {
      int a,b;
      b = 0;
      a = b * 2 + 40;
    }
}
```

```
.line 4
iconst_0
istore_2
.line 5
iload_2
iconst_2
imul
bipush 40
iadd
istore_1
```

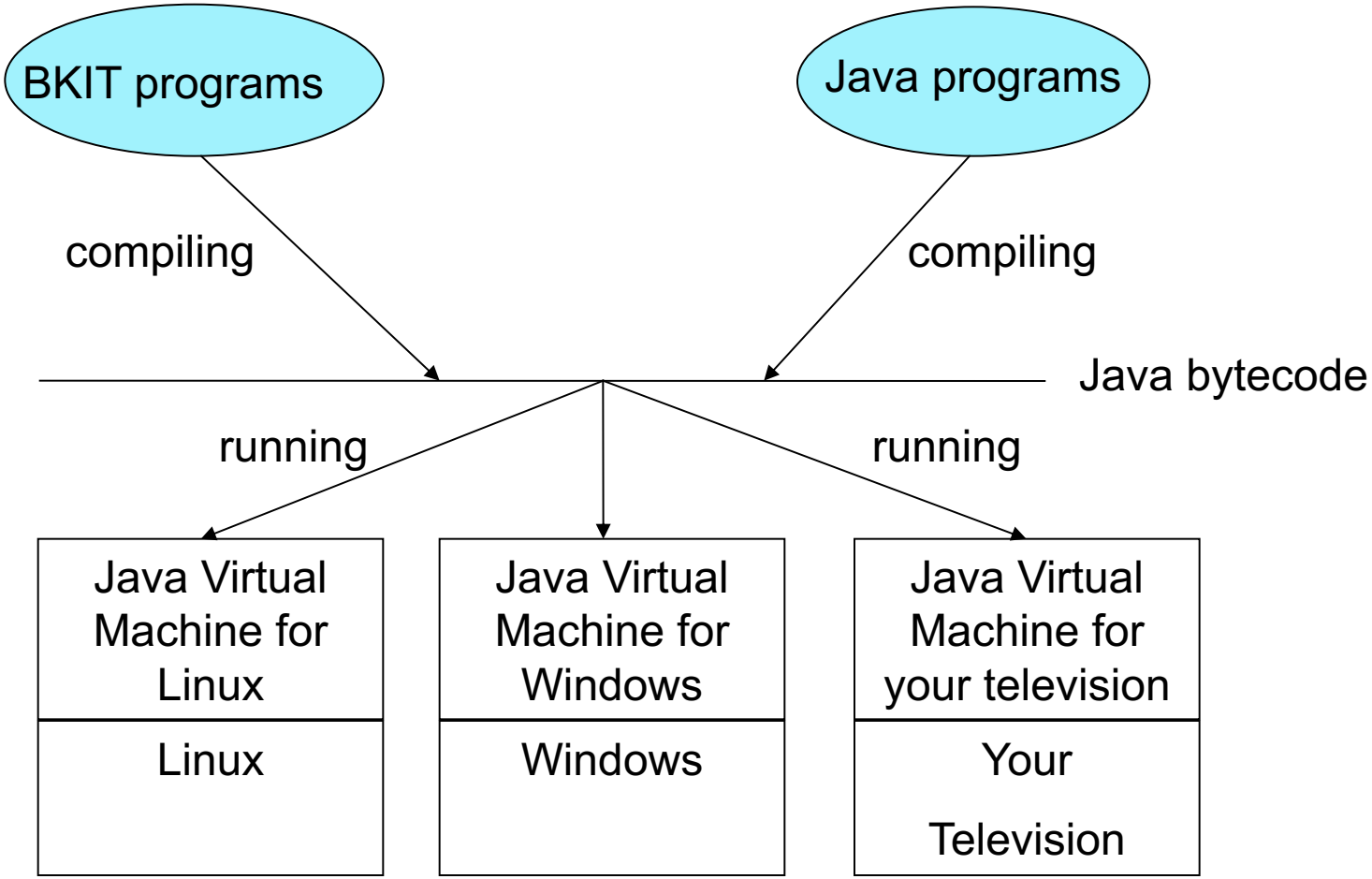
Java Byte Code

ca fe ba be 00 00 00 31 00 1b 0a 00 05 00 0e 09 00 0f 00 10 0a 00 11 00 12 07 00 13 07 00 14 01
00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e 75 6d 62
65 72 54 61 62 6c 65 01 00 04 6d 61 69 6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74
72 69 6e 67 3b 29 56 01 00 0a 53 6f 75 72 63 65 46 69 6c 65 01 00 07 56 44 2e 6a 61 76 61 0c 00
06 00 07 07 00 15 0c 00 16 00 17 07 00 18 0c 00 19 00 1a 01 00 02 56 44 01 00 10 6a 61 76 61 2f
6c 61 6e 67 2f 4f 62 6a 65 63 74 01 00 10 6a 61 76 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d 01 00
03 6f 75 74 01 00 15 4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 3b 01 00 13 6a
61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65 61 6d 01 00 07 70 72 69 6e 74 6c 6e 01 00 04 28
