The Java Virtual Machine

The Java™ programming language is a high-level language developed by Sun Microsystems, now a wholly owned subsidiary of Oracle, Inc.

The Java language is neither interpreted nor is it truly compiled into machine language. Instead, the language is compiled into a byte code form for interpretation by the **JVM** (Java Virtual Machine).

Strictly speaking, the term "Java Virtual Machine" was created by Sun Microsystems, Inc. to refer to the abstract specification of a computing machine designed to run Java programs by executing the bytecode.

As in many similar cases, the name is also applied to any software that actually implements the specification.

Many books might make claims such as "The Java Virtual Machine (JVM) is the software that executes compiled Java bytecode … the name given to the machine language inside compiled Java programs."

This lecture presents a brief description of part of the JVM.

The JVM Execution Engine

The JVM is built around two key data structures.

1. The **local variable array**, constructed as a 0-based array of 32-bit entries. Note that the standard Java language array is 0-based.

2. The **operand stack**, implemented as a standard LIFO (Last In First Out) stack of 32-bit items.

Unlike the Java language, the JVM uses only two basic storage types: 32-bit words and 64-bit double words. Double words occupy two slots in either the operand stack or the local variable array.

Values for the 8-bit, 16-bit, and 32-bit types in the Java language are stored in 32-bit single-word slots in either the operand stack or local variable array.

Each variable at the Java language level is referenced at the JVM level by its index in the local variable array. Interaction between the variable array and operand stack is handled by a number of instructions that replace push and pop.

iload_1 // Push the integer value in storage location 1 onto the stack

istore_2 // Pop the value off the stack and store as an integer in storage location 2.

The ADT (Abstract Data Type) Stack

The ADT stack is a LIFO (Last In First Out) data structure with very few operators.

As in the JVM case, we assume that all entries on the stack are 32 bits in length.

This brief discussion will use two of the stack operators found in the IA-32 design. These are the typical push and pop operators.

`PUSH X` Copy the value from address X and push it onto the stack.

`POP Y` Pop the top value from the stack and store it into address Y.

Logically, the stack has a top onto which entries are pushed and from which entries are removed.

At the implementation level, a stack is based on a **stack pointer** (SP or ESP for IA-32).

The precise correlation between the SP and the address of the stack top is very important for the implementation, but not so much for our discussion.

All we demand is that there is a consistent implementation.

Stack Example

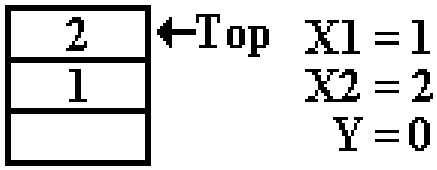
We begin by assuming the following associations of values with locations.

Location	Value
X1	1
X2	2
Y	0

In this sequence, we show only that part of the stack that is directly relevant.

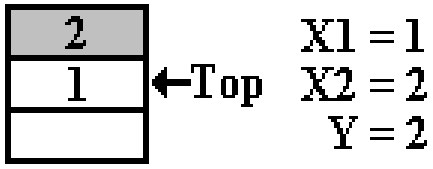
PUSH X1 ; Push the value at location X1 onto the stack.

PUSH X2 ; Push the value at location X2 onto the stack.

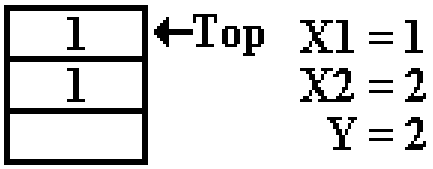


Stack Example (Part 2)

POP Y ; Pop the value from the stack and place into Y.
; The value 2 is no longer a part of the stack.



PUSH X1 ; Push the value at location X1 onto the stack.



Note that popping a value from the stack does not remove that from memory, just from the logical data structure.

A value could be erased after popping by a sequence such as the following.

```
POP  Y    ; Pop the value from the stack.
PUSH 0    ; Overwrite its old stack location.
POP  T    ; Then pop the zero from the stack.
```

The JVM Operand Stack

The JVM operand stack is handled fairly much as the abstract data type.
Here are the key differences.

1. The push operation is replaced by a variety of load instructions
   and constant instructions.

2. The pop operation is augmented by a variety of store operations.
   At least one class of store instructions pops two 32-bit words from the stack.

3. There are a number of operations that pop two operands at a time.

4. There are a number of operations that pop two operands from the stack,
   do a computation or comparison, and then push one result.

The Java Basic Data Types

The primitive data types for the JVM are those for the Java language. Here is a table.

