Chapter 7D – The Java Virtual Machine

This sub–chapter discusses another architecture, that of the **JVM** (**J**ava **V**irtual **M**achine). In general, a **VM** (**V**irtual **M**achine) is a hypothetical machine (implemented in either hardware or software) that directly executes a specific language. The JVM is a machine that directly executes **Java byte code**, which is the output of the Java compiler. While it is reliably rumored that a hardware implementation of the JVM has been designed, all existing commercial Java virtual machines are implemented as software running on top of other machines. It is worth note that the term "Java" in the context of a programming language is a trademark of the Oracle Corporation, which acquired Sun Microsystems on January 27, 2010.

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 [R024, R025]. Correct implementations of the JVM are required to support the **syntax** of the language (how to form expressions), as well as the **semantics** (what these expressions actually do).

As in many similar cases, the name is also applied to any software that actually implements the specification. So 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."[page 321, R019].

The term "Java" is used primarily in two contexts, that of a Java applet and that of a Java application. A **Java application** appears to be a standard program, which may or may not have a graphical appearance. A **Java applet** is run using a Java–enabled web browser.

When a Java method executes, it has its own stack frame, which is managed by the JVM. The two areas of the stack frame of interest to us are 1) the operand stack and 2) the local variable storage. As we shall see in a future lecture, a **stack frame** is a dynamically allocated section of memory used to store values for the execution of a particular method. While the stack frame has many of the attributes of a stack, it is accessed using methods other than the standard stack operators. On the other hand, the operand stack is handled much as the pure data type.

The Java Operand Stack

The operand stack holds operands for the Java instructions. Each entry in the stack occupies four bytes (32 bits), even those holding shorter data types. The operand stack is handled basically as a **LIFO** (**L**ast **I**n **F**irst **O**ut) data structure, except that certain instructions will pop two items at a time.

The local variable storage for a method is handled very much as a zero–based array. At the JVM level, each variable is identified by its index in this array. Interaction between the operand stack and local variable storage is illustrated by the following Java byte code examples, which will require more explanation.

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

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

We shall explain these instructions more when we discuss stack operations.

Operations on the JVM Operand Stack

Here we review the basic operations on a stack, viewed from both the **ADT** (**A**bstract **D**ata **T**ype) approach and also from the modified approach taken by the JVM. As an ADT, a stack is a LIFO data structure with only a few basic operations. The two basic operations we shall consider are Push and Pop. These are described first using the IA–32 assembler syntax.

```
PUSH X    ; Push the value at location X onto the stack.

POP  Y    ; Pop the last value pushed onto the stack
          ; and store it into the location Y.
```

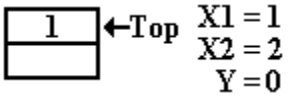
The description of the stack as a LIFO data structure illustrates its basic operation. Whatever is last put on the stack is the first removed. We shall adopt the standard terminology that any item is added to the top of the stack and removed from that location. While any correct stack implementation must be based on a **SP** (**S**tack **P**ointer), we shall not use that detail here. In the IA–32 architecture, the 32–bit stack pointer is called **ESP**.

Here is a short sequence of stack operations, just to serve as a review for the ADT.
We begin by assuming the following associations of values with locations.

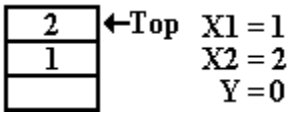
Location	Value
X1	1
X2	2
Y	0

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

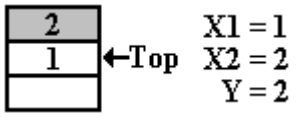
```
PUSH X1   ; Push the value at location X1 onto the stack.
```



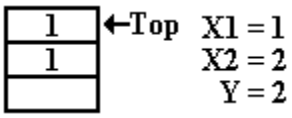

```
PUSH X2   ; Push the value at location X2 onto the stack.
```




```
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.
```



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.
2. The pop operation is augmented by a variety of store operations.
3. There are a number of operations that pop two operands at a time.
4. Some of the above operations pop two operands and then push one result.