49 29 56 00 21 00 04 00 05 00 00 00 00 00 02 00 01 00 06 00 07 00 01 00 08 00 00 00 1d 00 01 00
01 00 00 00 05 2a b7 00 01 b1 00 00 00 01 00 09 00 00 00 06 00 01 00 00 00 01 00 09 00 0a 00 0b
00 01 00 08 00 00 00 35 00 02 00 03 00 00 00 11 03 3d 1c 05 68 10 28 60 3c b2 00 02 1b b6 00 03
b1 00 00 00 01 00 09 00 00 00 12 00 04 00 00 00 04 00 02 00 05 00 09 00 06 00 10 00 07 00 01 00
0c 00 00 00 02 00 0d

Outline

- Our compiler
- Java Virtual Machine
  - Data types
  - Operand stack
  - Local variable array
  - Instructions

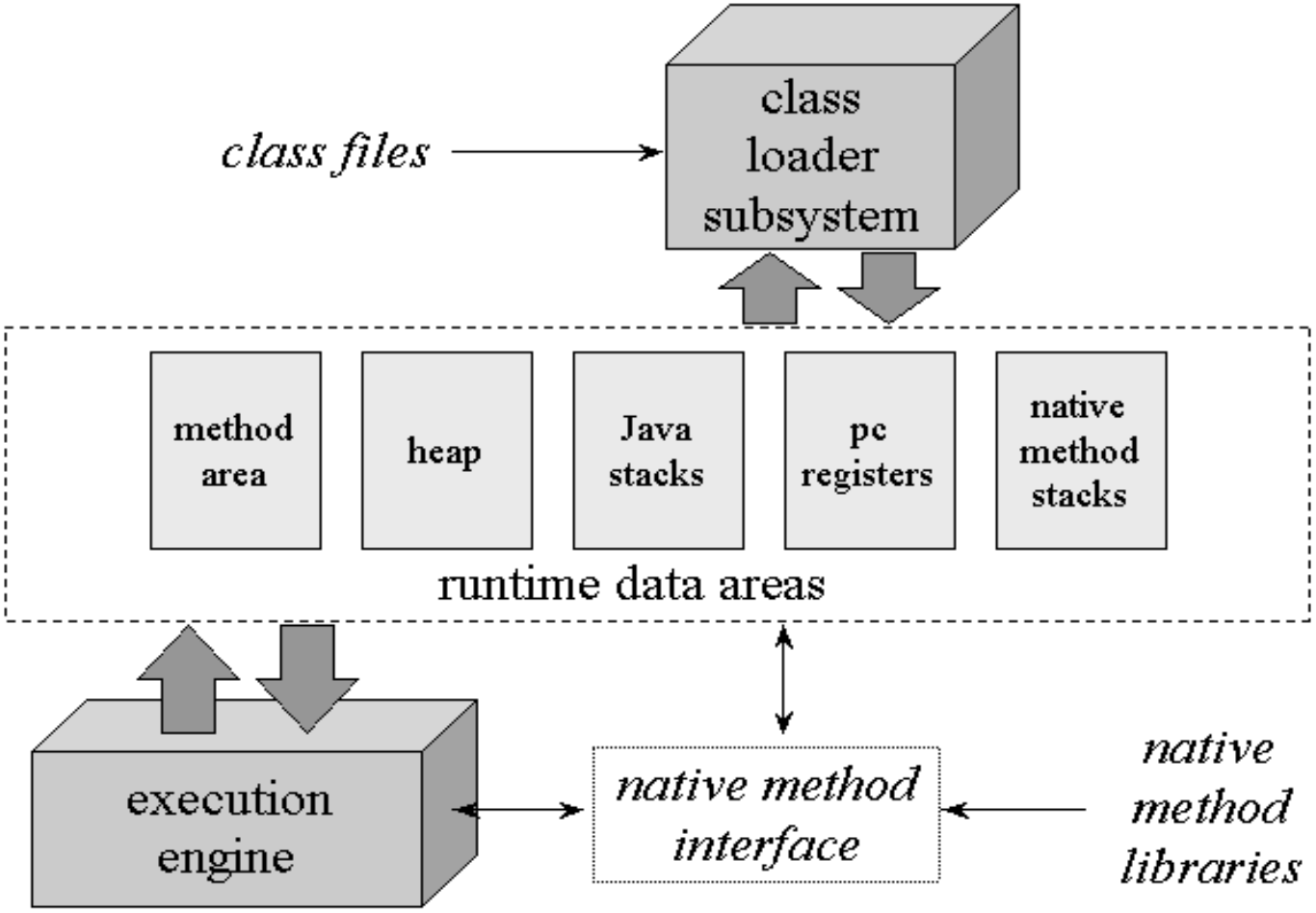
Java Virtual Machine



JVM = stack-based machine

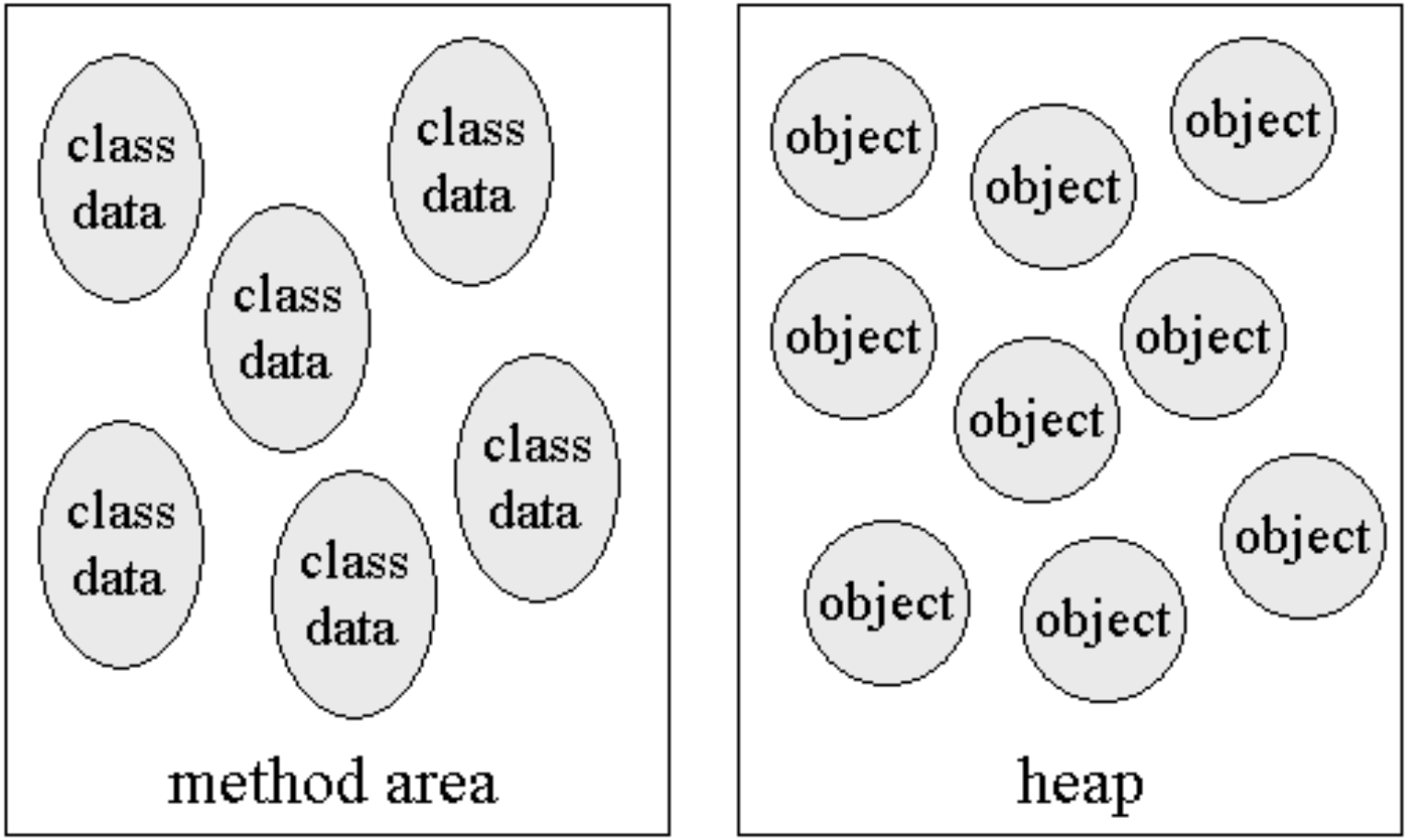
- A stack for each method
- The stack is used to store operands and results of an expression.
- It is also used to pass argument and receive returned value.
- Code generation for a stack-based machine is easier than that for a register-based one.

Internal Architecture of JVM



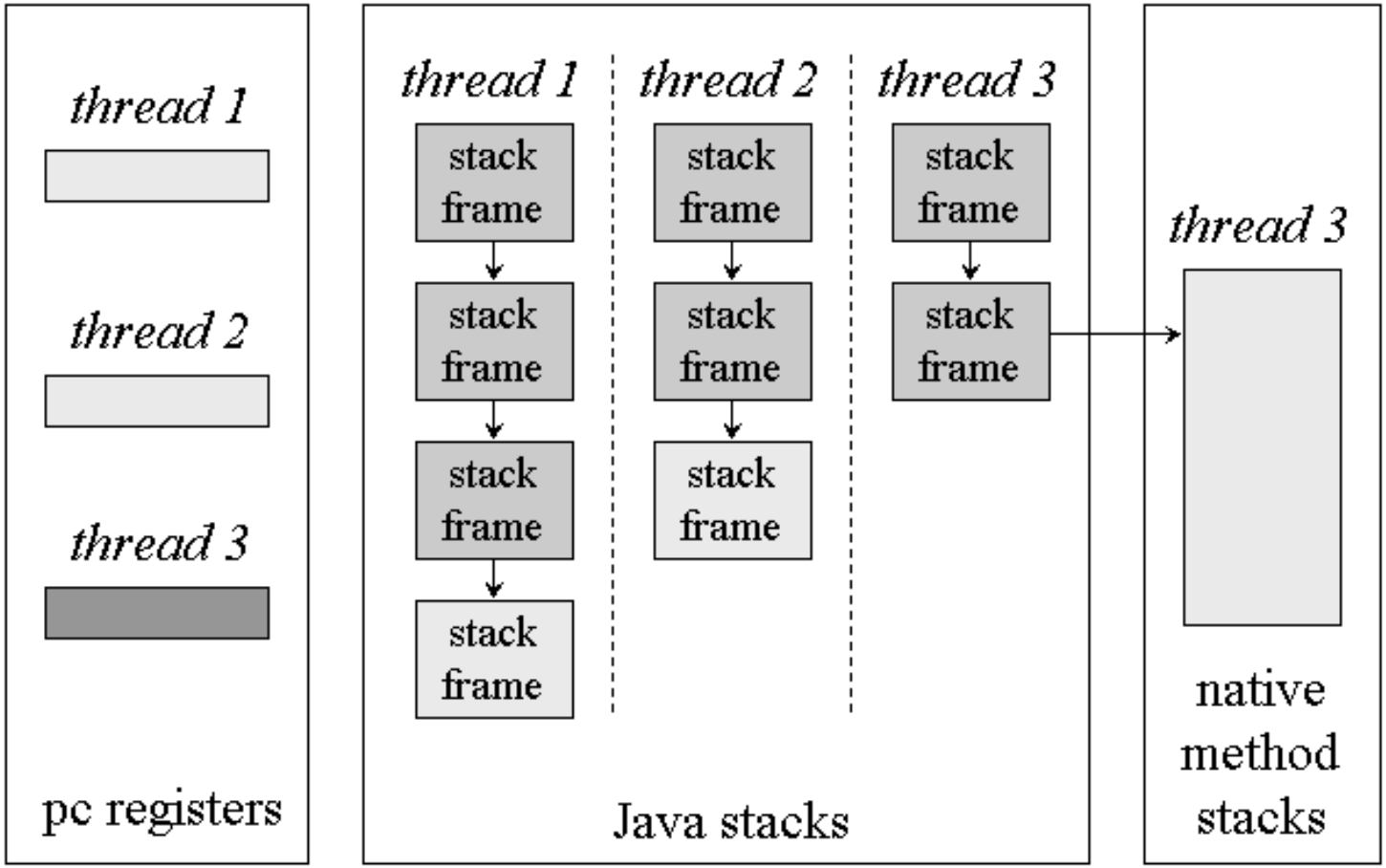
From [1]

Method Area and Heap



From [1]

Java Stacks



From [1]

Outline

- Our compiler
- Java Virtual Machine
  - Data types
  - Operand stack
  - Local variable array
  - Instructions

Data Types

Type	Range	Description
boolean	{0,1}	Z
byte	-2^7 to 2^7 - 1, inclusive	B
short	-2^{15} to 2^{15} – 1, inclusive	S
int	-2^{31} to 2^{31} – 1, inclusive	I
long	-2^{63} to 2^{63} – 1, inclusive	L
char	16 bit usigned Unicode (0 to 2^{16} -1)	C
float	32-bit IEEE 754 single-precision float	F
double	64-bit IEEE 754 double-precision float	D
returnAddress	address of an opcode within the same method	
class reference		Lclass-name;
interface reference		Linter-name;
array reference		[[..[component-type
void		V

Example

Java language type	JVM description
Object	Ljava/lang/Object;
String	Ljava/lang/String;
String []	[Ljava/lang/String;
int []	[I
float [] []	[[F
void main(String [] args)	([Ljava/lang/String;)V
int gcd(int a,int b)	(II)I
char foo(float a,Object b)	(FLjava/lang/Object;)C

Example (cont'd)

