JVM and Jasmin

Dr. Nguyen Hua Phung
Faculty of CSE
HCMUT

Java Bytecode:

Nó là cái dạng mã mà Java Compiler (cái chương trình dịch code Java của mày) tạo ra sau khi mày viết xong code *.java.
Đây không phải là mã máy trực tiếp mà CPU của mày hiểu được.
Nó là một dạng mã trung gian, được thiết kế đặc biệt để chạy trên Java Virtual Machine (JVM).
Giống như "ngôn ngữ chung" mà mọi JVM đều hiểu.
Java Virtual Machine (JVM):

Đây là một máy ảo (một chương trình máy tính mô phỏng một máy tính khác) chạy trên hệ điều hành của mày (Windows, L inux, macOS,...).
Nó có nhiệm vụ thực thi cái Java bytecode mà compiler tạo ra.
JVM sẽ phiên dịch bytecode thành mã máy cụ thể cho hệ điều hành đang chạy, rồi đưa cho CPU thực hiện.
Chính vì có JVM ở giữa nên chương trình Java mới có thể chạy trên nhiều hệ điều hành khác nhau mà không cần phải viết lại code (tính platform independence mà hình cuối mình xem đó).
Mối liên hệ:

Mày viết code Java (*.java).
Java Compiler dịch cái code đó thành Java bytecode (*.class).
Java Virtual Machine (JVM) sẽ đọc và chạy cái file *.class chứa bytecode đó.

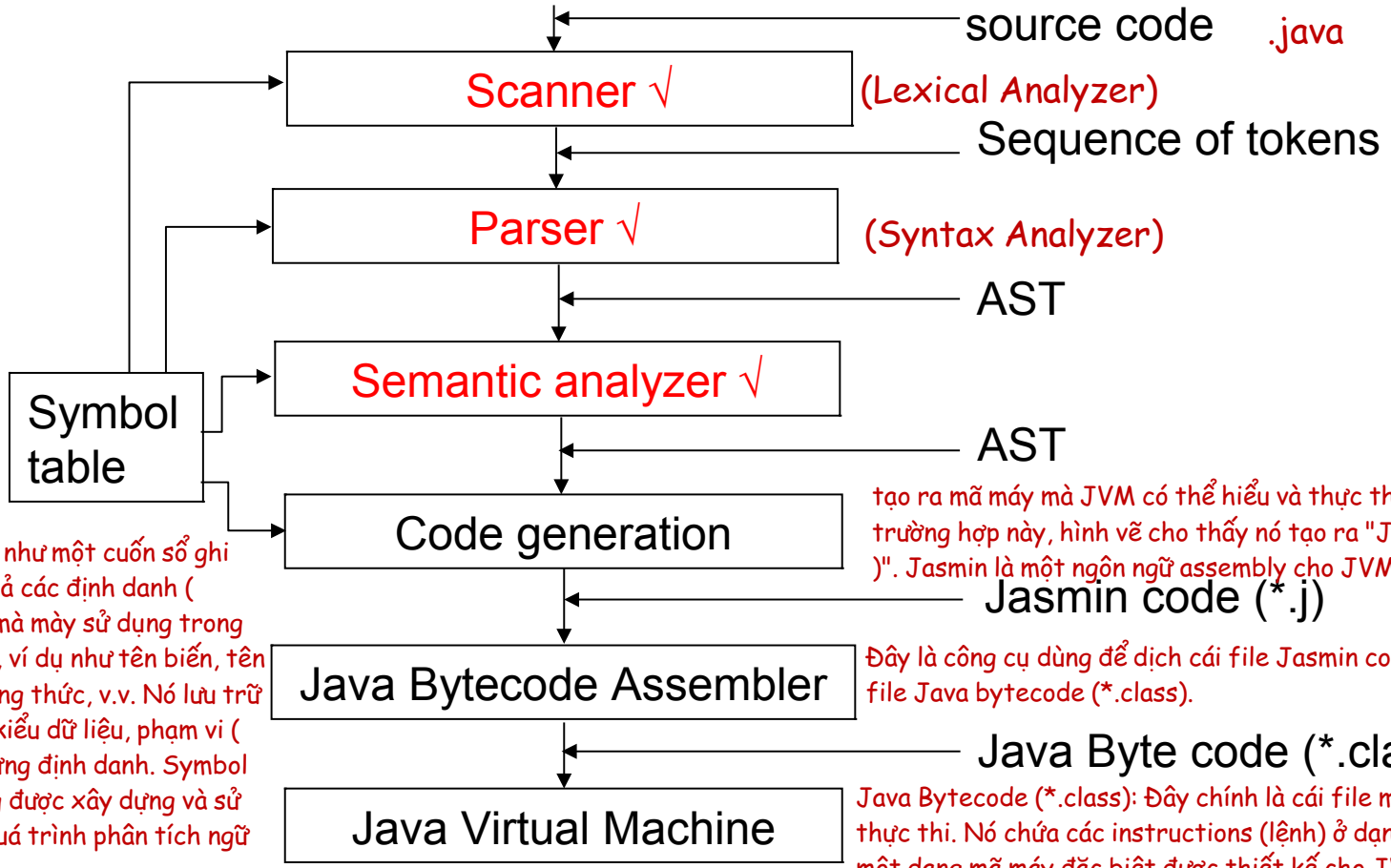
Liên quan:
Jasmin: Là một công cụ giúp mày viết bytecode một cách "thủ công" bằng các câu lệnh dễ hiểu hơn (mnemonic). Sau đó, Jasmin assembler sẽ chuyển nó thành bytecode thực sự. Thường thì mình không cần dùng trực tiếp Jasmin, compiler sẽ lo hết.
Compile-time environment: Là cái lúc mà code Java của mày được biên dịch ra bytecode.
Run-time environment: Là cái lúc mà JVM chạy cái bytecode đó.

Hiểu nôm na là mày viết tiếng người (Java code), compiler dịch sang "tiếng máy ảo" (bytecode), rồi JVM sẽ phiên dịch cái "tiếng máy ảo" đó ra "tiếng máy thật" (mã máy của CPU) để máy tính chạy.

Outline

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

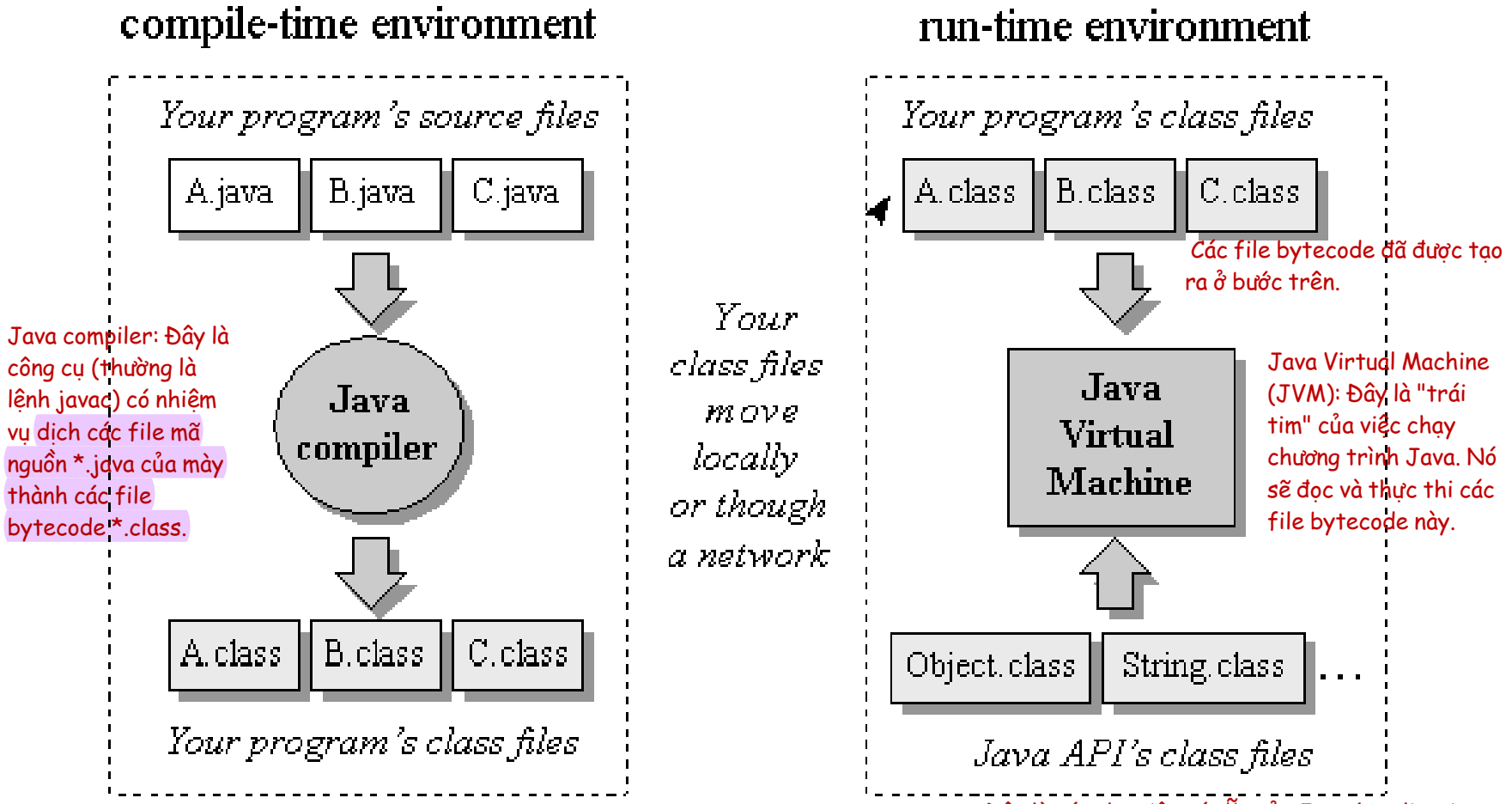
Our Compiler



Mối liên hệ giữa hai giai đoạn:
Giai đoạn biên dịch tạo ra các file *.class. Các file này sau đó sẽ được JVM sử dụng trong giai đoạn thực thi. Có thể hiểu đơn giản là mày "dịch" code một lần, rồi sau đó có thể "chạy" cái bản dịch đó nhiều lần trên bất kỳ máy nào có JVM.

Java Programming Environment

Ở giai đoạn này, mày viết code, sau đó dùng compiler để biến code đó thành bytecode.



Java compiler: Đây là công cụ (thường là lệnh javac) có nhiệm vụ dịch các file mã nguồn *.java của mày thành các file bytecode *.class.

Các file bytecode đã được tạo ra ở bước trên.

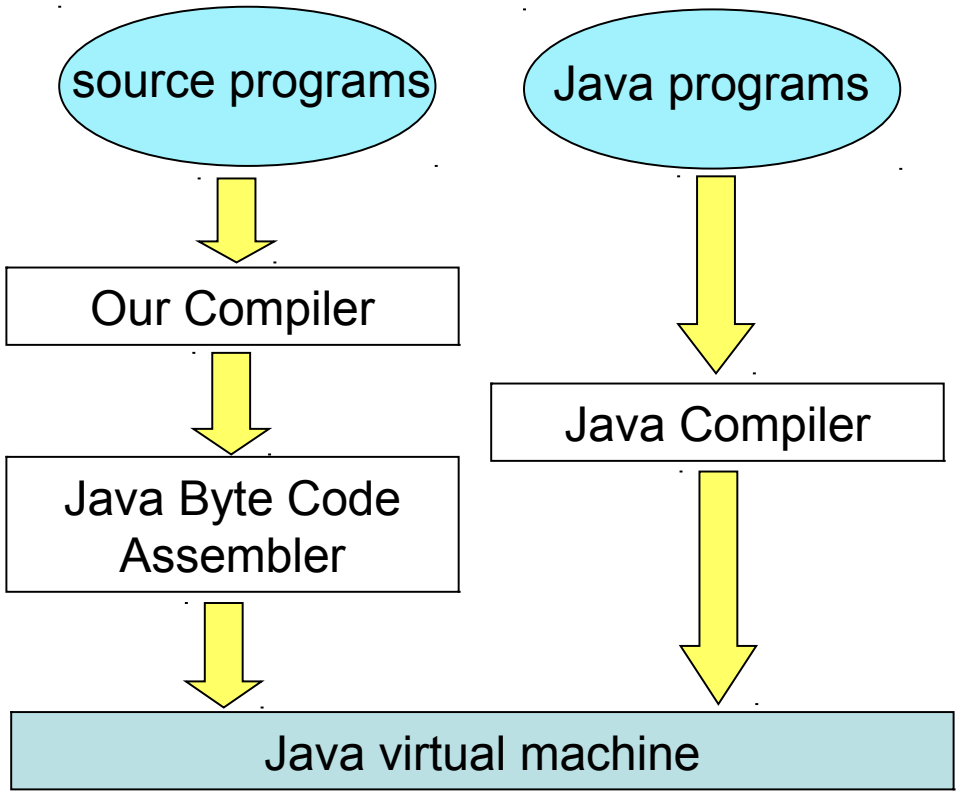
Java Virtual Machine (JVM): Đây là "trái tim" của việc chạy chương trình Java. Nó sẽ đọc và thực thi các file bytecode này.

Đây là các thư viện có sẵn của Java (Application Programming Interface). Khi chương trình của mày cần sử dụng các chức năng có sẵn (ví dụ như làm việc với chuỗi, tạo đối tượng...), JVM sẽ tìm đến các file class này để thực hiện.

From [1]

Cột bên trái thể hiện một quy trình mà một ngôn ngữ khác (không phải Java) có thể được biên dịch và chạy trên JVM thông qua một trình biên dịch riêng và một công cụ assembler để tạo ra bytecode. Ví dụ, các ngôn ngữ như Kotlin, Scala, Groovy cũng có thể được biên dịch thành bytecode và chạy trên JVM.