Data Type	**Bytes**	**Format**	**Range**
char	2	Unicode character	\u0000 to \uFFFF *
byte	1	Signed Integer	–128 to + 127
short	2	Signed Integer	– 32768 to + 32767
int	4	Signed Integer	– 2147483648 to + 2147483647
long	8	Signed Integer	$-(2^{63})$ to $2^{63} - 1$
float	4	IEEE Single Precision Float	Magnitude: $1.40 \bullet 10^{-45}$ to $3.40 \bullet 10^{38}$, and zero
double	8	IEEE Double Precision Float	Magnitude: $4.94 \bullet 10^{-3224}$ to $1.798 \bullet 10^{308}$, and zero

* The character with 16–bit code 0x0000 through that with 16–bit code 0xFFFF.

The 8-byte types (long and double) are stored in two JVM 32-bit words.

All other Java language types are stored in one JVM 32-bit word.

Double JVM Word Examples

Consider the following two Java byte code instructions

dadd pop two double–precision floating point values from the stack, add them, and push the result back onto the stack

ladd pop two long (64 bit) signed integer values from the stack, add them, and push the result back onto the stack.

In either case, we have the following stack conditions.

	Before	**After**
Top of Stack	Value1–Word1	Result–Word1
	Value1–Word2	Result–Word2
	Value2–Word1	
	Value2–Word1	

	Something else	

Java Bytecode

A typical JVM instruction comprises a one-byte operation code, followed by zero to three bytes for the operands.

Because each operation code occupies one byte,

1. The JVM language is called bytecode, and
2. There are a maximum of 256 simple opcodes.

The first three classes of JVM instructions for consideration are

instructions to load a constant value onto the stack,

instructions to load a local variable value onto the stack, and

instructions to pop a value and store it into the local variable array.

Loading Constant Values

Each of the following one-byte operators pushes a different constant onto the stack.

Opcode	Instruction	Comment
0x02	**iconst_m1**	Push the integer constant –1 onto the stack.
0x03	**iconst_0**	Push the integer constant 0 onto the stack
0x04	**iconst_1**	Push the integer constant 1 onto the stack
0x05	**iconst_2**	Push the integer constant 2 onto the stack
0x06	**iconst_3**	Push the integer constant 3 onto the stack
0x07	**iconst_4**	Push the integer constant 4 onto the stack
0x08	**iconst_5**	Push the integer constant 5 onto the stack

This set of constant values is a bit richer than expected.

Specifically, the instructions to push 3, 4, and 5 are a bit of a surprise.

Loading Local Variables

There are two variants of the **`iload`** instruction.

There are instructions to load from a general local variable array location.

There is one instruction dedicated to each of the first four array locations.

Here are the four one-byte dedicated load instructions. None of these has an argument.

Opcode	Instruction	Comment
0x1A	**`iload_0`**	Copy the integer in location 0 and push onto the stack.
0x1B	**`iload_1`**	Copy the integer in location 1 and push onto the stack.
0x1C	**`iload_2`**	Copy the integer in location 2 and push onto the stack.
0x1D	**`iload_3`**	Copy the integer in location 3 and push onto the stack.

The other variant of the **`iload`** instruction has opcode 0x15. Its appearance in byte code is as the 2 byte entry **`0x15 NN`**, where **`NN`** is the hexadecimal value of the variable index.

Thus the code **`0x15 0A`** would be read as
"push the value of the integer in element 10 of the variable array
onto the stack". Remember that 0x0A is the hexadecimal representation of decimal 10.

Storing Local Variables

There are two variants of the **`istore`** instruction,

one for general indices into the local variable array and

one dedicated to the first four elements of this array.

Here are the four dedicated store instructions each with its one byte opcode.

None of these has an argument.

Opcode	Instruction	Comment
0x3B	**`istore_0`**	Pop the integer and store into location 0.
0x3C	**`istore_1`**	Pop the integer and store into location 1.
0x3D	**`istore_2`**	Pop the integer and store into location 2.
0x3E	**`istore_3`**	Pop the integer and store into location 3.

The other variant of the **`istore`** instruction has opcode 0x36.

Its appearance in byte code is as the 2 byte entry **`0x36 NN`**,
where **`NN`** is the hexadecimal value of the variable index.

Sample JVM Coding

With all of that said, here is the Java source code example.

We shall examine its coding at the JVM bytecode level.

```
int a = 3 ;
int b = 2 ;
int sum = 0 ;
sum = a + b ;
```

Here is the local variable table for this example.

Note that we make the reasonable assumption that table space is reserved for the variables in the order in which they were declared.

Location Index	Variable stored
0	a
1	b
2	sum

What follows is the Java bytecode for the above simple source language sequence.

In this, we add a number of comments, each in the Java style, to help the reader make the connection.