```
public class GetType {
    public static void main(String [] args) {
        Object a = new Object();
        int [] b = new int[10];
        float[][] c = new float[2][3];
        String d = "csds";
        System.out.println("The class name of a is "+ a.getClass());
        System.out.println("The class name of b is " + b.getClass());
        System.out.println("The class name of c is " + c.getClass());
        System.out.println("The class name of d is " + d.getClass());
    }
}
```

Example (cont'd)

- boolean, byte, char and short are implemented as int

```
public class IntTypes {
    public static void
        main(String argv[]) {
        boolean z = true;
        byte b = 1;
        short s = 2;
        char c = 'a';
    }
}
```

```
.method public static
main([Ljava/lang/String;)V
...
.line 3
iconst_1
istore_1
.line 4
iconst_1
istore_2
.line 5
iconst_2
istore_3
.line 6
bipush 97
istore_4
Label0:
.line 8
return
.end method
```

Outline

- Our compiler
- Java Virtual Machine
  - Data types
  - Operand stack
  - Local variable array
  - Instructions

Operand Stack

- Accessed by pushing and popping values
  - storing operands and receiving the operations' results
  - passing arguments and receiving method results
- Integral expression:

  a = b * 2 + 40;
- Jasmin code

```
iload_2      // load variable 2 onto op stack
iconst_2     // push constant 2 onto op stack
imul         // pop 2 values on top of stack, multiple them and push the result
                onto stack
bipush 4.    // push 40 onto stack
iadd         // pop 2 values on top of stack, calculate and push the result onto
                stack
istore_1     // pop the value on top of stack and assign it to variable 1
```

Outline

- Our compiler
- Java Virtual Machine
  - Data types
  - Operand stack
  - Local variable array
  - Instructions

Local Variable Array

1. A new local variable array is created each time a method is called
2. Local variables addressed by indexing, starting from 0
3. Instance methods:
   - slot 0 given to *this*
   - Parameters (if any) given consecutive indices, starting from 1
   - The indices allocated to the other variables in any order
4. Class methods:
   - Parameters (if any) given consecutive indices, starting from 0
   - The indices allocated to the other variables in any order
5. One slot can hold a value of boolean, byte, char, short, int, float,reference and returnAdrress
6. One pair of slots can hold a value of long and double

From [2]

Example 1

```
public static void foo() {
    int a,b,c;
    a = 1;
    b = 2;
    c = (a + b) * 3;
}
```

```
.line 7
iconst_1
istore_0        // a
.line 8
iconst_2
istore_1        // b
.line 9
iload_0
iload_1
iadd
iconst_3
imul
istore_2        // c
```

Example 2

```
public void foo() {
    int a,b,c;
    a = 1;
    b = 2;
    c = (a + b) * 3;
}
```

```
.var 0 is this LVD2; from Label0 to Label1
.line 7
iconst_1
istore_1            // a
.line 8
iconst_2
istore_2            // b
.line 9
iload_1
iload_2
iadd
iconst_3
imul
istore_3            // c
```

Example 3

```
public void foo() {
    int     a = 1;
    long b = 2;
    int c = 3;
    long d = (a + b) * c;
}
```