Cột bên phải thì thể hiện quy trình thông thường khi mày viết chương trình bằng Java: dùng Java Compiler để dịch mã nguồn *.java trực tiếp thành bytecode mà JVM có thể hiểu.



Why Jasmin ?

Jasmin là một Java assembler: Cái này có nghĩa là Jasmin là một công cụ để viết mã bytecode của JVM một cách trực tiếp, nhưng nó dễ đọc và dễ viết hơn là bytecode "thô" (Giống như assembly language cho các loại chip xử lý thông thường).

Adopts a one-to-one mapping: Có nghĩa là gần như mỗi dòng lệnh trong Jasmin sẽ tương ứng với một lệnh bytecode duy nhất. Điều này giúp mày có thể kiểm soát rất chi tiết những gì sẽ xảy ra trên JVM.

Operation codes are represented by mnemonic: Thay vì phải nhớ các con số (opcode) của từng lệnh bytecode, Jasmin sử dụng các từ gợi nhớ (mnemonic) dễ hiểu hơn. Ví dụ, thay vì phải nhớ opcode cho việc load một biến integer là một con số nào đó, mày có thể viết iload.

- 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
```

lấy biến có index = 2

Giải thích ví dụ:

.line 4 và .line 5 chỉ dòng tương ứng trong file Java gốc.
iconst_0: Đẩy giá trị 0 (integer constant) lên stack.
istore_2: Lấy giá trị từ stack và lưu vào biến cục bộ có index là 2 (trong ví dụ này có thể là biến b).
iload 2: Lấy giá trị từ biến cục bộ có index là 2 (biến b) và đẩy lên stack.
iconst_2: Đẩy giá trị 2 lên stack.
imul: Lấy hai giá trị trên stack ra, nhân chúng lại, và đẩy kết quả trở lại stack.
bipush 40: Đẩy giá trị 40 (byte constant) lên stack.
iadd: Lấy hai giá trị trên stack ra, cộng chúng lại, và đẩy kết quả trở lại stack.
istore_1: Lấy giá trị từ stack và lưu vào biến cục bộ có index là 1 (trong ví dụ này có thể là biến a).

Java Byte Code

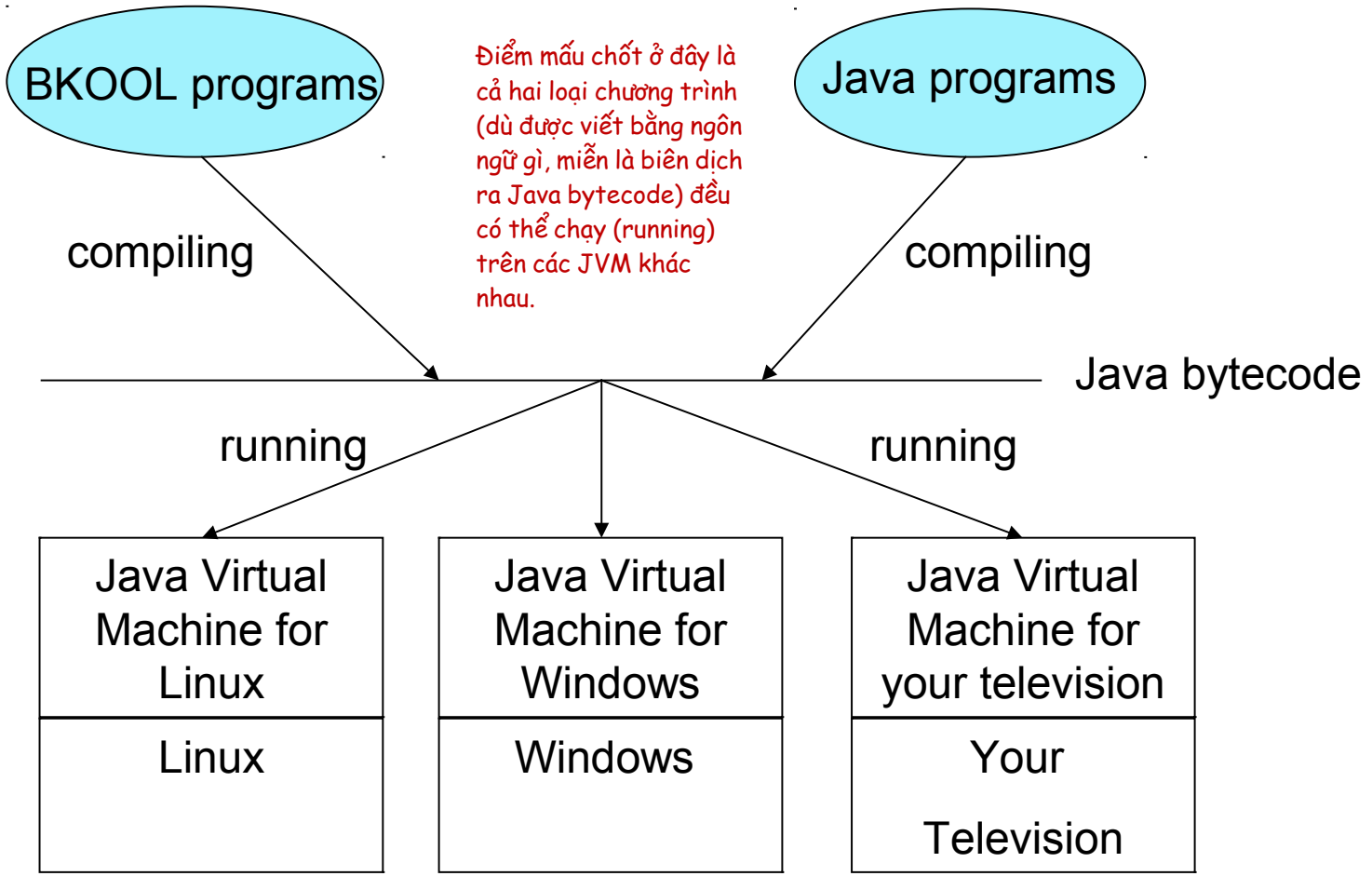
ca fe ba be 00 00 00 31 00 1b 0a 00 05 00 0e 09 00 0f 00 10 0a 00 11 00 12 07 00 13 07 00 14 01
00 06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f 64 65 01 00 0f 4c 69 6e 65 4e 75 6d 62
65 72 54 61 62 6c 65 01 00 04 6d 61 69 6e 01 00 16 28 5b 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74
72 69 6e 67 3b 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

Platform Independence

tính độc lập của nền tảng (Platform Independence) của Java.



Ý nghĩa của nó là gì?

Một khi chương trình của mày (dù là viết bằng Java hay một ngôn ngữ khác mà biên dịch ra bytecode) đã được biên dịch thành bytecode, thì nó có thể chạy trên bất kỳ thiết bị nào có cài đặt JVM phù hợp, mà không cần phải biên dịch lại cho từng hệ điều hành hay thiết bị cụ thể.

Đây chính là ý nghĩa của câu slogan nổi tiếng của Java: "Write Once, Run Anywhere" (Viết một lần, chạy ở mọi nơi).

JVM đóng vai trò như một lớp trừu tượng giữa chương trình của mày và hệ điều hành bên dưới. Chương trình của mày "nói chuyện" với JVM thông qua bytecode, và JVM sẽ lo phần còn lại để giao tiếp với hệ điều hành cụ thể mà nó đang chạy.

Vậy nên, dù mày code trên Windows, khi biên dịch ra bytecode, mày có thể mang file bytecode đó sang chạy trên máy Linux hoặc thậm chí là một cái TV thông minh có hỗ trợ JVM mà không cần phải sửa đổi hay biên dịch lại.

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.

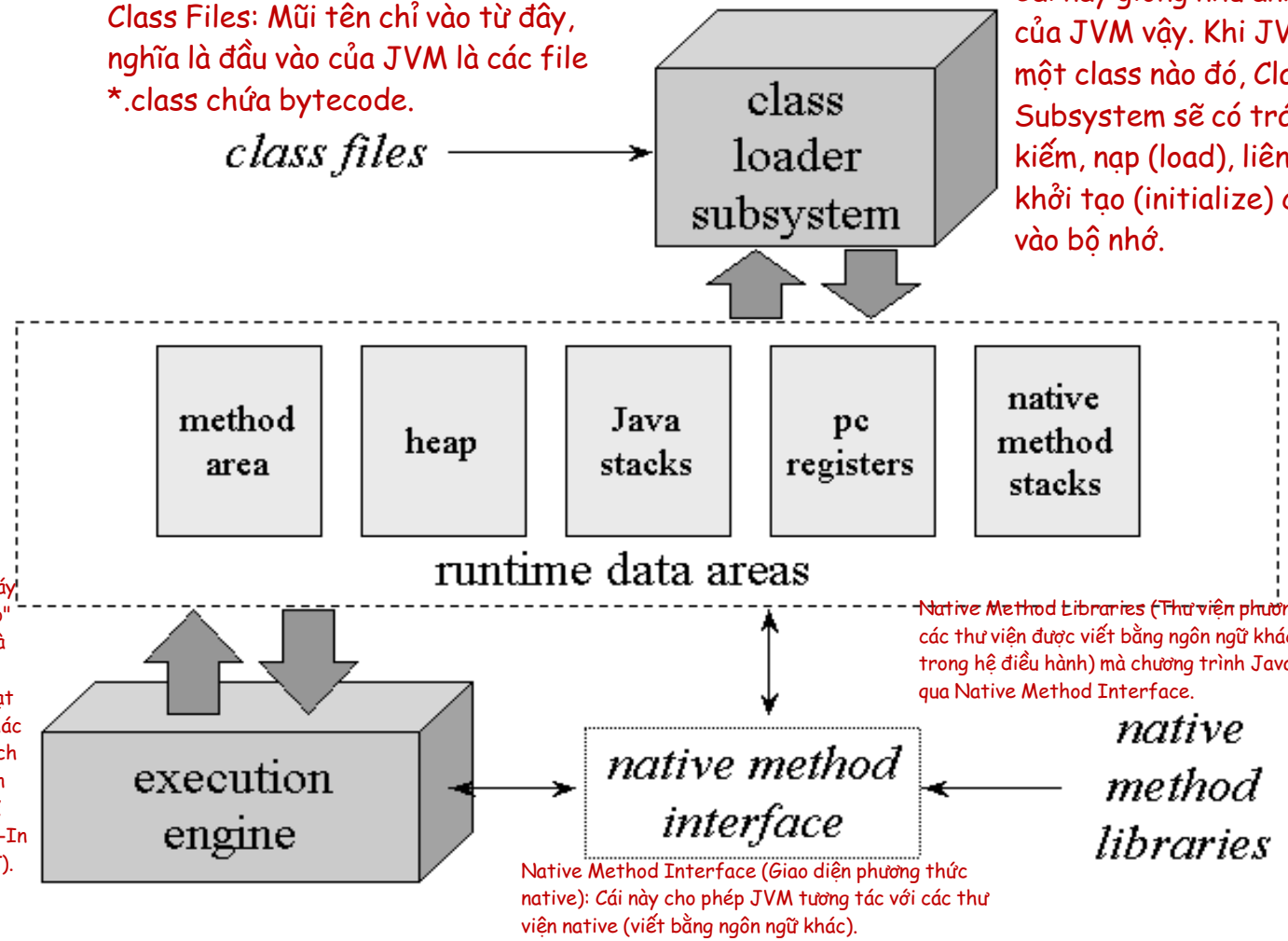
Cái này nói về cách JVM hoạt động bên trong. Khi người ta nói JVM là một stack-based machine, ý là nó sử dụng một cấu trúc dữ liệu gọi là stack (ngăn xếp) để thực hiện các phép tính và quản lý dữ liệu trong quá trình chạy chương trình.
Mày cứ hình dung cái stack giống như một chồng đĩa vậy đó: cái đĩa nào bỏ vào sau cùng thì sẽ được lấy ra đầu tiên (Last-In, First-Out - LIFO).
Giải thích từng ý trong slide:
A stack for each method (Một stack cho mỗi method): Khi một method (hàm) trong code của mày được gọi, JVM sẽ tạo ra một stack riêng cho method đó. Cái stack này được gọi là Java Virtual Machine Stack Frame (khung ngăn xếp của máy ảo Java). Nó sẽ chứa tất cả thông tin cần thiết cho việc thực hiện method đó.
The stack is used to store operands and results of an expression (Stack được dùng để lưu trữ toán hạng và kết quả của một biểu thức): Khi JVM thực hiện một phép tính (ví dụ như a = b + c), nó sẽ không dùng các thanh ghi (registers) trực tiếp như một số kiến trúc máy tính khác. Thay vào đó, nó sẽ:
Đẩy giá trị của b lên stack.
Đẩy giá trị của c lên stack.
Thực hiện phép cộng (+). Kết quả sẽ được đẩy trở lại stack.
Lấy kết quả từ stack và gán cho a.
It is also used to pass argument and receive returned value (Nó cũng được dùng để truyền tham số và nhận giá trị trả về): Khi mày gọi một method và truyền cho nó các tham số, những tham số này sẽ được đẩy lên stack của method đó. Khi method thực hiện xong và trả về một giá trị, giá trị đó cũng sẽ được đẩy lên stack để method gọi có thể lấy về.
Code generation for a stack-based machine is easier than that for a register-based one (Việc tạo mã cho một máy dựa trên stack dễ hơn so với máy dựa trên thanh ghi): Cái này là một ưu điểm quan trọng của kiến trúc stack-based. Trong kiến trúc dựa trên thanh ghi, compiler phải quyết định biến nào sẽ được lưu trữ trong thanh ghi nào, và việc này khá phức tạp, đặc biệt là khi số lượng thanh ghi có hạn. Với kiến trúc stack-based, compiler chỉ cần tạo ra các lệnh để đẩy (push) và lấy (pop) dữ liệu từ stack , và thực hiện các phép toán trên những giá trị nằm trên đỉnh stack. Điều này làm cho quá trình biên dịch bytecode trở nên đơn giản hơn và dễ dàng hơn cho các ngôn ngữ khác nhau muốn "nhắm mục tiêu" đến JVM.

Tóm lại: JVM sử dụng stack như là nơi làm việc chính để thực hiện các phép tính, lưu trữ dữ liệu tạm thời, và trao đổi thông tin giữa các method. Cách này giúp cho việc thiết kế JVM trở nên đơn giản hơn và cũng giúp cho việc biên dịch code từ nhiều ngôn ngữ khác nhau sang bytecode dễ dàng hơn.

Internal Architecture of JVM

Class Files: Mũi tên chỉ vào từ đây, nghĩa là đầu vào của JVM là các file *.class chứa bytecode.

Cái này giống như anh "gác cổng" của JVM vậy. Khi JVM cần chạy một class nào đó, Class Loader Subsystem sẽ có trách nhiệm tìm kiếm, nạp (load), liên kết (link), và khởi tạo (initialize) cái class đó vào bộ nhớ.



Runtime Data Areas (Vùng dữ liệu thời gian chạy): Đây là nơi JVM quản lý dữ liệu trong quá trình chạy chương trình.

Execution Engine (Bộ máy thực thi): Đây là "bộ não" thực sự của JVM, nơi mà các lệnh bytecode được thực hiện. Nó có thể hoạt động theo nhiều cách khác nhau, ví dụ như thông dịch từng lệnh một, hoặc biên dịch bytecode thành mã máy thật rồi chạy (Just-In-Time compilation - JIT).