Java Primitive 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.

While the primitive data types are as expected, the JVM storage is based on 32–bit (4 byte) entities. Specifically, the operand stack is organized into 32–bit slots and the local variable storage is considered as an array of 32–bit words of no particular data type.

The correspondence between the Java language primitive data types and the storage allocations of the JVM is simple: the two 64–bit data types (long and double) are each stored in two 32–bit entries in the local variable array and stored as two entries on the operand stack. All other data types are assigned one 32–bit storage location.

Put another way, from the JVM storage view, each data type is either a single word or a double word. The JVM instructions interpret these as appropriate. 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
Value1–Word1	Result–Word1
Value1–Word2	Result–Word2
Value2–Word1	
Value2–Word1	

Something else	

It is interesting to note that Java stores all multi–byte values in big–endian order, as opposed to the Intel architectures, which use little–endian order. Big–endian order is also called "**network order**", in that it is used for values transmitted over the global Internet. As the JVM provides no instructions for direct access of the memory of the base hardware (Intel, Motorola, Sparc, etc.), this hardware independence causes no problem.

A Sample of Some Instructions

At this time, it might be convenient to show a sample of Java source code and discuss its representation as instructions at the bytecode level. To avoid confusion, this author finds it expedient to discuss the bytecode instructions before using them.

In Java bytecodes, each instruction contains a one–byte opcode followed by zero or more operands. The name "**bytecode**" comes from the size of the opcode. It appears that the JVM can accommodate at most $2^8 = 256$ different opcodes; though there is a way around this limit.

The following example shows two ways to load integers; that is, push the values onto the operand stack. The **`iload`** instructions load from the indexed variable storage, and an interesting variety of **`iconst`** instructions load constant values. We first examine the **`iconst`** instructions, noting that each has a separate opcode.

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

There are two variants of the **`iload`** instruction, one for general indices into the local variable array and one dedicated to the first four elements of this array. Here we focus on the latter instructions, each with its one byte opcode. 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 integer in element 10 of the variable array onto the stack". Remember that 0x0A is the hexadecimal representation of decimal 10.

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 we focus on the latter 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.

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

```
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

Here 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.

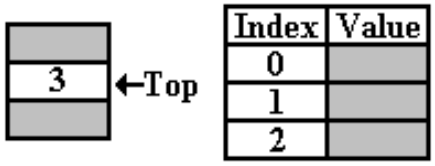
```
//   int a = 3 ;
     iconst_3         // Push the constant value 3 onto the stack
     istore_0         // Pop the value and store in location 0.
//   int b = 2 ;
     iconst_2         // Push the constant value 2 onto the stack
     istore_1         // Pop the value and store in location 1
//   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.
```

Just to be very explicit about this, we shall show the state of the operand stack and variable array as the bytecode above is executed.

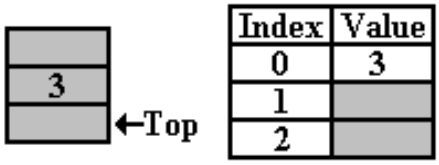
In the diagrams below, the gray color in a box indicates that the entry is not valid logically within the current context. For example, following the execution of the **`iconst_3`** instruction, the stack has only one valid entry, which contains the value 3. Any entry on the stack that is "below" the current entry might have meaning within another context, but not in this one.

Similarly, there are values in the variable array, but the values have no meaning in this context.

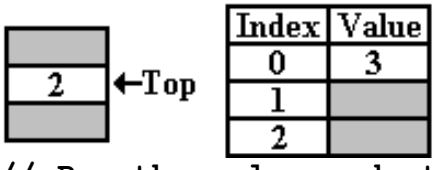
```
//   int a = 3 ;
     iconst_3          // Push the constant value 3 onto the stack
```



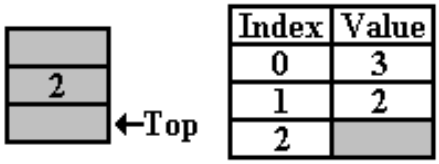