```
.line 6
iconst_1
istore_1          // a
.line 7
ldc2_w 2
lstore_2          // 2,3 for b
.line 8
iconst_3
istore 4          // c
.line 9
iload_1
i2l               // conversion
lload_2
ladd
iload 4
i2l               // conversion
lmul
lstore 5          // 5,6 for d
```

Outline

- Our compiler
- Java Virtual Machine
  - Data types
  - Operand stack
  - Local variable array
  - Instructions

Jasmin Instructions

1. Arithmetic Instructions
2. Load and store instructions
3. Control transfer instructions
4. Type conversion instructions
5. Operand stack management instructions
6. Object creation and manipulation
7. Method invocation instructions
8. Throwing instructions (not used)
9. Implementing **finally** (not used)
10. Synchronisation (not used)

Arithmetic Instructions

- Add: *iadd*, *ladd*, *fadd*, *dadd*.
- Subtract: *isub*, *lsub*, *fsub*, *dsub*.
- Multiply: *imul*, *lmul*, *fmul*, *dmul*.
- Divide: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Remainder: *irem*, *lrem*, *frem*, *drem*.
- Negate: *ineg*, *lneg*, *fneg*, *dneg*.
- Shift: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
- Bitwise OR: *ior*, *lor*.
- Bitwise AND: *iand*, *land*.
- Bitwise exclusive OR: *ixor*, *lxor*.
- Local variable increment: *iinc*.
- Comparison: *dcmpg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

From ($3.11.3,[3])

Load and Store

- Load a local variable onto the operand stack:

  iload, *iload_<n>*, ⇒ n:0..3, used for int, boolean, byte, char or short
  lload, *lload_<n>*, ⇒ n:0..3, used for long
  fload, *fload_<n>*, ⇒ n:0..3, used for float
  dload, *dload_<n>*, ⇒ n:0..3, used for double
  aload, *aload_<n>,* ⇒ n:0..3, used for a reference
  Taload. ⇒ T:b,s,i,l,f,d,c,a

  Bigger than 3 --> No underscore
  Difference:
  - iload: 2 bytes instruction
  - iload_<n>: 1 byte instruction

- Store a value from the operand stack into a local variable:

  istore, *istore_<n>*, □ n:0..3, used for int, boolean, byte, char or short
  lstore, *lstore_<n>*, □ n:0..3, used for long
  fstore, *fstore_<n>*, □ n:0..3, used for float
  dstore, *dstore_<n>*, □ n:0..3, used for double
  astore, *astore_<n>,* □ n:0..3, used for a reference and returnAddress
  Tastore. ⇒ T:b,s,i,l,f,d,c,a

From ($11.3.2,[3])

Load and Store (cont'd)

- Load a constant onto the operand stack:

  bipush, ⇒ for an integer constant from -2^7 to $2^7 - 1$
  sipush, ⇒ for an integer constant from -2^{15} to $2^{15} - 1$
  ldc, ⇒ for a constant that is an integer, float or a quoted string
  ldc_w,
  ldc2_w, ⇒ for a constant that is a long or a double
  aconst_null, ⇒ for a null
  iconst_m1, ⇒ for -1
  iconst_<i>, ⇒ for 0,…,5
  lconst_<l>, ⇒ for 0,1
  fconst_<f>, ⇒ for 0.0,1.0 and 2.0
  dconst_<d>. ⇒ for 0.0,1.0

From ($11.3.2,[3])

Example 4

```
int a = 1 ;
int b = 100;
int c = 1000;
int d = 40000;
int e = a * b + c – d;
```

```
.line 6
iconst_1
istore_1
.line 7
bipush 100
istore_2
.line 8
sipush 1000
istore_3
.line 9
ldc 40000
istore 4
```

```
.line 10
iload_1
iload_2
imul
iload_3
iadd
iload 4
isub
istore 5
```

Example 5

```
float a = 1.0F ;
float b = 2.0F;
float c = 3.0F;
float d = 4.0F;
float e = a * b + c - d;
```

```
.line 6
fconst_1
fstore_1
.line 7
fconst_2
fstore_2
.line 8
ldc 3.0
fstore_3
.line 9
ldc 4.0
fstore 4
```

```
.line 10
fload_1
fload_2
fmul
fload_3
fadd
fload 4
fsub
fstore 5
```

Example 6

a[0] = 100;

b = a[1];

```
.line 8
aload_0         // push address of array referred by a
iconst_0        // push 0
bipush 100      // push 100
iastore         // a[0] = 100

.line 9
aload_0         // push address of array referred by a
iconst_1        // push 1
iaload          // pop a and 1, push a[1]
istore_1        // store to b
```

Use iconst_0 from 0..5

Use bipush from -2^7-1 to 2^7

Use sipush if very large

Control Transfer Instructions

- Unconditional branch:

  goto, *goto_w*, *jsr*, *jsr_w*, *ret*.

- Conditional branch:

  ifeq, *iflt*, *ifle*, *ifne*, *ifgt*, *ifge*, ⇒ compare an integer to zero

  ifnull, *ifnonnull*, ⇒ compare a reference to null

  if_icmpeq, *if_icmpne*, *if_icmplt*, *if_icmpgt*, *if_icmple*, *if_icmpge*, ⇒ compare two integers

  if_acmpeq, *if_acmpne*. ⇒ compare two references

- Compound conditional branch:

  tableswitch, *lookupswitch*.

From ($3.11.7,[3])

Example 7