Native Method Libraries (Thư viện phương thức native): Đây là các thư viện được viết bằng ngôn ngữ khác (ví dụ như các hàm trong hệ điều hành) mà chương trình Java có thể gọi đến thông qua Native Method Interface.

Native Method Interface (Giao diện phương thức native): Cái này cho phép JVM tương tác với các thư viện native (viết bằng ngôn ngữ khác).

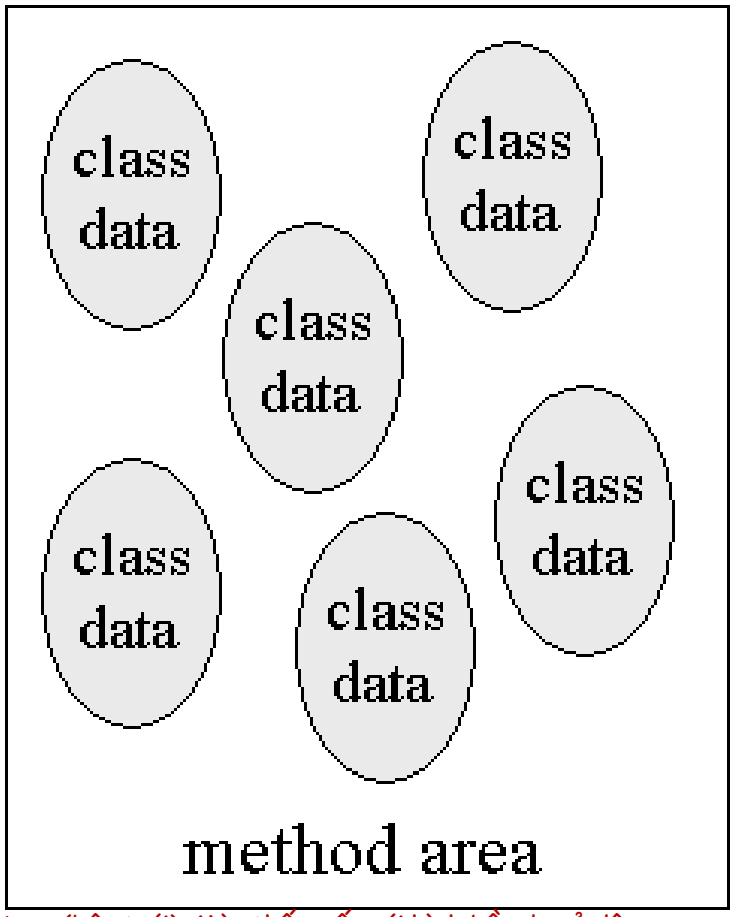
From [1]

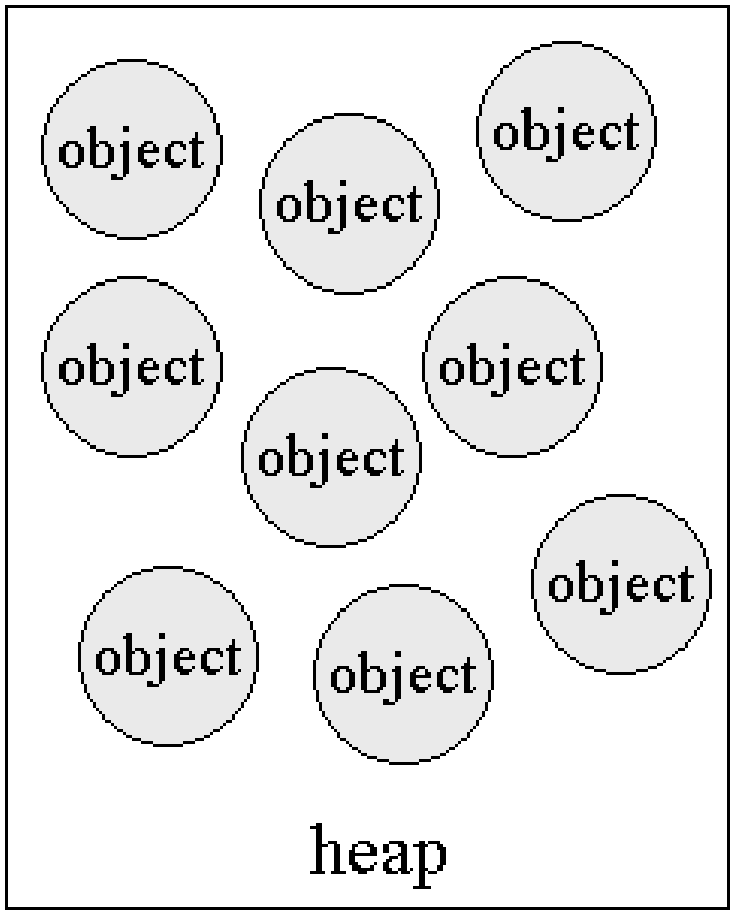
Runtime Data Areas (Vùng dữ liệu thời gian chạy): Đây là nơi JVM quản lý dữ liệu trong quá trình chạy chương trình. Nó bao gồm mấy vùng quan trọng sau:

Method Area (Vùng phương thức): Chứa thông tin về các class và interface đã được nạp, các phương thức, biến static, v.v. Nó giống như "bản thiết kế" của các đối tượng.
Heap: Đây là vùng bộ nhớ lớn nhất, được dùng để lưu trữ các đối tượng (instances) của các class mà mày tạo ra trong chương trình. Nó giống như "nhà kho" chứa đồ đạc của mày vậy.
Java Stacks (Ngăn xếp Java): Mỗi thread (luồng) trong chương trình Java sẽ có một Java Stack riêng. Cái stack này dùng để lưu trữ các stack frame, mà mỗi stack frame lại chứa thông tin về việc thực hiện một method (như biến cục bộ, toán hạng, địa chỉ trả về...). Cái này giống như "bảng nháp" của từng công việc đang được thực hiện.
PC Registers (Bộ đếm chương trình): Với mỗi thread, JVM sẽ theo dõi lệnh bytecode nào đang được thực hiện tiếp theo bằng một bộ đếm chương trình (Program Counter Register).
Native Method Stacks (Ngăn xếp phương thức native): Nếu chương trình của mày gọi các phương thức native (viết bằng ngôn ngữ khác như C/C++), thì những phương thức này sẽ được thực hiện trong Native Method Stacks.

Method Area: Lưu trữ thông tin về các class.
Heap: Lưu trữ các đối tượng được tạo ra từ các class đó.

Method Area and Heap





Method Area (bên trái): Mày thấy mấy cái hình bầu dục ở đây đều ghi là "class data". Đúng như tên gọi, vùng này dùng để lưu trữ thông tin liên quan đến class. Cụ thể là:
Thông tin về bản thân class (tên, superclass, interfaces,...).
Các biến static (static fields).
Thông tin về các method (tên, tham số, bytecode,...).
Constant pool (bảng các hằng số).

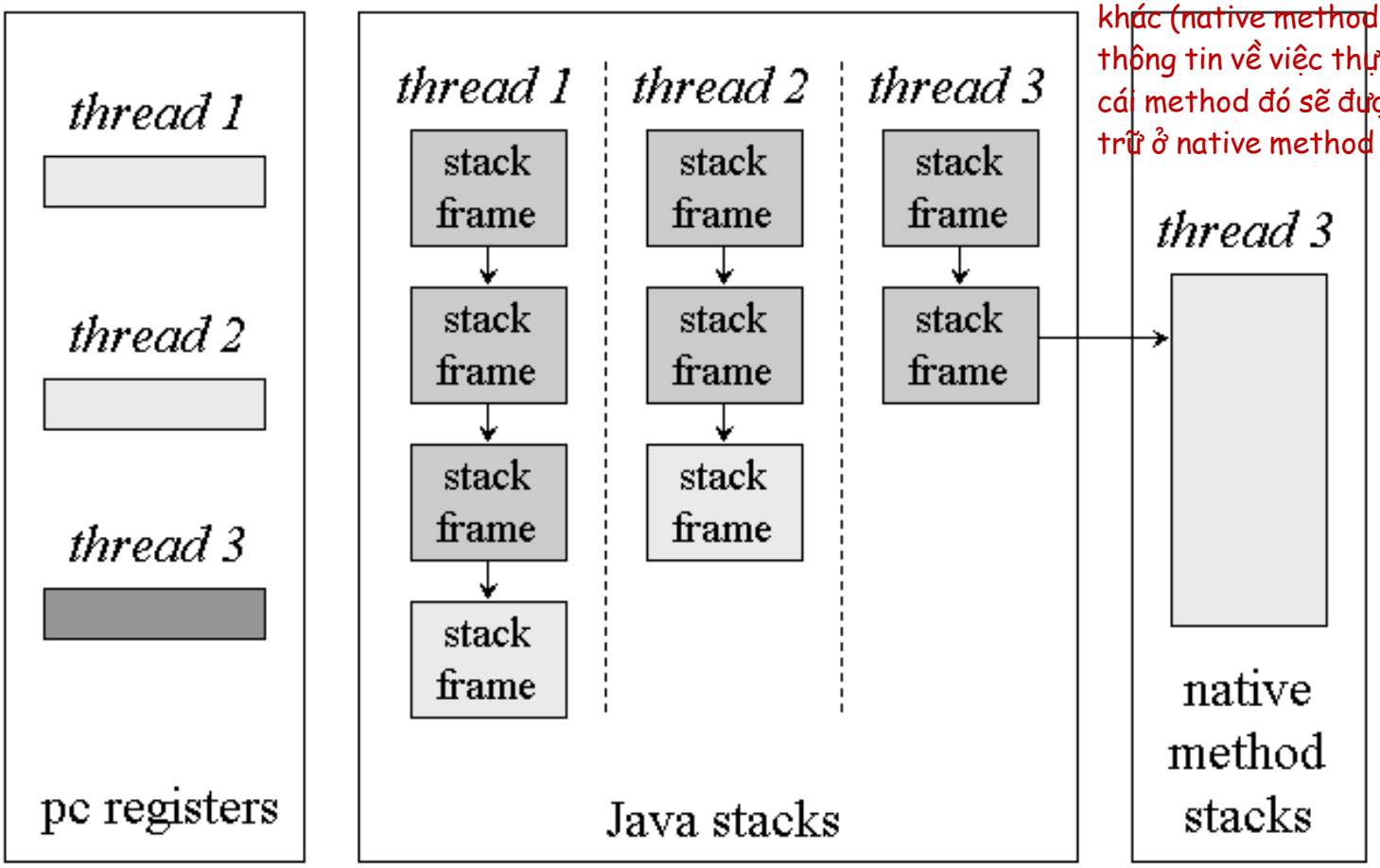
Heap (bên phải): Ở đây, mấy cái hình bầu dục lại ghi là "(object)". Vùng Heap là nơi mà tất cả các đối tượng (instances của các class) được tạo ra trong quá trình chạy chương trình sẽ được lưu trữ.
Khi mày dùng từ khóa new để tạo một đối tượng (ví dụ Student s = new Student();), thì cái đối tượng Student đó sẽ được cấp phát bộ nhớ ở vùng Heap.
Heap giống như khu dân cư, nơi mà các ngôi nhà (đối tượng) được xây dựng dựa trên bản thiết kế (thông tin class trong Method Area).

mỗi thread (luồng) trong một chương trình Java sẽ có một Java Stack riêng.

Bên trái: Nó cho thấy có thể có nhiều thread đang chạy đồng thời (thread 1, thread 2, thread 3). Mỗi thread sẽ có một vùng nhớ riêng để lưu trữ thông tin của nó.

Java Stacks

Bên phải: Nó lại nhắc đến native method stacks. Nếu một thread gọi một method được viết bằng ngôn ngữ khác (native method), thì thông tin về việc thực hiện cái method đó sẽ được lưu trữ ở native method stacks.



From [1]

Chính giữa: Đây là phần quan trọng nhất, nó minh họa chi tiết hơn về Java Stacks.
Mỗi thread (thread 1, thread 2, thread 3) có một stack riêng.
Trong mỗi stack lại chứa các stack frame. Mỗi stack frame tương ứng với một method (hàm) đang được thực hiện.
Khi một method được gọi, một stack frame mới sẽ được đẩy (push) lên stack. Khi method đó hoàn thành, stack frame của nó sẽ được lấy ra (pop) khỏi stack.
Mày thấy mũi tên đi xuống không? Nó thể hiện thứ tự gọi method. Ví dụ, ở thread 1, method đầu tiên gọi method thứ hai, rồi gọi tiếp method thứ ba. Cái method thứ ba sẽ được thực hiện xong trước, rồi đến method thứ hai, và cuối cùng là method đầu tiên. Đúng theo nguyên tắc LIFO (Last-In, First-Out) của stack.

Tóm lại:

Mỗi thread trong Java có một stack riêng.
Stack này chứa các stack frame, mỗi frame đại diện cho một method đang được thực hiện.
Stack hoạt động theo cơ chế LIFO.
Có cả stack riêng cho các native method.

Outline

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

Data Types

cột "Description" là cái ký hiệu mà JVM dùng để biểu diễn các kiểu dữ liệu này trong bytecode. Ví dụ, Z là boolean, B là byte, I là int, v.v.

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

returnAddress: Cái này là địa chỉ của một opcode (một lệnh trong bytecode) bên trong cùng một method. Nó được dùng bởi một số lệnh đặc biệt như jsr (jump to subroutine) và ret (return from subroutine), nhưng mà mấy lệnh này ít khi được dùng trong code Java hiện đại do có cấu trúc điều khiển tốt hơn rồi. Cái này mày cứ hiểu là nó liên quan đến việc nhảy và quay lại trong quá trình thực thi method thôi.

class reference: Cái này là một tham chiếu đến một class. Trong bytecode, nó thường được biểu diễn như L<tên-class>;. Ví dụ, tham chiếu đến class java.lang.String sẽ là Ljava/lang/String;.

interface reference: Tương tự như class reference, nhưng mà dành cho interface. Nó cũng được biểu diễn tương tự: L<tên-interface>;.

array reference: Cái này là tham chiếu đến một mảng. Cách biểu diễn của nó hơi phức tạp hơn một chút. Nó bắt đầu bằng một hoặc nhiều dấu [tùy thuộc vào số chiều của mảng, sau đó là kiểu dữ liệu của các phần tử trong mảng. Ví dụ:

Mảng các số nguyên (int): [I
Mảng hai chiều các số thực (float): [[F
Mảng các đối tượng String (String): [Ljava/lang/String;
void: Cái này thì chắc mày biết rồi, nó biểu thị là không có giá trị trả về (thường dùng cho các method). Trong bytecode, nó được ký hiệu là 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 là một phần của stack frame (khung ngăn xếp)

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

Lưu trữ các toán hạng: Khi mày viết một biểu thức như a = b * 2 + 40;, các giá trị của b, 2, và 40 sẽ được đẩy lên Operand Stack.
Nhận kết quả của các phép toán: Sau khi thực hiện các phép tính như * (nhân) hoặc + (cộng), kết quả của phép tính đó sẽ được đẩy lên Operand Stack.
Truyền tham số và nhận kết quả trả về của method: Khi mày gọi một method, các tham số mày truyền cho method đó sẽ được đẩy lên Operand Stack. Khi method đó thực hiện xong và trả về một giá trị, giá trị đó cũng sẽ được đẩy lên Operand Stack để mày lấy về.