```
     istore_0          // Pop the value and store in location 0.
```



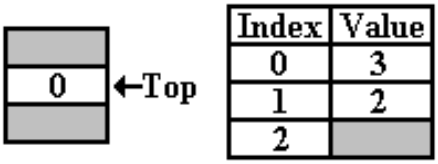

```
//   int b = 2 ;
     iconst_2          // Push the constant value 2 onto the stack
```



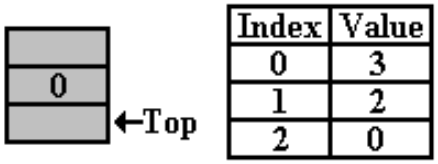

```
     istore_1          // Pop the value and store in location 1
```



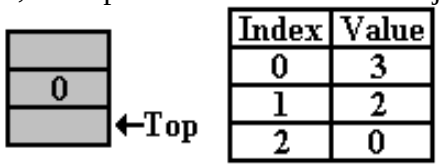

```
//   int sum = 0 ;
     iconst_0          // Push the constant value 0 onto the stack
```



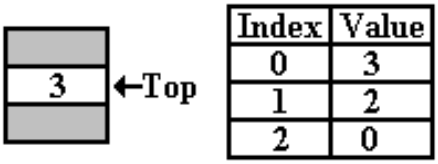

```
     istore_2          // Pop the value and store in location 2
```



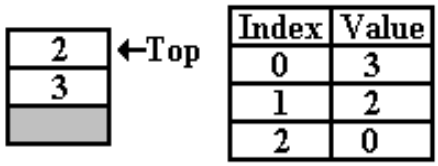
To facilitate reading the example, we repeat the state of the JVM just before the addition.



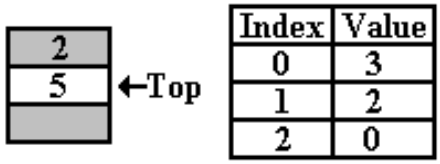

```
//    sum = a + b ;
      iload_0           // Push value from location 0 onto stack
```



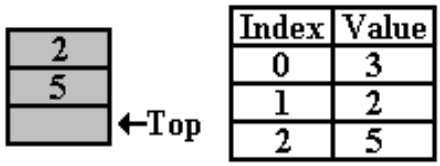

```
      iload_1           // Push value from location 1 onto stack
```




```
      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.
```



To complete the discussion of the example, we need to list the integer (32 bit two's complement) arithmetic operations. Here is a table of the standard operations.

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.

Here is the Java bytecode for the short sequence used in the example above. It has ten bytes.

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

The Conditional Instructions

Conditional instructions are one of the class of instructions that separate a stored program computer from a simple calculator. It must be possible to compare two values of the same type and make branching decisions based on the result. The facility to compare two values of different types (integer and float, float and double) will probably be provided by the compiler of a higher level language, and involve type conversions.

The JVM divides these operations into two classes: conditional branch instructions and comparison instructions. The division between the two classes is based on operand type, with the operators called "comparison instructions" used to compare long integers (64 bit), single precision floating point values, and double precision floating point values.

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.

Byte	Type	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

For each of these instructions, the branch target address is computed as (***pc*** + ***branch_offset***), where ***pc*** (program counter) 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.

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 an object 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

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

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

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.

The JVM includes a number of other useful bytecode instructions, including the following.

1. Instructions to push other integer constant values onto the stack.
2. Instructions to handle other stack operations, such as discarding the top elements, and swapping the top element with the one just below it.
3. A complete suite of logical operations.
4. A complete suite of type conversion operations.
5. Instructions for unconditional branches, subroutine calls, and subroutine returns.

The purpose of this sub–chapter has been to use the JVM (Java Virtual Machine) to illustrate a commonly used stack–oriented architecture. A full discussion of the JVM must include very much more.