```
int a,b,c;
if (a > b)
    c = 1;
else
    c = 2;
```

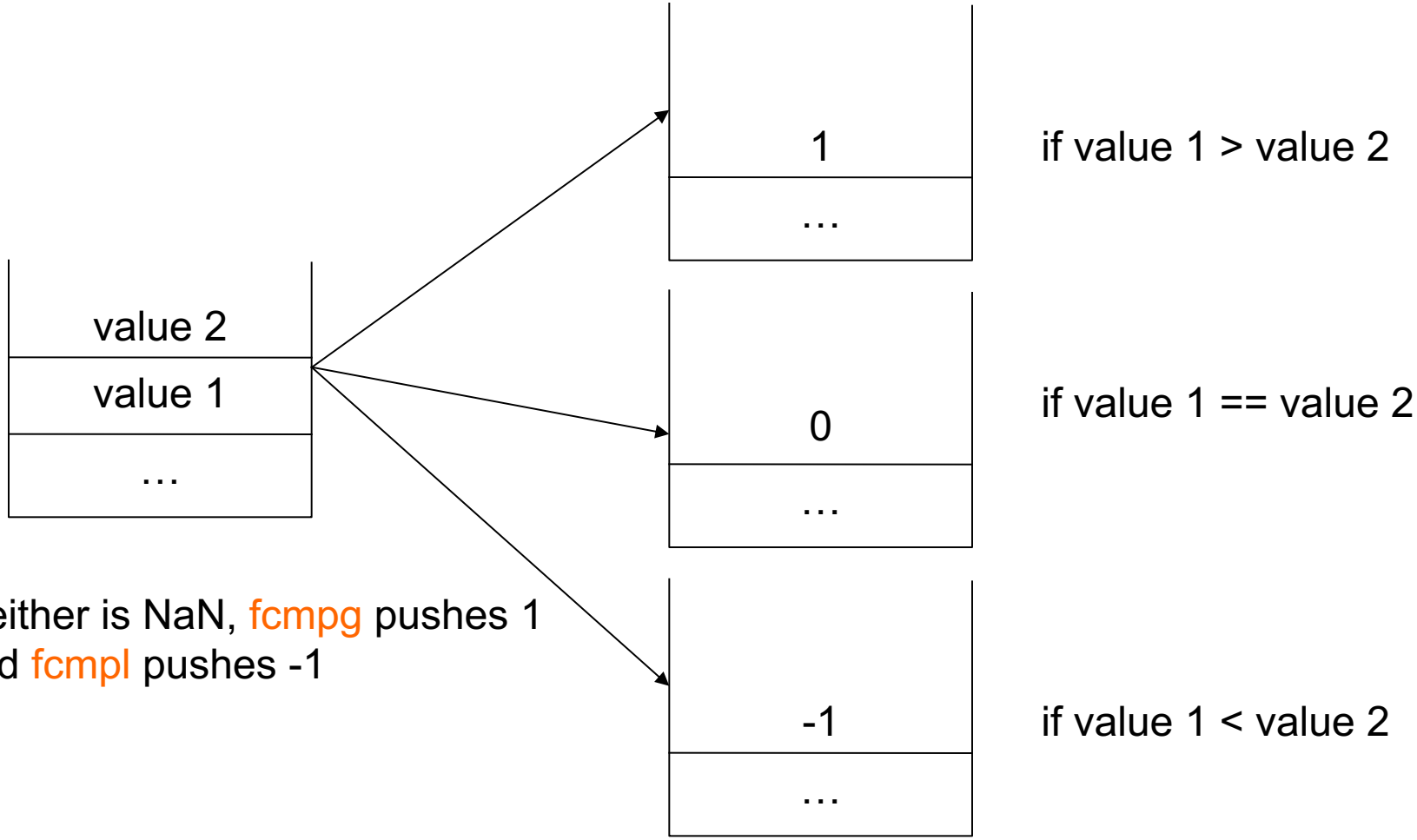
```
.line 7
iload_0          // push a
iload_1          // push b
if_icmple Label0
.line 8
iconst_1
istore_2         // c = 1
goto Label1
Label0:
.line 10
iconst_2
istore_2         // c = 2
Label1:
```

Example 8

```
float a,b; int c;
if (a > b)
    c = 1;
else
    c = 2;
```

```
.line 7
    fload_0          // push a
    fload_1          // push b
    fcmpl            // pop a,b, push 1 if a > b, 0 otherwise
    ifle Label0      // goto Label0 if top <= 0
.line 8
    iconst_1
    istore_2
    goto Label1
Label0:
.line 10
    iconst_2
    istore_2
Label1:
```

fcmpg and fcmpl



If either is NaN, fcmpg pushes 1 and fcmpl pushes -1

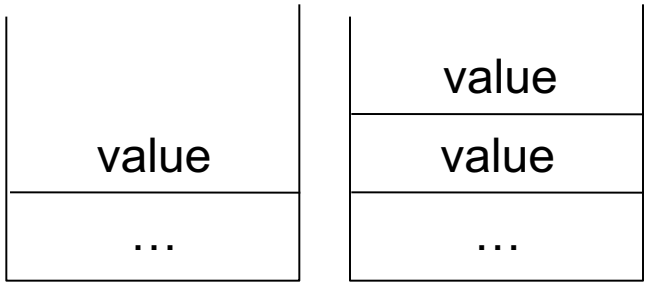
Type Conversion Instructions

- *i2l*, *i2f*, *i2d*, *l2f*, *l2d*, and *f2d*.
- Only *i2f* is used in MP compiler

```
int b = 1;        .line 6
                  iconst_1
                  istore_0
                  .line 7
float a = b;      iload_0
                  i2f
                  fstore_1
```

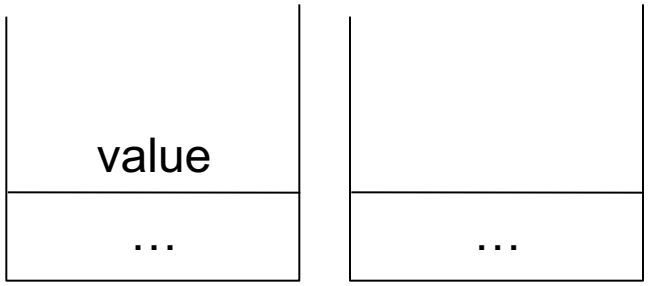
Operand Stack Management Instructions

- dup duplicate the stack top operand
- pop remove the stack top operand



dup

used when translating a = b = …



pop

used when translating 1;

- others: pop2, dup2, swap,…

Example 10