```
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
```

Operand Stack bây giờ rỗng.

Outline

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

Local Variable Array

(Một mảng biến cục bộ mới được tạo ra mỗi khi một method được gọi): Mỗi khi một method bắt đầu thực thi, JVM sẽ tạo ra một mảng mới để chứa các biến cục bộ của riêng method đó. Khi method kết thúc, cái mảng này cũng biến mất. Nó giống như mỗi method có một "bảng đen" riêng để ghi chú các biến của nó.

1. A new local variable array is created each time a method is called
2. Local variables addressed by indexing, starting from 0

(Các biến cục bộ được truy cập bằng cách sử dụng chỉ số, bắt đầu từ 0): Các biến cục bộ trong mảng này được sắp xếp theo thứ tự và mỗi biến sẽ có một chỉ số (index) để JVM có thể truy cập đến nó. Chỉ số này bắt đầu từ 0 cho biến đầu tiên.

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

(Các method instance - không có từ khóa static):

slot 0 given to this: Trong các method instance, cái biến this (tham chiếu đến đối tượng hiện tại) luôn luôn được lưu trữ ở vị trí đầu tiên của mảng, tức là ở chỉ số 0.
Parameters (if any) given consecutive indices, starting from 1: Các tham số mà mày truyền vào method khi gọi nó sẽ được lưu trữ ở các vị trí tiếp theo, bắt đầu từ chỉ số 1.
The indices allocated to the other variables in any order: Các biến cục bộ khác mà mày khai báo bên trong method sẽ được cấp phát các chỉ số còn lại theo một thứ tự nào đó (thường là theo thứ tự chúng xuất hiện trong code, nhưng không nhất thiết).

4. Class methods:
   - Parameters (if any) given consecutive indices, starting from 0
   - The indices allocated to the other variables in any order

Class methods (Các method class - có từ khóa static):

Parameters (if any) given consecutive indices, starting from 0: Trong các method static, không có this. Vì vậy, các tham số truyền vào sẽ được lưu trữ ở các vị trí bắt đầu từ chỉ số 0.
The indices allocated to the other variables in any order: Tương tự như instance methods, các biến cục bộ khác sẽ được cấp phát các chỉ số còn lại.

5. One slot can hold a value of boolean, byte, char, short, int, float,reference and returnAdrress

One slot can hold a value of boolean, byte, char, short, int, float, reference and returnAddress (Một "ô" (slot) có thể chứa một giá trị kiểu boolean, byte, char, short, int, float, tham chiếu và returnAddress): Hầu hết các kiểu dữ liệu "nhỏ" (trừ long và double) chỉ cần một "ô" trong mảng để lưu trữ giá trị của chúng. "Reference" ở đây là tham chiếu đến các đối tượng (nằm ở Heap). "returnAddress" là cái địa chỉ trả về mà mình đã nói trước đó.

6. One pair of slots can hold a value of long and double

One pair of slots can hold a value of long and double (Một cặp "ô" có thể chứa một giá trị kiểu long và double): Vì long và double là các kiểu dữ liệu 64-bit (gấp đôi kích thước của int và float), chúng cần hai "ô" liên tiếp trong mảng để lưu trữ giá trị của mình.

Instance Methods (Phương thức instance):

Không có từ khóa static: Khi mày định nghĩa một method mà không có từ khóa static ở phía trước, nó là instance method.
Thuộc về đối tượng: Instance methods gắn liền với một đối tượng cụ thể của class. Muốn gọi nó, mày phải có một đối tượng trước.
Có quyền truy cập vào this: Bên trong instance method, mày có thể sử dụng từ khóa this để tham chiếu đến đối tượng hiện tại mà method đó đang được gọi. Điều này cho phép mày truy cập và thay đổi các thuộc tính (biến instance) của đối tượng đó.
Ví dụ: Mày có một class Dog. Method bark() (tiếng chó sủa) là một instance method. Mỗi con chó (mỗi đối tượng Dog) sẽ có thể sủa theo cách riêng của nó.

Đặc điểm	Instance Methods	Class Methods (Static Methods)
Từ khóa	Không có `static`	Có `static`
Thuộc về	Đối tượng (cần tạo đối tượng để gọi)	Class (gọi trực tiếp qua tên class)
`this`	Có thể sử dụng `this`	Không thể sử dụng `this` trực tiếp
Mục đích	Thao tác trên trạng thái của một đối tượng cụ thể	Thao tác chung, không phụ thuộc vào đối tượng cụ thể

Class Methods (Phương thức class hay phương thức static):

Có từ khóa static: Khi mày định nghĩa một method có từ khóa static ở phía trước, nó là class method.
Thuộc về class: Class methods thuộc về chính class đó, chứ không phải một đối tượng cụ thể nào. Mày có thể gọi nó trực tiếp thông qua tên class mà không cần tạo ra đối tượng trước.
Không có quyền truy cập trực tiếp vào this: Bên trong class method, mày không thể sử dụng từ khóa this vì nó không được gọi trên một đối tượng cụ thể nào.
Class methods thường được dùng cho các thao tác mang tính chất chung, không liên quan đến trạng thái của một đối tượng cụ thể.
Ví dụ: Trong class Math, các method như sqrt() (tính căn bậc hai) hay random() (tạo số ngẫu nhiên) là các class methods. Mày gọi chúng bằng cách viết Math.sqrt(16) chứ không cần tạo ra một đối tượng Math nào cả.

Example 1

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

```
.line 7
                    1 store vào 0 (a)
    iconst_1
    istore_0        // a
.line 8
    iconst_2        2 store vào 1 (b)
    istore_1        // b
.line 9
    iload_0
    iload_1
    iadd
    iconst_3
    imul
    istore_2        // c
```

```
public static void foo() {
   int a,b,c; // a ở index 0, b ở index 1, c ở index 2
   a = 1;     // istore_0 (lưu giá trị 1 vào biến ở index 0 - là a)
   b = 2;     // istore_1 (lưu giá trị 2 vào biến ở index 1 - là b)
   c = (a + b) * 3; // ... istore_2 (lưu kết quả vào biến ở index 2 - là c)
}
```

Trong method static này, không có đối tượng this. Vì vậy, các biến cục bộ a, b, c được cấp phát index bắt đầu từ 0.

Trong method instance này, vị trí index 0 trong Local Variable Array sẽ được dành cho tham chiếu this (tham chiếu đến đối tượng mà method này đang được gọi). Do đó, các biến cục bộ a, b, c sẽ được cấp phát index bắt đầu từ 1.

Example 2

Do thằng này không có static -> Nó là class method, khi đó nó có .this, nên index của a sẽ nằm ở 1 thay vì 0

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

Cái dòng .var 0 is this LVD2 này chính là để khai báo rằng biến ở index 0 là this (với kiểu dữ liệu là LVD2; - có thể là tên class).

```
.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
```

Không có static -> Có .this -> Index của biến bắt đầu từ 1

Example 3

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

```
public void foo() {
  int    a = 1;   // a ở index 1
  long b = 2;    // b ở index 2 và 3 (chiếm 2 slot)
  int c = 3;     // c ở index 4
  long d = (a + b) * c; } // d ở index 5 và 6 (chiếm 2 slot)
```

```
.line 6
    iconst_1
    istore_1            // a
.line 7
    ldc2_w 2            Đẩy giá trị 2 (kiểu long) lên Operand Stack. Chú ý là hằng số long 2 được load bằng ldc2_w.
    lstore_2            // 2,3 for b
.line 8
    iconst_3
    istore 4            // c
.line 9
    iload_1             Lấy giá trị của a (int) từ index 1 và đẩy lên stack.
    i2l                 // conversion   Chuyển đổi giá trị int trên stack thành long và đẩy kết quả (long) trở lại stack. Cái này là vì mình sắp cộng nó với b (kiểu long).
    lload_2             Lấy giá trị của b (long) từ index 2 và 3 và đẩy lên stack.
    ladd                ladd: Lấy hai giá trị long trên stack ra, cộng chúng lại, và đẩy kết quả (long) trở lại stack.
    iload 4             iload 4: Lấy giá trị của c (int) từ index 4 và đẩy lên stack.
    i2l                 // conversion   Chuyển đổi giá trị int trên stack thành long và đẩy kết quả (long) trở lại stack.
    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 (Lệnh số học): Nó bao gồm các lệnh thực hiện các phép toán số học cơ bản như cộng (iadd, ladd, fadd, dadd), trừ (isub, lsub, fsub, dsub), nhân (imul, lmul, fmul, dmul), chia (idiv, ldiv, fdiv, ddiv), chia lấy dư (irem, lrem, frem, drem), và các phép toán bitwise (AND, OR, XOR, SHIFT).

Load and store instructions (Lệnh nạp và lưu trữ): Nhóm lệnh này dùng để di chuyển dữ liệu giữa Local Variable Array và Operand Stack.

Load: Lấy giá trị từ một vị trí trong Local Variable Array và đẩy nó lên Operand Stack (ví dụ: iload, lload, fload, dload, aload - load reference).
Store: Lấy giá trị từ đỉnh Operand Stack và lưu nó vào một vị trí trong Local Variable Array (ví dụ: istore, lstore, fstore, dstore, astore).
Ngoài ra còn có các lệnh để load các hằng số (ví dụ: iconst, lconst, fconst, dconst, bipush, sipush, ldc).

Control transfer instructions (Lệnh chuyển điều khiển): Nhóm lệnh này dùng để điều khiển luồng thực thi của chương trình, ví dụ như các lệnh rẽ nhánh (if-else), vòng lặp (for, while), và gọi method. Ví dụ: goto, ifeq, ifne, iflt, ifgt, jsr, ret, tableswitch, lookupswitch.

Type conversion instructions (Lệnh chuyển đổi kiểu dữ liệu): Nhóm lệnh này dùng để chuyển đổi giữa các kiểu dữ liệu nguyên thủy khác nhau (ví dụ: i2l - int to long, f2d - float to double, d2i - double to int, v.v.).

Operand stack management instructions (Lệnh quản lý Operand Stack): Nhóm lệnh này dùng để trực tiếp thao tác với Operand Stack, ví dụ như đẩy một giá trị trùng lặp lên stack (dup), hoán đổi hai giá trị trên đỉnh stack (swap), hoặc loại bỏ một giá trị từ đỉnh stack (pop).