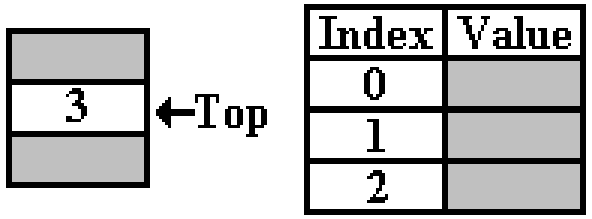
Each Java language statement is now shown as a comment.

The JVM Bytecode

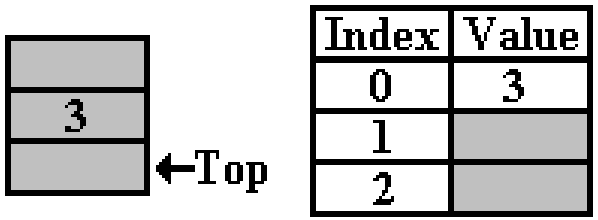
```
//   int a = 3 ;
     iconst_3   // Push the constant value 3 onto the stack
     istore_0   // Pop the value and store in location 0.
//   int b = 2 ;
     iconst_2   // Push the constant value 2 onto the stack
     istore_1   // Pop the value and store in location 1
//   int sum = 0 ;
     iconst_0   // Push the constant value 0 onto the stack
     istore_2   // Pop the value and store in location 2
//   sum = a + b ;
     iload_0    // Push value from location 0 onto stack
     iload_1    // Push value from location 1 onto stack
     iadd       // Pop the top two values, add them,
                // and push the value onto the stack.
     istore_2   // Pop the value and store in location 2.
```

The Execution Illustrated

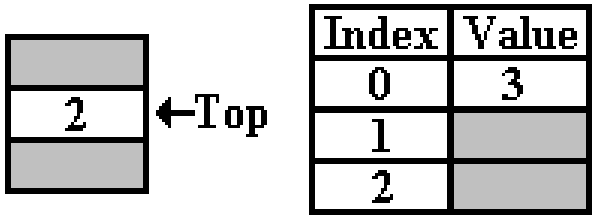
```
//    int a = 3 ;
      iconst_3   // Push the constant value 3 onto the stack
```



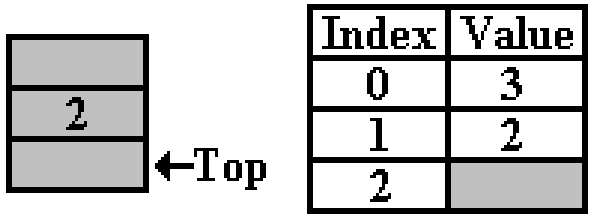

```
      istore_0   // Pop the value and store in location 0.
```



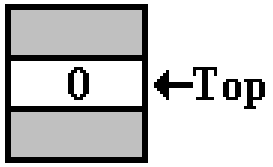
```
//    int b = 2 ;
      iconst_2   // Push the constant value 2 onto the stack
```




```
      istore_1       // Pop the value and store in location 1
```

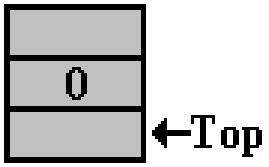


```
//    int sum = 0 ;
      iconst_0   // Push the constant value 0 onto the stack
```



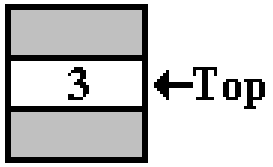
Index	Value
0	3
1	2
2	

```
      istore_2       // Pop the value and store in location 2
```



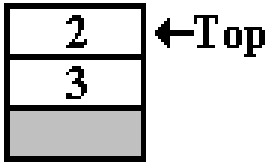
Index	Value
0	3
1	2
2	0

```
//    sum = a + b ;
      iload_0        // Push value from location 0 onto stack
```



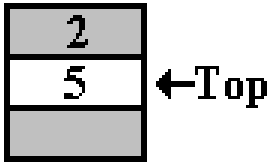
Index	Value
0	3
1	2
2	0

```
      iload_1        // Push value from location 1 onto stack
```



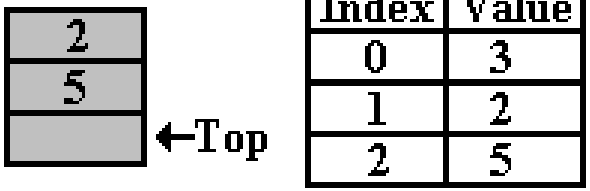
Index	Value
0	3
1	2
2	0

```
      iadd           // Pop the top two values, add them,
                     // and push the value onto the stack.
```



Index	Value
0	3
1	2
2	0

```
      istore_2       // Pop the value and store in location 2.
```



Integer Arithmetic Operators in JVM

Opcode	Instruction	Comment
0x60	iadd	Pop the top two values, add them, and push the result onto the stack.
0x64	isub	Pop the two values, subtract the top one from the second one (next to top on the stack before subtraction), and push the result onto stack.
0x68	imul	Pop the two values, multiply them, and push the result onto the stack.
0x6C	idiv	Pops the two values, divide the value that was second from top by the value that was at the top of the stack, truncates the result to the nearest integer, and pushes the result onto the stack.