```
int a,b,c;
a = b = c = 1;
```

```
.line 7
iconst_1
dup
istore_2
dup
istore_1
istore_0
```

Tạo const = 1
Dup xong store vào c
Dup tiếp xong store vào b
Còn dư 1 cái store vào a

```
int a,b,c;
1 + (a = 2);
```

In MC, not in Java

```
.line 7
iconst_1
iconst_2
dup
istore_0
iadd
pop
```

Object Creation and Manipulation

- Create a new class instance: *new*.
- Create a new array: *newarray*, *anewarray*, *multianewarray*.
- Access fields of classes (static fields, known as class variables) and fields of class instances (non-static fields, known as instance variables): *getfield*, *putfield*, *getstatic*, *putstatic*.
- Load an array component onto the operand stack: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Store a value from the operand stack as an array component: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- …

Example 11

BKOOL:

a:integer[10];

a[0] = a[1] + 2;

Java:

int a[] = new int [10];

a[0] = a[1] + 2;

```
.line 6
bipush 10
newarray int
astore_0
.line 7
aload_0
iconst_0
aload_0
iconst_1
iaload
iconst_2
iadd
iastore
```

Field Instructions

- getstatic
- pustatic
- getfield
- putfield

} <field_spec> <descriptor>

- E.g.

getstatic $\underbrace{\text{java.lang.System}}_{\text{class name}}$.out $\underbrace{\text{Ljava/io/PrintStream;}}_{\text{field type}}$

field name

Example 12

```
public class VD12 {
    static int a;
    int b;
    static VD12 c;
    VD12 d;
    public static void
            main(String[] arg) {
        VD12 e;
        e = new VD12();
        a = 1;
        e.b = a + 1;
        e.d = new VD12();
        c = e.d;
    }
}
```

```
new VD12
dup
invokespecial VD12/<init>()V
astore_1
iconst_1
putstatic VD12.a I
aload_1
getstatic VD12.a I
iconst_1
iadd
putfield VD12.b I
aload_1
new VD12
dup
invokespecial VD12/<init>()V
putfield VD12.d LVD12;
aload_1
getfield VD12.d LVD12;
putstatic VD12.c LVD12;
```

Method Invocation Instructions

- invokestatic
- invokevirtual
- invokespecial

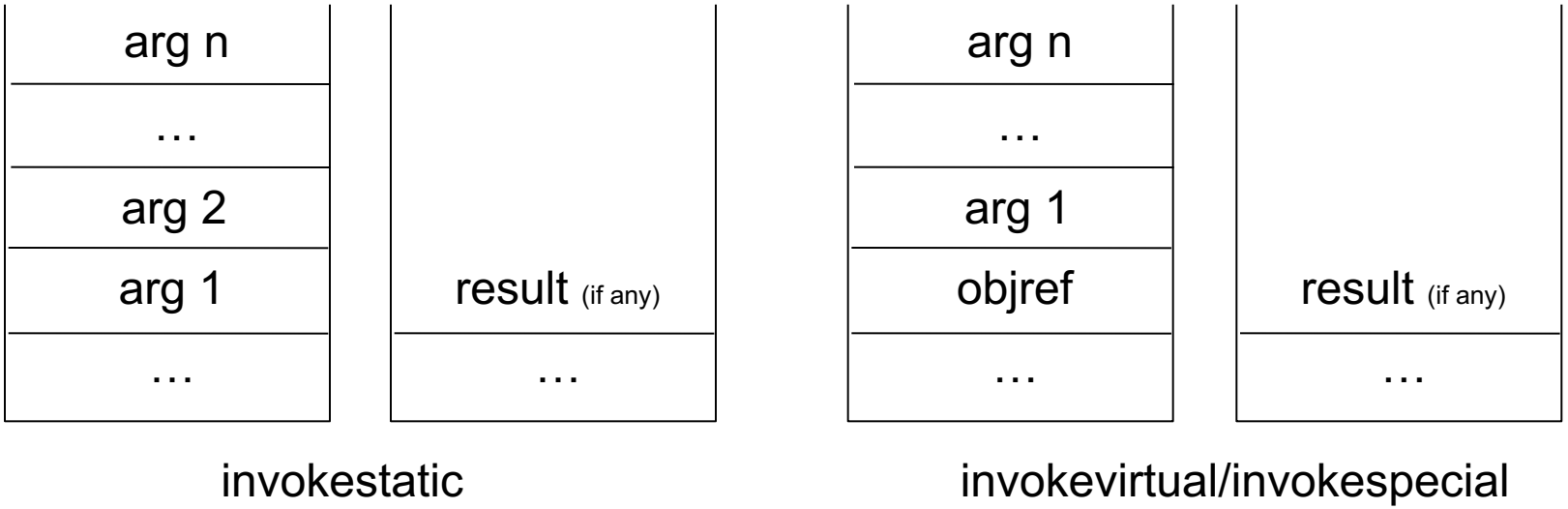
  } <method-spec>

  - the constructor method <init>
  - a private method
  - a method in a super class
- invokeinterface <method-spec> <num-args>

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

class name — method name — type desc

Method Invocation Instructions (cont'd)



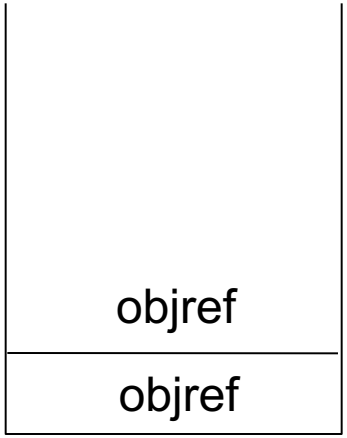
- invokevirtual: based on the real type of objref
- invokestatic: based on the static class

Example 13