Object creation and manipulation (Lệnh tạo và thao tác với đối tượng): Nhóm lệnh này dùng để tạo mới đối tượng (new), truy cập các trường (fields) của đối tượng (getfield, putfield), và truy cập các trường static (getstatic, putstatic).

Method invocation instructions (Lệnh gọi method): Nhóm lệnh này dùng để gọi các method khác nhau. Có nhiều loại lệnh gọi method tùy thuộc vào loại method (instance, static, private, interface, constructor). Ví dụ: invokevirtual, invokestatic, invokespecial, invokeinterface.

Throwing instructions (Lệnh ném exception - not used): Trong bytecode JVM vẫn có lệnh athrow để ném exception, nhưng slide này nói là "not used" có thể là trong ngữ cảnh của Jasmin hoặc là một nhận xét riêng của người soạn slide. Thông thường thì lệnh này vẫn được sử dụng khi có exception xảy ra.

Implementing finally (not used): Tương tự như trên, bytecode vẫn có cơ chế để xử lý khối finally (ví dụ dùng jsr và ret), nhưng có lẽ slide này muốn nói đến một cách tiếp cận cụ thể nào đó trong Jasmin mà không được sử dụng nữa.

Synchronization (not used): Bytecode có các lệnh như monitorenter và monitorexit để thực hiện đồng bộ hóa (synchronization), nhưng slide này lại ghi là "not used". Có thể là trong phạm vi của bài giảng này hoặc do một lý do cụ thể nào đó mà nó không được đề cập chi tiết.

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*.

Negate (Đổi dấu): Đổi từ = -> - và ngược lại

ishl: Dịch trái số nguyên (int) có dấu.
ishr: Dịch phải số nguyên (int) có dấu (bảo toàn dấu).
iushr: Dịch phải số nguyên (int) không dấu (zero-fill right shift).

Tăng giá trị của một biến cục bộ kiểu nguyên (int) lên một lượng nhất định (thường là 1). Đây là một lệnh đặc biệt để tối ưu hóa việc tăng biến cục bộ

dcmpg: So sánh hai số thực dấu chấm động độ chính xác kép (double). Trả về 1 nếu value1 > value2, -1 nếu value1 < value2, và 0 nếu value1 == value2. Nếu một trong hai là NaN (Not-a-Number), nó trả về 1.

From (§3.11.3,[3])

Cái này dùng để "bốc" dữ liệu từ Local Variable Array lên Operand Stack (Load) và "cất" dữ liệu từ Operand Stack xuống Local Variable Array (Store).

Load and Store

iload: Lệnh này dùng để nạp một giá trị kiểu int (hoặc boolean, byte, char, short vì chúng thường được xử lý như int trên stack) từ Local Variable Array lên Operand Stack. Sau lệnh này, giá trị của biến sẽ nằm trên đỉnh stack.

- 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

Ví dụ, iload_0 tương đương với iload 0, nhưng nó ngắn gọn và thường nhanh hơn một chút.

Các lệnh có hậu tố _<n> (ví dụ iload_0, astore_3) là các phiên bản ngắn gọn và thường hiệu quả hơn để truy cập các biến cục bộ ở những index đầu tiên (0, 1, 2, 3).

istore: Lệnh này lấy một giá trị kiểu int (hoặc boolean, byte, char, short) từ đỉnh Operand Stack và lưu nó vào một biến cục bộ trong Local Variable Array.

- 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])

Lệnh Load Variable (iload, lload, aload, etc.): Lấy giá trị của một biến đã được lưu trữ trước đó trong Local Variable Array và đẩy nó lên Operand Stack. Mày cứ hình dung là mày đang "lấy" một cái gì đó từ trong "tủ" (Local Variable Array) ra để dùng.

Lệnh Load Constant (bipush, sipush, ldc, iconst_0, etc.): Đẩy trực tiếp một giá trị hằng số (giá trị cố định đã được xác định trong code) lên Operand Stack. Mày cứ hình dung là mày đang "tạo ra" một giá trị mới và đưa nó vào để dùng.

Nguồn gốc dữ liệu: Load variable lấy dữ liệu từ Local Variable Array (nơi chứa các biến cục bộ). Load constant thì lấy dữ liệu trực tiếp từ hằng số được định nghĩa trong code (hoặc constant pool).
Tính linh hoạt: Load variable linh hoạt hơn vì giá trị của biến có thể thay đổi trong quá trình chạy. Load constant thì giá trị luôn cố định.

iload_1: Lấy giá trị của biến ở index 1 trong Local Variable Array.
iconst_5: Đẩy trực tiếp giá trị hằng số 5 lên stack.

Load and Store (cont'd)

Nạp hằng số lên Operand Stack, thay vì phải load từ Local Variable Array, đôi khi mình cần dùng trực tiếp các giá trị hằng số.

- Load a constant onto the operand stack:

bipush: Dùng để đẩy một hằng số nguyên (integer) kiểu byte (tức là từ -128 đến 127) lên stack. Cái này là viết tắt của "byte push".

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])

bipush: Dùng để đẩy một hằng số nguyên (integer) kiểu byte (tức là từ -128 đến 127) lên stack. Cái này là viết tắt của "byte push".

sipush: Dùng để đẩy một hằng số nguyên (integer) kiểu short (tức là từ -32768 đến 32767) lên stack. Cái này là viết tắt của "short push".

ldc: Dùng để đẩy một hằng số là số nguyên (int), số thực dấu chấm động (float), hoặc một chuỗi (String) đã được định nghĩa trong constant pool của class.

ldc_w: Tương tự như ldc, nhưng được dùng cho các hằng số kiểu long hoặc double mà index trong constant pool lớn hơn phạm vi của ldc. Nó cũng có thể dùng cho int, float, và String. Chữ 'w' ở đây có nghĩa là "wide" (rộng).

ldc2_w: Dùng đặc biệt cho các hằng số kiểu long hoặc double. Vì long và double chiếm 2 slot trong constant pool, nên cần lệnh này.

aconst_null: Dùng để đẩy giá trị null (tham chiếu rỗng) lên stack. Chữ 'a' ở đây thường dùng cho "reference".

iconst_m1: Dùng để đẩy hằng số nguyên -1 lên stack. Chữ 'i' là viết tắt của "integer", và 'm1' là -1.

iconst_<i> (với i từ 0 đến 5): Đây là các lệnh đặc biệt để đẩy các hằng số nguyên từ 0 đến 5 lên stack một cách nhanh chóng. Ví dụ: iconst_0 đẩy 0, iconst_1 đẩy 1, ..., iconst_5 đẩy 5.

lconst_<l> (với l là 0 hoặc 1): Tương tự như iconst_<i>, đây là các lệnh nhanh để đẩy hằng số nguyên lớn (long) 0 (lconst_0) hoặc 1 (lconst_1) lên stack. Chữ 'l' là viết tắt của "long".

fconst_<f> (với f là 0.0, 1.0, hoặc 2.0): Lệnh nhanh để đẩy hằng số thực dấu chấm động (float) 0.0 (fconst_0), 1.0 (fconst_1), hoặc 2.0 (fconst_2) lên stack. Chữ 'f' là viết tắt của "float".

dconst_<d> (với d là 0.0 hoặc 1.0): Lệnh nhanh để đẩy hằng số thực dấu chấm động độ chính xác kép (double) 0.0 (dconst_0) hoặc 1.0 (dconst_1) lên stack. Chữ 'd' là viết tắt của "double".

Example 4

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

Lấy giá trị 1 từ stack và lưu vào biến cục bộ ở index 1 (biến a). Chú ý là đây là instance method (không thấy static), nên index 0 là this.

```
.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
```

bipush 100: Đẩy hằng số nguyên 100 (kiểu byte, nằm trong khoảng -128 đến 127) lên Operand Stack.

sipush 1000: Đẩy hằng số nguyên 1000 (kiểu short, nằm trong khoảng -32768 đến 32767) lên Operand Stack.

ldc 40000: Đẩy hằng số nguyên 40000 lên Operand Stack. Giá trị này lớn hơn phạm vi của sipush nên dùng ldc.

dòng code Jasmin ở line 6:

.line 6 iconst_1 istore_1

Hoàn toàn có thể thay bằng:

.line 6 bipush 1 istore_1

JVM sẽ hiểu cả hai lệnh này và thực hiện cùng một việc là đẩy giá trị 1 lên Operand Stack rồi lưu vào biến a (ở index 1).

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 a
        iconst_0        // push 0
        bipush 100      // push 100
        iastore         // a[0] = 100
```

iastore: Lấy ba giá trị trên đỉnh stack ra (giá trị, index, tham chiếu mảng), và gán giá trị 100 vào phần tử có index 0 của mảng a.

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

Dùng khi rẽ nhánh, nhảy tới nhảy lui...

Control Transfer Instructions

Unconditional branch (nhảy không điều kiện):
goto: Nhảy đến một label (nhãn) được chỉ định trong code. Chương trình sẽ tiếp tục thực thi từ cái label đó.
goto_w: Tương tự goto, nhưng dùng cho các label ở xa hơn (có offset lớn hơn). Chữ 'w' là viết tắt của "wide".
jsr: Nhảy đến một subroutine (chương trình con) và lưu địa chỉ trả về trên Operand Stack. Thường dùng cho việc triển khai khối finally.
jsr_w: Tương tự jsr nhưng cho địa chỉ xa hơn.
ret: Trả về từ một subroutine. Nó lấy địa chỉ trả về từ Local Variable Array (được lưu bởi jsr hoặc jsr_w) và nhảy đến đó.

- 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])

2. Conditional branch (Nhảy có điều kiện):
Nhóm này sẽ kiểm tra một điều kiện nào đó, và nếu điều kiện đúng thì sẽ nhảy đến một label, còn nếu sai thì sẽ tiếp tục thực hiện lệnh kế tiếp.

So sánh số nguyên với 0:
ifeq: Nhảy nếu giá trị trên đỉnh stack bằng 0.
iflt: Nhảy nếu giá trị trên đỉnh stack nhỏ hơn 0.
ifle: Nhảy nếu giá trị trên đỉnh stack nhỏ hơn hoặc bằng 0.
ifne: Nhảy nếu giá trị trên đỉnh stack khác 0.
ifgt: Nhảy nếu giá trị trên đỉnh stack lớn hơn 0.
ifge: Nhảy nếu giá trị trên đỉnh stack lớn hơn hoặc bằng 0.

So sánh tham chiếu với null:
ifnull: Nhảy nếu tham chiếu trên đỉnh stack là null.
ifnonnull: Nhảy nếu tham chiếu trên đỉnh stack không phải là null.

So sánh hai số nguyên:
if_icmpeq: Nhảy nếu hai giá trị integer trên đỉnh stack bằng nhau. ('icmp' là integer compare)
if_icmpne: Nhảy nếu hai giá trị integer trên đỉnh stack khác nhau.
if_icmplt: Nhảy nếu giá trị integer thứ nhất trên đỉnh stack nhỏ hơn giá trị integer thứ hai.
if_icmpgt: Nhảy nếu giá trị integer thứ nhất trên đỉnh stack lớn hơn giá trị integer thứ hai.
if_icmple: Nhảy nếu giá trị integer thứ nhất trên đỉnh stack nhỏ hơn hoặc bằng giá trị integer thứ hai.
if_icmpge: Nhảy nếu giá trị integer thứ nhất trên đỉnh stack lớn hơn hoặc bằng giá trị integer thứ hai.

So sánh hai tham chiếu:
if_acmpeq: Nhảy nếu hai tham chiếu trên đỉnh stack tham chiếu đến cùng một đối tượng. ('acmp' là reference compare)
if_acmpne: Nhảy nếu hai tham chiếu trên đỉnh stack không tham chiếu đến cùng một đối tượng.

Compound conditional branch (Nhảy điều kiện phức tạp):

tableswitch: Dùng cho các trường hợp switch-case mà các giá trị case là các số nguyên liên tiếp nhau. Nó sẽ nhảy đến một label cụ thể dựa trên giá trị của biến trên stack.
lookupswitch: Dùng cho các trường hợp switch-case mà các giá trị case không liên tiếp nhau. Nó sẽ so sánh giá trị trên stack với một bảng các giá trị và nhảy đến label tương ứng.

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   nếu a <= b -> nhảy tới label 0
.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:
```