The Example and Its Bytecode

Here is the original fragment of Java source code.

```
int a = 3 ;
int b = 2 ;
int sum = 0 ;
sum = a + b ;
```

Here is the ten-byte sequence of Java bytecode that will be executed by the JVM.

```
06 3B 05 3C 03 3D 1A 1B 60 3D
```

The Conditional Instructions

The JVM divides these operations into two classes: conditional branch instructions and comparison instruct ions. The division between the two classes is based on operand type.

In the JVM, the **conditional each branch** instructions pops one value from the stack, examines it, and branches accordingly.

There are fourteen instructions in this class. Each of these fourteen is a three-byte instruction, with the following form.

ByteType	**Range**	**Comment**	
1	unsigned 1-byte integer field	0 to 255	8-bit opcode
2, 3	signed 2-byte integer field	-32768 to 32767	branch offset

The branch target address is computed as (***pc*** **+** ***branch_offset***),
where ***pc*** references the address of the opcode of the branch instruction,
and ***branch_offset*** is a 16-bit signed integer indicating the offset of the branch target.

In the JVM, the **comparison operators** pop two values from the stack, compare them, then push a single integer onto the top of the stack to indicate the result.

The result can then be tested by a conditional branch instruction.

The Conditional Branches (Part 1)

The first set of instructions to discuss cover the Java null object.

Opcode	Instruction	Comment
0xC6	ifnull	This pops the top item from the operand stack. If it is a reference to the special object **`null`**, the branch is taken.
0xC7	ifnonnull	The branch is taken if the item does not reference **`null`**.

The following twelve instructions treat each numeric value as a 32-bit signed integer. While the Java language supports signed integers with length of 8 bits and 16 bits, it appears that each of these types is extended to 32 bits when pushed onto the stack.

Each of the instructions in the next set pops a single item off the stack, examines it as a 32-bit integer value, and branches accordingly.

Opcode	Instruction	Comment
0x99	ifeq	jump if the value is zero
0x9E	ifle	jump if the value is zero or less than zero (not positive)
0x9B	iflt	jump if the value is less than zero (negative)
0x9C	ifge	jump if the value is zero or greater than zero (not negative)

0x9D	ifgt	jump if the value is greater than zero (positive)
0x9A	ifne	jump if the value is not zero

Two Operand Conditional Branch Instructions

Each of the next instructions pops two items off the stack, compares them as 32-bit signed integers, and branches accordingly.

For each of these instructions, we assume that the stack status before the execution of the branch instruction is as follows.

V1 the 32-bit signed integer at the top of the stack
V2 the 32-bit signed integer next to the top of the stack.

Opcode	Instruction	Comment
0x9F	if_icmpeq	Branch if V1 == V2
0xA2	if_icmpge	Branch if V2 ≥ V1
0xA3	if_icmpgt	Branch if V2 > V1
0xA4	if_icmple	Branch if V2 ≤ V1
0xA1	if_icmplt	Branch if V2 < V1
0xA0	if_icmple	Branch if V1 ≠ V2

Comparison Instructions

In the JVM, the comparison operators pop two values from the stack, compare them, and then push a single integer onto the top of the stack to indicate the result. Assume that, prior to the execution of the comparison operation; the state of the stack is as follows.

V1_Word 1
V1_Word 2
V2_Word 1
V2_Word 2

Here is a description of the effect of this class of instructions.

Result of comparison	Value Pushed onto Stack
V1 > V2	– 1
V1 = V2	0

V1 < V2	1

Comparison Instructions (Part 2)

Here is a list of the comparison instructions.

Opcode	Instruction	Comment
0x94	lcmp	Takes long integers from the stack (each being two words, four stack entries are popped), compares them, and pushes the result.
0x97 0x98	fcmpl fcmpg	Each takes single–precision floats from the stack (two stack entries in total), compares them and pushes the result. The values +0.0 and –0.0 are treated as being equal.
0x95 0x96	dcmpl dcmpg	Each takes double–precision floats from the stack (four stack entries in total), compares them and pushes the result. The values +0.0 and –0.0 are treated as being equal.

Note that there are two comparison operations for each floating point type. This is based on how one wants to compare the IEEE standard value NaN (Not a Number, use for results of arithmetic operations such as 0/0).

The **fcmpl** and **dcmpl** instructions return –1 if either value is NaN.

The **fcmpg** and **dcmpg** instructions return 1 if either value is NaN.