```
public class VD13 {
    public static void main(String[] arg) {
        goo(new VD13());
    }
    float foo(int a, float b) {
        return a + b;
    }
    static void goo(VD13 x){
        x.foo(1,2.3F);
    }
}
```

Example 13 (cont'd)

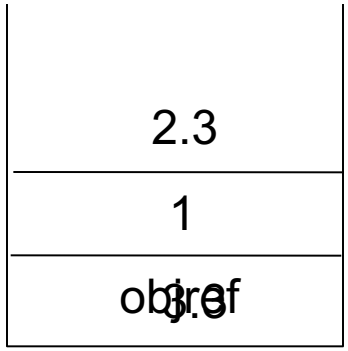
```
public static void  main(String[] arg) {
        goo(new VD13());
}
```




```
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.var 0 is arg0 [Ljava/lang/String; from Label0 to Label1

.line 3
      new VD13
      dup
      invokespecial VD13/<init>()V
      invokestatic VD13/goo(LVD13;)V

.line 4
      return

.end method
```

Example 13 (cont'd)

```
static void goo(VD13 x) {
x.foo(1,2.3F);
}
```



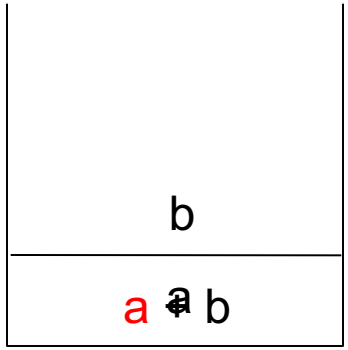

```
.method static goo(LVD13;)V
.limit stack 3
.limit locals 1
.var 0 is arg0 LVD13; from Label0 to Label1

.line 9
    aload_0
    iconst_1
    ldc 2.3
    invokevirtual VD13/foo(IF)F
    pop
Label1:
.line 10
    return

.end method
```

Method Return

- All methods in Java are terminated by a return instruction
  - return void
  - ireturn int,short,char,boolean, byte
  - freturn float
  - lreturn long
  - dreturn double
  - areturn reference

Example 13 (cont'd)

```
float foo(int a, float b) {
    return a + b;
}
```




```
.method  foo(IF)F
.limit stack 2
.limit locals 3
.var 0 is this LVD13; from Label0 to Label1
.var 1 is arg0 I from Label0 to Label1
.var 2 is arg1 F from Label0 to Label1

Label0:
iload_1
i2f
fload_2
fadd
Label1:
freturn

.end method
```

Jasmin Directives

- .source <source.java>
- .class <the current class>
- .super <the super class>
- .limit
- .method <the method description>
- .field <the field description>
- .end
- .var <the variable description>
- .line <the line number in source code>

Example 14

```
public class VD14 {
    int a;
    static int b;

    public static void
            main(String[] arg) {
            (new
    VD14()).foo(1,2.3F);
    }

    float foo(int a, float b) {
            return a * b;
    }

}
```

```
.source VD14.java
.class public VD14
.super java/lang/Object

.field  a I
.field static b I
```

Example 14 (cont'd)

```
public class VD14 {
    int a;
    static int b;

    public static void
            main(String[] arg) {
            (new
    VD14()).foo(1,2.3F);
    }

    float foo(int a, float b) {
            return a * b;
    }

}
```

```
.method public <init>()V
.limit stack 1
.limit locals 1
.var 0 is this LVD14; from Label0 to Label1

Label0:
.line 1
    aload_0
    invokespecial java/lang/Object/<init>()V
Label1:
    return

.end method
```

Example 14 (cont'd)

```
public class VD14 {
    int a;
    static int b;

    public static void
            main(String[] arg) {
            (new
    VD14()).foo(1,2.3F);
    }

    float foo(int a, float b) {
            return a * b;
    }

}
```

```
.method public static main([Ljava/lang/String;)V
.limit stack 3
.limit locals 1
.var 0 is arg0 [Ljava/lang/String; from Label0 to
Label1

Label0:
.line 5
new VD14
dup
invokespecial VD14/<init>()V
iconst_1
ldc 2.3
invokevirtual VD14/foo(IF)F
pop
Label1:
.line 6
return

.end method
```

Example 14 (cont'd)

```
public class VD14 {
    int a;
    static int b;

    public static void
            main(String[] arg) {
            (new
    VD14()).foo(1,2.3F);
    }

    float foo(int a, float b) {
            return a * b;
    }

}
```

```
.method  foo(IF)F
.limit stack 2
.limit locals 3
.var 0 is this LVD14; from Label0 to Label1
.var 1 is arg0 I from Label0 to Label1
.var 2 is arg1 F from Label0 to Label1

Label0:
.line 8
iload_1
i2f
fload_2
fmul
Label1:
freturn

.end method
```

References

[1] Bill Venner, Inside the Java Virtual Machine, http://www.artima.com/insidejvm/ed2/

[2] J.Xue, Prog. Lang. and Compiler, http://www.cse.unsw.edu.au/~cs3131

[3] Java Virtual Machine Specification, http://java.sun.com/docs/books/vmspec/

[4] Jasmin Home Page, http://jasmin.sourceforge.net/