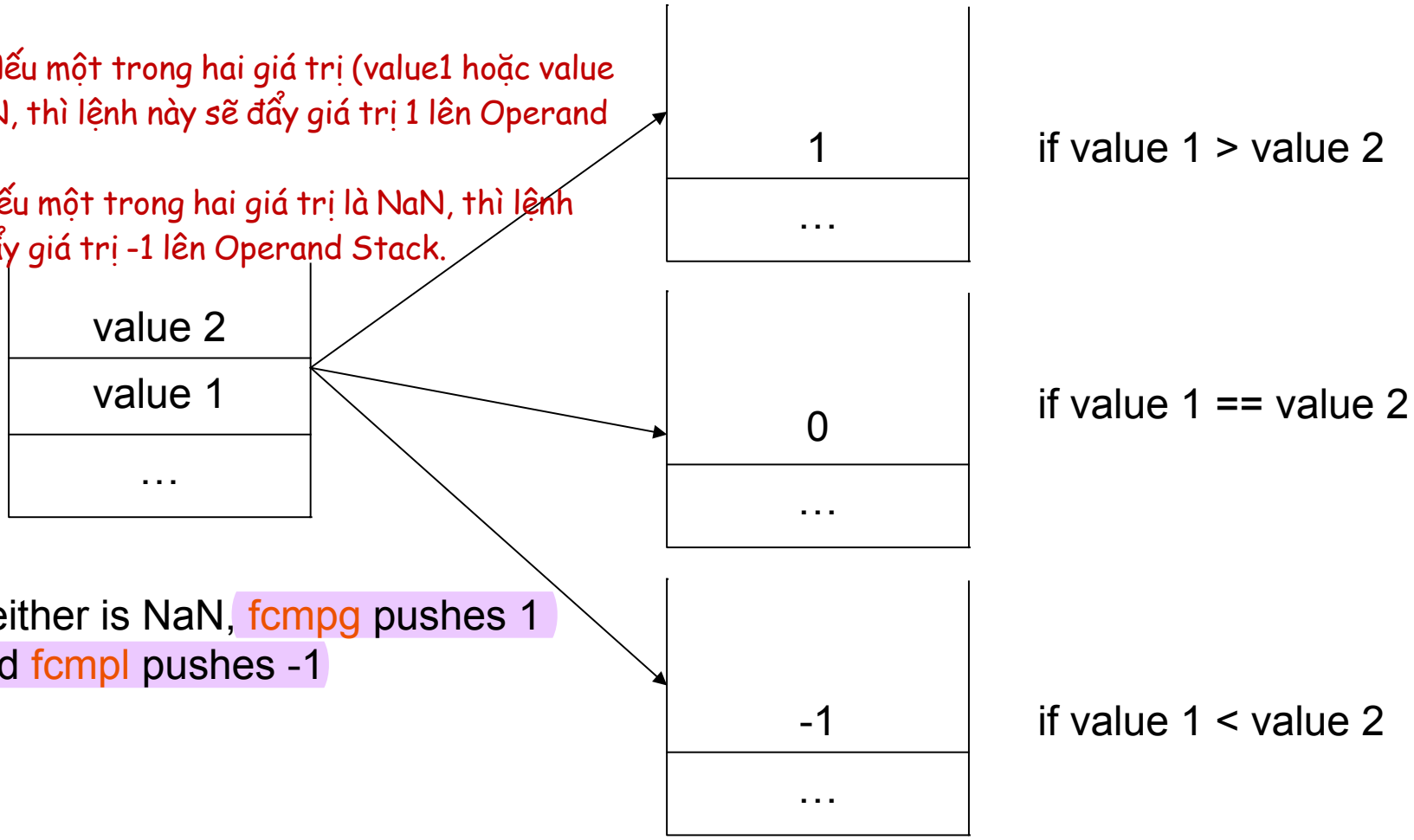
lệnh fcmpl so sánh float và push kết quả lên stack: -1 nếu a < b, 0 nếu a == b, 1 nếu a > b, -1 nếu có NaN

ifle Label0 nhảy nếu kết quả trên stack nhỏ hơn hoặc bằng 0

value2 được đẩy lên trước value1

fcmpg and fcmpl

Điểm khác biệt quan trọng giữa fcmpg và fcmpl nằm ở chỗ chúng xử lý giá trị NaN (Not-a-Number) như thế nào:
fcmpg: Nếu một trong hai giá trị (value1 hoặc value 2) là NaN, thì lệnh này sẽ đẩy giá trị 1 lên Operand Stack.
fcmpl: Nếu một trong hai giá trị là NaN, thì lệnh này sẽ đẩy giá trị -1 lên Operand Stack.

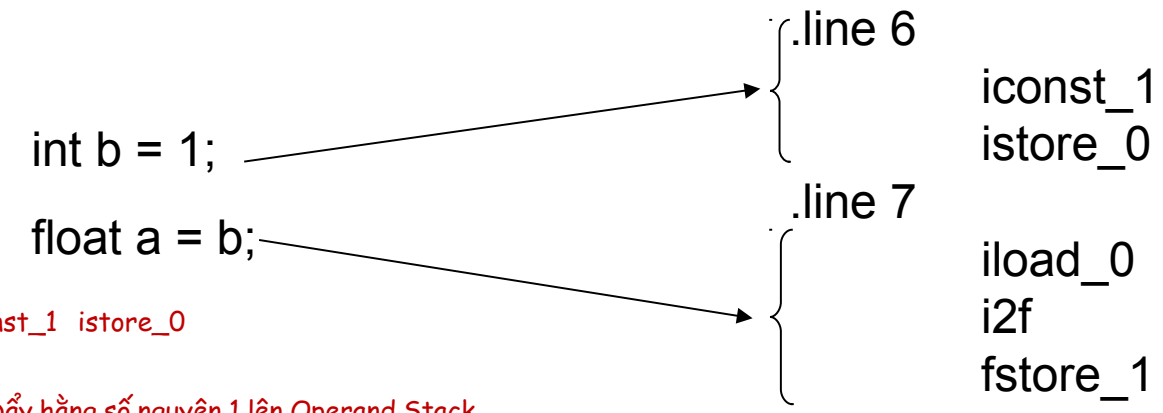


If either is NaN, fcmpg pushes 1 and fcmpl pushes -1

Type Conversion Instructions

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

i2l: Chuyển đổi từ kiểu int sang kiểu long.
i2f: Chuyển đổi từ kiểu int sang kiểu float.
i2d: Chuyển đổi từ kiểu int sang kiểu double.
l2f: Chuyển đổi từ kiểu long sang kiểu float.
l2d: Chuyển đổi từ kiểu long sang kiểu double.
f2d: Chuyển đổi từ kiểu float sang kiểu double.



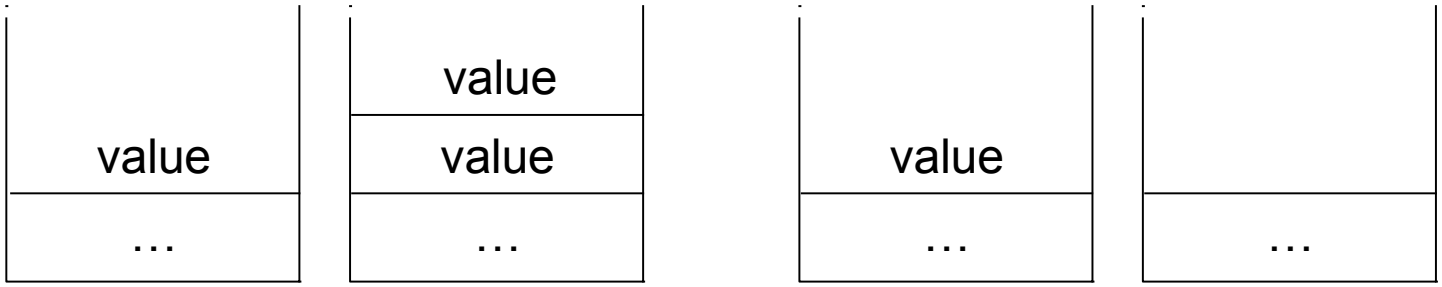
.line 6 iconst_1 istore_0

iconst_1: Đẩy hằng số nguyên 1 lên Operand Stack.
istore_0: Lấy giá trị 1 từ stack và lưu vào biến cục bộ ở index 0 (biến b).
.line 7 iload_0 i2f fstore_1

iload_0: Lấy giá trị của biến b (ở index 0) và đẩy lên Operand Stack (giá trị là 1).
i2f: Lấy giá trị int trên đỉnh stack (là 1) và chuyển đổi nó thành giá trị float tương ứng (là 1.0f). Kết quả float này được đẩy trở lại stack.
fstore_1: Lấy giá trị float từ stack (là 1.0f) và lưu vào biến cục bộ ở index 1 (biến a).

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,…

dup: Lệnh này dùng để nhân đôi giá trị nằm trên đỉnh Operand Stack. Mày nhìn cái hình minh họa sẽ thấy, nếu trên đỉnh stack đang có một value, sau khi dup thì trên đỉnh stack sẽ có thêm một bản sao nữa của value.

Khi nào dùng? Slide có ghi là thường dùng khi dịch các phép gán liên tiếp kiểu a = b = ... Ví dụ, nếu mày muốn gán cùng một giá trị cho nhiều biến, mày có thể dup giá trị đó trên stack rồi store nó vào từng biến.
pop: Lệnh này dùng để loại bỏ giá trị nằm trên đỉnh Operand Stack. Sau khi pop, cái giá trị trên cùng sẽ biến mất khỏi stack.

Khi nào dùng? Slide có ví dụ là khi dịch các câu lệnh kiểu 1;. Trong trường hợp này, có thể một phép toán nào đó đã được thực hiện và kết quả được đẩy lên stack, nhưng sau đó mình không cần dùng kết quả đó nữa thì mình sẽ pop nó đi.
others: pop2, dup2, swap,...: Slide cũng nhắc đến một số lệnh khác nữa:

pop2: Loại bỏ hai giá trị trên đỉnh stack. Thường dùng cho các kiểu dữ liệu chiếm 2 slot như long và double.
dup2: Nhân đôi hai giá trị trên đỉnh stack (coi như một đơn vị). Dùng cho các kiểu dữ liệu chiếm 2 slot hoặc hai giá trị 1 slot.
swap: Đảo vị trí của hai giá trị trên đỉnh stack. Ví dụ, nếu trên đỉnh stack có value2 rồi đến value1, sau khi swap thì sẽ thành value1 rồi đến value2.

Example 10

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

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

iconst_1: Đẩy giá trị 1 lên Operand Stack. Stack: [1]
dup: Nhân đôi giá trị trên đỉnh stack. Stack: [1, 1]
istore_2: Lấy một giá trị 1 từ đỉnh stack và lưu vào biến cục bộ ở index 2 (biến c). Stack: [1]
dup: Nhân đôi giá trị trên đỉnh stack (vẫn là 1). Stack: [1, 1]
istore_1: Lấy một giá trị 1 từ đỉnh stack và lưu vào biến cục bộ ở index 1 (biến b). Stack: [1]
istore_0: Lấy giá trị 1 cuối cùng từ đỉnh stack và lưu vào biến cục bộ ở index 0 (biến a). Stack: `

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

In MC, not in Java

a = 2
1 + 2

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

iconst_1: Đẩy giá trị 1 lên stack. Stack: [1]
iconst_2: Đẩy giá trị 2 lên stack. Stack: [1, 2]
dup: Nhân đôi giá trị trên đỉnh stack (là 2). Stack: [1, 2, 2]
istore_0: Lấy một giá trị 2 từ đỉnh stack và lưu vào biến cục bộ ở index 0 (biến a). Stack: [1, 2]
iadd: Lấy hai giá trị trên đỉnh stack (2 và 1), cộng chúng lại (2 + 1 = 3), và đẩy kết quả 3 lên stack. Stack: [3]
pop: Loại bỏ giá trị trên đỉnh stack (là 3). Trong trường hợp này, có vẻ như kết quả của phép cộng không được gán cho biến nào nên nó bị loại bỏ.

Cái này liên quan đến việc tạo ra các đối tượng (instances của class) và mảng, cũng như cách truy cập và thay đổi dữ liệu bên trong chúng.

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*.
- …

Create a new class instance: new.

Lệnh new dùng để tạo ra một đối tượng mới của một class nào đó. Khi gặp lệnh này, JVM sẽ cấp phát bộ nhớ cho đối tượng đó trên Heap. Sau lệnh new, một tham chiếu đến đối tượng mới sẽ được đẩy lên Operand Stack.

Create a new array: newarray, anewarray, multianewarray.

newarray: Dùng để tạo một mảng các kiểu dữ liệu nguyên thủy (primitive types) như int, byte, boolean, v.v. Mày cần chỉ định kiểu dữ liệu và kích thước của mảng.

anewarray: Dùng để tạo một mảng các đối tượng (reference types). Mày cần chỉ định kiểu class của các đối tượng trong mảng và kích thước của mảng.

multianewarray: Dùng để tạo các mảng đa chiều. Mày cần chỉ định kiểu dữ liệu và kích thước của từng chiều.

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.

getfield: Dùng để lấy giá trị của một field (biến thành viên) không static (instance field) của một đối tượng. Mày cần một tham chiếu đến đối tượng trên stack, và chỉ định tên và kiểu của field muốn lấy. Giá trị của field sẽ được đẩy lên stack.

putfield: Dùng để gán giá trị cho một field không static của một đối tượng. Mày cần một tham chiếu đến đối tượng và giá trị muốn gán trên stack, cùng với tên và kiểu của field.

getstatic: Dùng để lấy giá trị của một field static (class field) của một class. Mày cần chỉ định class và tên, kiểu của field. Giá trị của field sẽ được đẩy lên stack.

putstatic: Dùng để gán giá trị cho một field static của một class. Mày cần giá trị muốn gán trên stack, cùng với class, tên và kiểu của field.

Load an array component onto the operand stack: baload, caload, saload, iaload, laload, faload, daload, aaload.

Đây là các lệnh để lấy một phần tử từ một mảng và đẩy nó lên Operand Stack. Tên của lệnh cho biết kiểu dữ liệu của phần tử trong mảng (ví dụ: iaload cho mảng int, baload cho mảng byte hoặc boolean, aaload cho mảng tham chiếu). Mày cần tham chiếu đến mảng và index của phần tử trên stack.

Store a value from the operand stack as an array component: bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore.

Đây là các lệnh để lưu một giá trị từ Operand Stack vào một phần tử của mảng. Tương tự như trên, tên lệnh cho biết kiểu dữ liệu của phần tử trong mảng. Mày cần tham chiếu đến mảng, index của phần tử, và giá trị muốn lưu trên stack.

Example 11

BKOOL:

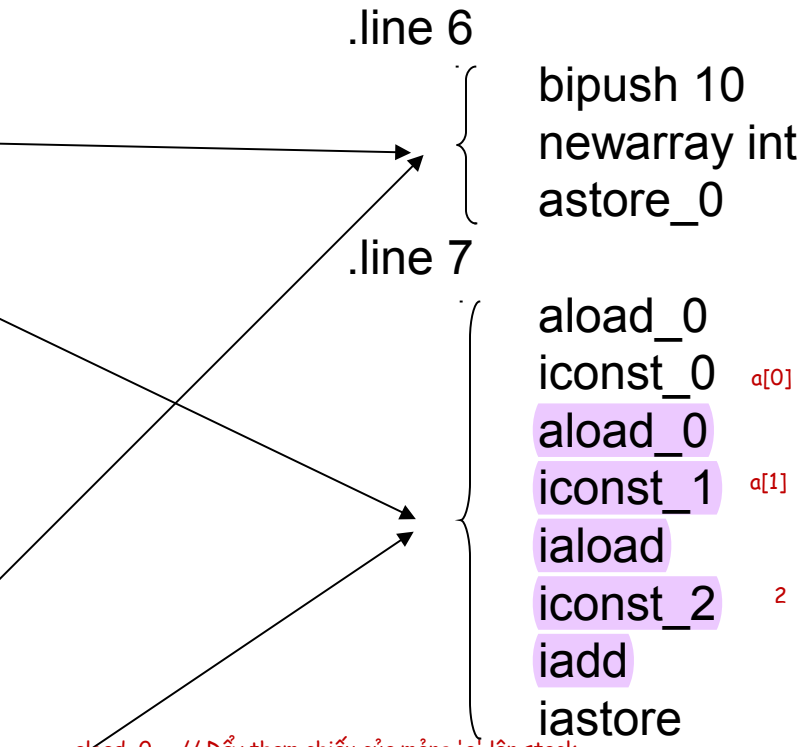
a:integer[10];

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

Java:

int a[] = new int [10];

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



bipush 10: Đẩy giá trị 10 (kích thước mảng) lên Operand Stack.
newarray int: Tạo một mảng mới gồm 10 phần tử kiểu int. Một tham chiếu đến mảng mới này sẽ nằm trên đỉnh stack.
astore_0: Lấy tham chiếu mảng từ stack và lưu vào biến cục bộ ở index 0 (chính là biến a).

```
aload_0    // Đẩy tham chiếu của mảng 'a' lên stack
iconst_0   // Đẩy index 0 lên stack
aload_0    // Đẩy lại tham chiếu của mảng 'a' lên stack
iconst_1   // Đẩy index 1 lên stack
iaload     // Lấy giá trị của a[1] từ mảng và đẩy lên stack
iconst_2   // Đẩy giá trị 2 lên stack
iadd       // Cộng giá trị trên đỉnh stack (a[1] và 2) và đẩy kết quả lên stack
iastore    // Lấy kết quả, index 0, và tham chiếu mảng 'a' từ stack và gán kết quả vào a[0]
```

Field Instructions

- getstatic
- pustatic
- getfield
- putfield
- E.g.

getstatic: Lệnh này dùng để lấy giá trị của một field static (biến tĩnh) của một class. Static field là field thuộc về class chứ không phải là một instance cụ thể của class.
putstatic: Lệnh này dùng để gán giá trị cho một field static của một class.
getfield: Lệnh này dùng để lấy giá trị của một field non-static (instance field) của một đối tượng. Để dùng lệnh này, mày cần có một tham chiếu đến đối tượng trên Operand Stack.
putfield: Lệnh này dùng để gán giá trị cho một field non-static của một đối tượng. Tương tự như getfield, mày cần một tham chiếu đến đối tượng và giá trị muốn gán trên stack.

<field_spec> <descriptor>

dòng getstatic java.lang.System.out Ljava/io/PrintStream ; nó có nghĩa là:

getstatic: Lệnh để lấy giá trị của một static field.
java.lang.System: Tên của class chứa field này.
out: Tên của field mà mình muốn lấy.
Ljava/io/PrintStream;: Đây là descriptor (mô tả kiểu dữ liệu) của field out. Trong trường hợp này, nó cho biết out là một đối tượng thuộc class java.io.PrintStream. Cái chữ L ở đầu là ký hiệu cho một kiểu tham chiếu (reference), và sau đó là tên đầy đủ của class được bao quanh bởi dấu /, kết thúc bằng dấu ;.

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

class name — field name — field type

```
public class VD12 {
  static int a;
  int b;
  static VD12 c;
  VD12 d;

  public static void main(Stringarg) {
    VD12 e;
    e = new VD12();    // Tạo một đối tượng VD12 mới và gán cho e
    a = 1;             // Gán giá trị 1 cho field static a
    e.b = a + 1;       // Lấy giá trị của a, cộng 1, rồi gán cho field b của đối tượng e
    e.d = new VD12();  // Tạo một đối tượng VD12 mới và gán cho field d của đối tượng e
    c = e.d;           // Gán giá trị của field d của đối tượng e cho field static c
  }
}
```

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;
```

invokespecial VD12/<init>() V: Gọi constructor (phương thức khởi tạo) của class VD 12. Lệnh này sẽ lấy một tham chiếu từ stack (cái mà mình vừa dup) và thực hiện việc khởi tạo đối tượng . Chữ V ở cuối descriptor có nghĩa là phương thức này không trả về giá trị (void).

putstatic VD12.a I: Lấy giá trị 1 từ stack và gán nó cho field static a của class VD12. Chữ I trong descriptor có nghĩa là kiểu int.

```
.line 7
new VD12
dup
invokespecial VD12/<init>()V
astore_1       // e = new VD12()

.line 8
iconst_1
putstatic VD12.a I // a = 1

.line 9
aload_1        // Đẩy tham chiếu của e lên stack
getstatic VD12.a I // Lấy giá trị của VD12.a và đẩy lên stack
iconst_1       // Đẩy giá trị 1 lên stack
iadd           // Cộng hai giá trị trên đỉnh stack (a + 1)
putfield VD12.b I // Gán kết quả vào field b của đối tượng e

.line 10
aload_1        // Đẩy tham chiếu của e lên stack
new VD12
dup
invokespecial VD12/<init>()V
putfield VD12.d LVD12; // Gán đối tượng VD12 mới cho field d của đối tượng e

.line 11
aload_1        // Đẩy tham chiếu của e lên stack
getfield VD12.d LVD12; // Lấy giá trị của field d của đối tượng e và đẩy lên stack
putstatic VD12.c LVD12; // Gán giá trị này cho field static c của class VD12
```

.line 7:

new VD12: Tạo một instance mới của class VD12 trên heap. Một tham chiếu đến instance này được đẩy lên Operand Stack.
dup: Nhân đôi tham chiếu trên đỉnh stack. Bây giờ trên stack có hai tham chiếu đến cùng một đối tượng VD12.
invokespecial VD12/<init>()V: Gọi constructor (phương thức khởi tạo) của class VD12. Lệnh này sẽ lấy một tham chiếu từ stack (cái mà mình vừa dup) và thực hiện việc khởi tạo đối tượng. Chữ V ở cuối descriptor có nghĩa là phương thức này không trả về giá trị (void).
astore_1: Lấy một tham chiếu đối tượng từ đỉnh stack và lưu vào biến cục bộ ở index 1 (tương ứng với biến e trong code Java).
.line 8:

iconst_1: Đẩy hằng số nguyên 1 lên stack.
putstatic VD12.a I: Lấy giá trị 1 từ stack và gán nó cho field static a của class VD12. Chữ I trong descriptor có nghĩa là kiểu int.
.line 9:

aload_1: Lấy tham chiếu đến đối tượng e (đã lưu ở biến cục bộ index 1) và đẩy lên stack.
getstatic VD12.a I: Lấy giá trị của field static a từ class VD12 và đẩy lên stack.
iconst_1: Đẩy hằng số nguyên 1 lên stack.
iadd: Lấy hai giá trị trên đỉnh stack (giá trị của a và 1), cộng chúng lại, và đẩy kết quả lên stack.
putfield VD12.b I: Lấy kết quả từ stack và gán nó cho field instance b của đối tượng mà tham chiếu đang ở dưới trong stack (đó là đối tượng e).
.line 10:

aload_1: Lấy tham chiếu đến đối tượng e lên stack.
new VD12: Tạo một instance mới của class VD12 trên heap. Một tham chiếu đến instance này được đẩy lên stack.
dup: Nhân đôi tham chiếu trên đỉnh stack.
invokespecial VD12/<init>()V: Gọi constructor của đối tượng VD12 vừa tạo.
putfield VD12.d LVD12;: Lấy tham chiếu đối tượng vừa tạo từ stack và gán nó cho field instance d của đối tượng e. Chữ LVD12; là descriptor cho biết kiểu của field d là một tham chiếu đến class VD12.
.line 11:

aload_1: Lấy tham chiếu đến đối tượng e lên stack.
getfield VD12.d LVD12;: Lấy giá trị của field instance d của đối tượng e (đó là một tham chiếu đến một đối tượng VD12 khác) và đẩy nó lên stack.
putstatic VD12.c LVD12;: Lấy tham chiếu đối tượng từ stack và gán nó cho field static c của class VD12.

Method Invocation Instructions

invokestatic: Dùng để gọi các phương thức static (tĩnh) của một class. Vì nó là static nên mày không cần một đối tượng cụ thể nào để gọi nó. Mày chỉ cần biết tên class và tên phương thức thôi.

invokevirtual: Dùng để gọi các phương thức non-static (instance method) của một đối tượng. Để gọi được phương thức này, mày cần phải có một tham chiếu đến đối tượng mà mày muốn gọi phương thức đó nằm trên Operand Stack

- invokestatic
- invokevirtual
- invokespecial

<method-spec>

invokespecial: Dùng để gọi các phương thức đặc biệt, bao gồm:

Phương thức khởi tạo (constructor) có tên đặc biệt là <init>.
Các phương thức private của class hiện tại.
Các phương thức của lớp cha (superclass). Tóm lại, nó dùng cho những cái mà thường không được gọi theo kiểu "đa hình" (virtual dispatch).

  - the constructor method <init>
  - a private method
  - a method in a super class

invokeinterface: Dùng để gọi các phương thức được định nghĩa trong một interface. Cái này cần thêm một số thông tin nữa như số lượng tham số của phương thức.

- invokeinterface <method-spec> <num-args>

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

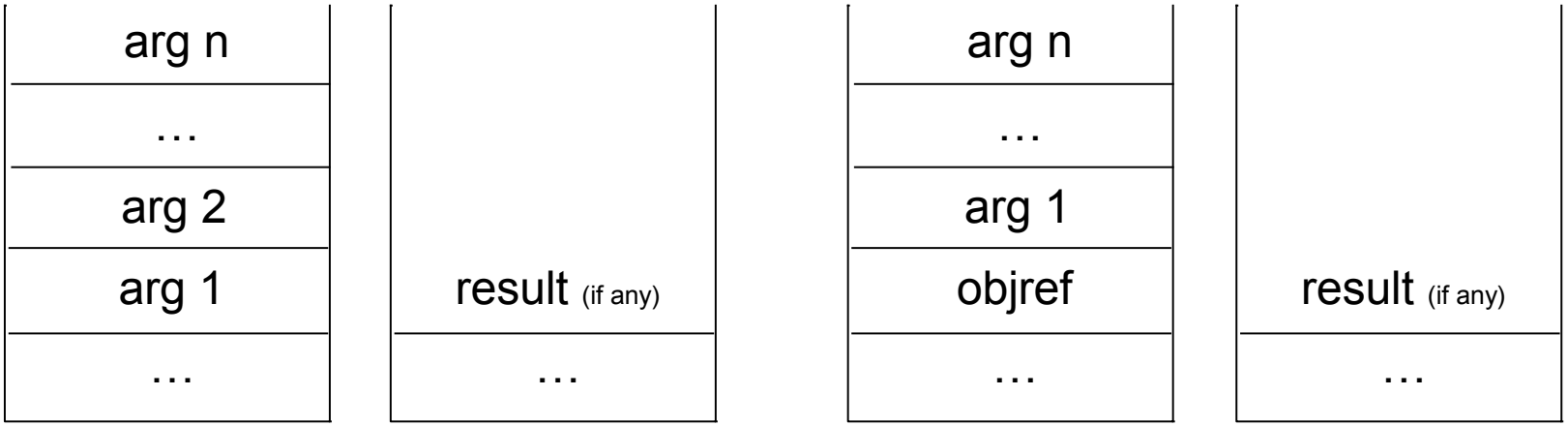
class name — method name — type desc

Sau mỗi lệnh gọi phương thức (trừ invokestatic nếu nó không thuộc về một interface), mày thường sẽ thấy cái <method-spec>. Cái này nó chỉ định cái phương thức mà mày muốn gọi, thường bao gồm tên class, tên phương thức, và một cái gọi là "type descriptor" (mô tả kiểu dữ liệu của các tham số và kiểu trả về của phương thức).

dòng invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V có nghĩa là:

invokevirtual: Mình đang gọi một phương thức instance.
java/io/PrintStream: Đây là class mà phương thức thuộc về.
println: Đây là tên của phương thức mà mình muốn gọi.
(Ljava/lang/String;)V: Đây là type descriptor của phương thức println. Nó cho biết phương thức này nhận một tham số là một đối tượng kiểu java.lang.String (cái L ở đầu là ký hiệu cho object, sau đó là tên class, kết thúc bằng ;) và nó không trả về giá trị gì (V là ký hiệu cho void).

Method Invocation Instructions (cont'd)



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

Chỗ invokestatic:

Trước khi gọi: Trên Operand Stack sẽ có các tham số (arguments) mà mày truyền vào cho cái phương thức static đó (từ arg 1 đến arg n).
Sau khi gọi: Các tham số này sẽ bị "pop" ra khỏi stack, và nếu cái phương thức static đó có trả về giá trị gì thì cái giá trị kết quả (result) sẽ được đẩy lên lại stack.
Quan trọng: Slide có nhắc lại là invokestatic là dựa trên static class. Tức là khi gọi phương thức static, JVM biết chắc chắn cái phương thức đó thuộc về class nào, không cần quan tâm đến instance cụ thể nào hết.

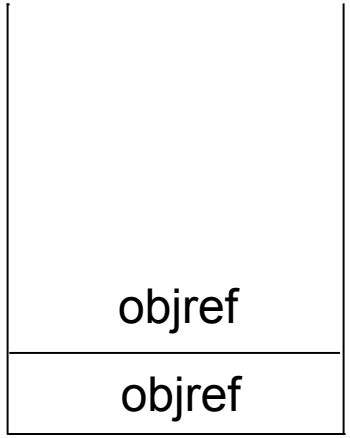
Chỗ invokevirtual / invokespecial:
Trước khi gọi: Trên Operand Stack sẽ có các tham số (từ arg 1 đến arg n) và thêm một cái objref ở dưới. Cái objref này chính là tham chiếu đến đối tượng mà mày muốn gọi phương thức instance (non-static) hoặc phương thức đặc biệt (như constructor) trên đó.
Sau khi gọi: Tương tự như invokestatic, các tham số và cái objref sẽ bị pop ra, và nếu phương thức có trả về giá trị thì result sẽ được đẩy lên.
Quan trọng: Slide nhấn mạnh là invokevirtual là dựa trên real type of objref. Cái này có nghĩa là khi gọi một phương thức bằng invokevirtual, JVM sẽ xác định xem cái đối tượng thực tế mà objref đang trỏ tới thuộc class nào, rồi nó sẽ gọi cái phiên bản phương thức phù hợp với class đó (đây chính là cơ chế đa hình trong Java).

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());
}
```

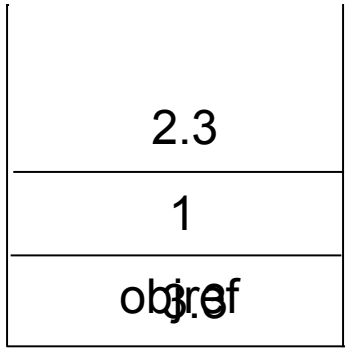



```
.method public static main([Ljava/lang/String;)V
.limit stack 2
.limit locals 1
.var 0 is arg0 [Ljava/lang/String; from Label0 to Label1

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

.line 4
      return

.end method
```

.method public static main([Ljava/lang/String;)V: Khai báo phương thức main, nó là public, static, trả về void (V), và nhận một mảng các String ([Ljava/lang/String;) làm tham số.
.limit stack 2: Chỉ định kích thước tối đa của Operand Stack mà phương thức này cần là 2.
.limit locals 1: Chỉ định số lượng biến cục bộ mà phương thức này sử dụng là 1. Ở đây, biến cục bộ index 0 sẽ là mảng String arg.
.var 0 is arg0 [Ljava/lang/String; from Label0 to Label1: Khai báo biến cục bộ ở index 0 là arg0 (tên do Jasmin đặt), có kiểu là mảng String, và phạm vi của nó từ label Label0 đến Label1 (trong trường hợp này là toàn bộ phương thức main).
.line 3: Chỉ ra rằng dòng bytecode tiếp theo tương ứng với dòng số 3 trong file Java.
new VD13: Tạo một instance mới của class VD13 trên heap. Một tham chiếu đến instance này được đẩy lên Operand Stack. Stack: [objref]
dup: Nhân đôi tham chiếu trên đỉnh stack. Stack: [objref, objref]
invokespecial VD13/<init>()V: Gọi phương thức khởi tạo (constructor) của class VD13. Lệnh này lấy một tham chiếu đối tượng từ stack và thực hiện việc khởi tạo. Sau lệnh này, stack còn lại một tham chiếu. Stack: [objref]
invokestatic VD13/goo(LVD13;)V: Đây chính là lệnh gọi phương thức goo.
invokestatic: Vì goo là một phương thức static.
VD13/goo: Chỉ định class là VD13 và phương thức là goo.
(LVD13;)V: Đây là method descriptor. Nó cho biết phương thức goo nhận một tham số là một tham chiếu đến đối tượng kiểu VD13 (LVD13;) và trả về void (V). Lệnh này sẽ lấy tham chiếu đối tượng VD13 (mà mình vừa tạo và còn trên stack) làm tham số cho phương thức goo. Sau khi gọi, stack sẽ trống nếu goo không trả về giá trị.
.line 4: Chỉ ra rằng dòng bytecode tiếp theo tương ứng với dòng số 4 trong file Java.
return: Trả về từ phương thức main. Vì main được khai báo là void, nó không trả về giá trị gì.
.end method: Kết thúc định nghĩa của phương thức main.

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 static goo(LVD13;)V: Khai báo phương thức goo, nó là static, trả về void (V), và nhận một tham số là một tham chiếu đến đối tượng kiểu VD13 (LVD13;). Cái tham số này sẽ được lưu ở biến cục bộ index 0.
.limit stack 3: Chỉ định kích thước tối đa của Operand Stack mà phương thức này cần là 3.
.limit locals 1: Chỉ định số lượng biến cục bộ mà phương thức này sử dụng là 1 (để chứa tham số x).
.var 0 is arg0 LVD13; from Label0 to Label1: Khai báo biến cục bộ ở index 0 là arg0 (tên do Jasmin đặt), có kiểu là tham chiếu đến VD13, và phạm vi của nó từ label Label0 đến Label1 (toàn bộ phương thức goo).
.line 9: Chỉ ra rằng dòng bytecode tiếp theo tương ứng với dòng số 9 trong file Java (x.foo(1, 2.3F);).
aload_0: Lấy giá trị của biến cục bộ ở index 0 (là tham chiếu đến đối tượng VD13 mà mình đặt tên là x trong Java) và đẩy nó lên Operand Stack. Để gọi một phương thức instance (như foo), mình cần đối tượng mà mình muốn gọi phương thức đó trên stack.
iconst_1: Đẩy hằng số nguyên 1 lên stack. Đây là tham số đầu tiên (int a) cho phương thức foo.
ldc 2.3: Đẩy hằng số thực 2.3 lên stack. Đây là tham số thứ hai (float b) cho phương thức foo. Chữ F ở cuối 2.3F trong Java cho biết nó là kiểu float. Trong bytecode, hằng số float thường được load bằng lệnh ldc.
invokevirtual VD13/foo(IF)F: Đây là lệnh gọi phương thức foo.
invokevirtual: Vì foo là một phương thức instance (không static).
VD13/foo: Chỉ định class là VD13 và phương thức là foo.
(IF)F: Đây là method descriptor. Nó cho biết phương thức foo nhận một tham số kiểu int (I) và một tham số kiểu float (F), và nó trả về một giá trị kiểu float (F). Lệnh này sẽ lấy tham chiếu đối tượng VD13, giá trị int (1), và giá trị float (2.3) từ stack, gọi phương thức foo trên đối tượng đó với các tham số đã cho. Kết quả trả về (một số float) sẽ được đẩy lên lại stack.
pop: Lệnh này lấy giá trị trên đỉnh stack (là kết quả trả về của phương thức foo) và loại bỏ nó. Trong trường hợp này, giá trị trả về của foo không được sử dụng ở đâu cả.
.line 10: Chỉ ra rằng dòng bytecode tiếp theo tương ứng với dòng số 10 trong file Java (dấu đóng ngoặc của phương thức goo).
return: Trả về từ phương thức goo. Vì goo được khai báo là void, nó không trả về giá trị gì.
.end method: Kết thúc định nghĩa của phương thức goo.

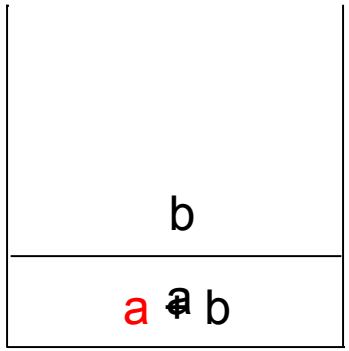
Method Return

- 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

Tất cả các phương thức trong Java đều kết thúc bằng một lệnh return. Tùy thuộc vào kiểu dữ liệu mà phương thức trả về (hoặc nếu nó không trả về gì), mình sẽ dùng các lệnh return khác nhau:
return: Dùng cho các phương thức mà được khai báo là void (không trả về giá trị). Khi gặp lệnh này, phương thức sẽ kết thúc và không có giá trị nào được đẩy lên Operand Stack.
ireturn: Dùng cho các phương thức trả về giá trị có kiểu là int, short, char, boolean, hoặc byte. Trước khi gọi lệnh này, giá trị trả về (kiểu int hoặc các kiểu nhỏ hơn được tự động ép kiểu lên int) phải nằm trên đỉnh Operand Stack.
freturn: Dùng cho các phương thức trả về giá trị có kiểu là float. Giá trị float trả về phải nằm trên đỉnh Operand Stack.
lreturn: Dùng cho các phương thức trả về giá trị có kiểu là long. Giá trị long (chiếm 2 slot trên stack) trả về phải nằm trên đỉnh Operand Stack.
dreturn: Dùng cho các phương thức trả về giá trị có kiểu là double. Giá trị double (chiếm 2 slot trên stack) trả về phải nằm trên đỉnh Operand Stack.
areturn: Dùng cho các phương thức trả về một giá trị là một tham chiếu (reference) đến một đối tượng (ví dụ như một instance của một class, một array, v.v.). Tham chiếu trả về phải nằm trên đỉnh Operand Stack.

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
```

.method foo(IF)F: Khai báo phương thức foo, nó là một instance method (không có static), nhận một tham số kiểu int (I) và một tham số kiểu float (F), và trả về một giá trị kiểu float (F).
.limit stack 2: Chỉ định kích thước tối đa của Operand Stack mà phương thức này cần là 2.
.limit locals 3: Chỉ định số lượng biến cục bộ mà phương thức này sử dụng là 3. Biến cục bộ index 0 là this (tham chiếu đến đối tượng VD13 hiện tại), index 1 là tham số a (kiểu int), và index 2 là tham số b (kiểu float).
.var 0 is this LVD13; from Label0 to Label1: Khai báo biến cục bộ ở index 0 là this, có kiểu là tham chiếu đến VD13, và phạm vi của nó từ label Label0 đến Label1 (toàn bộ phương thức foo).
.var 1 is argo I from Label0 to Label1: Khai báo biến cục bộ ở index 1 là argo (tên do Jasmin đặt), có kiểu là int (I), và phạm vi từ Label0 đến Label1. Đây tương ứng với tham số a trong code Java.
.var 2 is arg1 F from Label0 to Label1: Khai báo biến cục bộ ở index 2 là arg1 (tên do Jasmin đặt), có kiểu là float (F), và phạm vi từ Label0 đến Label1. Đây tương ứng với tham số b trong code Java.
Label0:: Một label đánh dấu vị trí bắt đầu của code trong phương thức.
iload_1: Lấy giá trị của biến cục bộ ở index 1 (là giá trị của tham số a kiểu int) và đẩy nó lên Operand Stack.
i2f: Lấy giá trị int trên đỉnh stack (là giá trị của a) và chuyển đổi nó thành giá trị float. Kết quả float này được đẩy trở lại stack.
fload_2: Lấy giá trị của biến cục bộ ở index 2 (là giá trị của tham số b kiểu float) và đẩy nó lên Operand Stack . Bây giờ trên stack có hai giá trị float (giá trị của a sau khi chuyển đổi và giá trị của b).
fadd: Lấy hai giá trị float trên đỉnh stack, cộng chúng lại, và đẩy kết quả (kiểu float) trở lại stack. Đây chính là kết quả của phép toán a + b.
Label1:: Một label đánh dấu vị trí kết thúc (trước khi return) của code trong phương thức.
freturn: Lấy giá trị float trên đỉnh stack (là kết quả của phép cộng) và trả về nó. Lệnh này kết thúc việc thực thi phương thức foo.
.end method: Kết thúc định nghĩa của phương thức foo.

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>

.source <source.java>: Chỉ thị này dùng để chỉ định tên của file source Java gốc (nếu có) mà từ đó file .class này được tạo ra. Ví dụ: .source VD14.java.

.class <the current class>: Chỉ thị này dùng để khai báo tên và các thuộc tính (ví dụ: public, final, abstract) của class mà mày đang định nghĩa. Ví dụ: .class public VD14.

.super <the super class>: Chỉ thị này dùng để chỉ định lớp cha (superclass) của class hiện tại. Ví dụ: .super java/lang/Object.

.limit: Chỉ thị này thường đi kèm với stack hoặc locals để chỉ định kích thước tối đa của Operand Stack và số lượng biến cục bộ mà một phương thức sẽ sử dụng. Ví dụ: .limit stack 3, .limit locals 1.

.method <the method description>: Chỉ thị này dùng để khai báo một phương thức, bao gồm tên, các thuộc tính (ví dụ: public, static), kiểu trả về và kiểu của các tham số. Ví dụ: .method public static main([Ljava/lang/String;)V.

.field <the field description>: Chỉ thị này dùng để khai báo một field (biến thành viên) của class, bao gồm tên, các thuộc tính (ví dụ: public, static), và kiểu dữ liệu. Ví dụ: .field a I, .field static b I.

.end: Chỉ thị này thường được dùng để đánh dấu sự kết thúc của một phương thức (.end method) hoặc một số cấu trúc khác.

.var <the variable description>: Chỉ thị này dùng để khai báo một biến cục bộ trong một phương thức, bao gồm index của biến, tên (thường để dễ đọc), kiểu dữ liệu, và phạm vi (từ label nào đến label nào). Ví dụ: .var 0 is arg0 [Ljava/lang/String; from Label0 to Label1.

.line <the line number in source code>: Chỉ thị này dùng để chỉ ra số dòng trong file source code Java tương ứng với dòng bytecode tiếp theo. Cái này hữu ích cho việc debug. Ví dụ: .line 5.

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/