



SIAM Tutorial: Java in High-Performance Computing: Quick Introduction to Java

George K. Thiruvathukal
Loyola University, Department of Computer Science
Northwestern University, Electrical and Computer Engineering Department

A Glimpse of Java, Java Grande, etc.

This is a tutorial on the Java language. We assume that the audience has limited experience with Java but does have experience with at least one other programming language.

Here we will answer:

- What is Java?
- Why use Java?
- JavaGrande in a Nutshell
- The JVM and Its Design
- Objects (PowerPoint)
- Containment (PowerPoint)
- Classes (PowerPoint)

Exhibit 4-1.
What is Java?

Java is a *programming language* and a *platform*.

Considering it as a language:

- a programming language with a syntax resembling C and C++
- higher-level (low-level would be machine language, highest-level would be scripting languages such as Python and Perl)
- purely object-oriented
- with extensive “standard” libraries

Considering it as a platform:

- 100% Pure Java - Applications may only be called 100% Pure Java if they are written exclusively in Java (or do not require execution outside the JVM).
- Ultimate goal is to support every relevant enterprise technology, including databases, directories, etc.
- Available in several tailored editions from handheld, to real-time, to personal, to enterprise.

Exhibit 4-2.
Brief History

Java is no longer a “new kid on the block” so to speak:

- over five years old and aging!
- created at Sun; originally called Oak
- estimated over 500,000 Java programmers world-wide. (total U.S. professional computer scientists, programmers, etc.: 1,400,000)

Strong support from companies

- IBM; >2500 Java programmers, round-the-world/round-the-clock
- Microsoft was an early adopter. They’re still somewhat supporting it as compliance with Sun’s “standard” is increasing (slowly) in their JVM implementation. (Their ultimate destiny, however, is .NET and C#, their new C-derived programming language.)

Exhibit 4-3.*Why is Java Popular?*

Java's popularity is attributable these early factors:

- Hyped by advertising blitz
- up to 10-fold increase in productivity (5-7-fold more common)
 - Vs. C? C is assembly language with a nicer syntax.
 - Vs. C++? C++ is terribly complex. It takes 18 months for professional programmers to become proficient in it.

Is it really true? Consider this (worst case) example from Cay Horstman's book on OOP in C++ & Java, "The March of Progress" to emit a floating point result to *stdout*:

- C

```
printf("%10.2f", x);
```

- C++

```
cout << setw(10) << setprecision(2) << showpoint << x;
```

- Java

```
java.text.NumberFormat formatter  
    = java.text.NumberFormat.getNumberInstance();  
formatter.setMinimumFractionDigits(2);  
formatter.setMaximumFractionDigits(2);
```

Why is Java Popular?

```
String s = formatter.format(x);  
for (int i = s.length(); i < 10; i++)  
    System.out.print(' ');  
System.out.print(s);
```

Marketing truly can make you believe *just about anything*.

The above example demonstrates that getting results in Java does not always simplify matters. In fact, in this particular case, the code bloat is significant (and very inefficient).

However, there are some important advantages to this code that are not obvious at first glance. More will be said about this later.

Exhibit 4-4.
Why use Java?

There are several good things about Java:

- garbage collection and the pure object discipline
- cross-platform portability
- testability
- simple multithreading (esp. synchronization)
- robustness and security
- on-the-fly upgradability (via dynamic class loading)

Exhibit 4-5.
Why not to use Java?

There are several reasons to consider not using Java, although these reasons may not apply to those coming from FORTRAN, C, or C++:

- The language is not easy to learn for those who do not know C++ or another object oriented language well.
- Although based on C++, some significant omissions (e.g. operator overloading and templates) result in excruciating coding experiences.
- Not a good language for writing scripts. Despite being a high-level language, higher-level languages such as Perl, Icon, Python, JavaScript, and Ruby are much more appropriate choices for scripting tasks (the so-called *low-performance computing* world).
- Not able to take advantage of the platform-specific capabilities easily. Java Native Interface allows native code to run but at a major penalty. Memory management between JVM and external libraries is a significant challenge.
- Numerics performance considered harmful. You will hear more about this.

Exhibit 4-6.*Java Grande in a Nutshell*

The Java Grande Forum Charter Principles:

- Founded by Geoffrey C. Fox on March 1, 1998, at the third Java Grande Conference (Stanford University)
- Some Goals:
 - To make Java the premiere choice for computational science applications above others (including C, C++, and FORTRAN).
 - To develop community consensus and recommendations for either changes to Java or establishment of standards (frameworks) for Grande libraries and services.
 - The Java Grande Forum does not intend to be a standards body for the Java™ language per se. Rather, JGF intends to act in an advisory capacity to ensure those working on Grande applications have a unified voice to address Java language design and implementation issues and communicate this input directly to Sun or a prospective Java standards group.
- Membership in the form is open to any qualified member of academia, industry or government who is willing to play an active role.

Exhibit 4-7.

Java Grande Form Working Groups and Contacts

Working Groups

- **Numerics Working Group:** Led by Roldan Pozo and Ron Boisvert, both of NIST.
- **Concurrency and Applications (Benchmarks) Group:** Led by Dennis Gannon of Indiana University and Denis Caromel, I.N.R.I.A. Nice/Sophia Antipolis.
- **Message Passing Group:** Led by Vladimir Getov of Westminster University.
- **Benchmarking,** Led d by EPCC at Edinburgh University Scotland
- **Computing Portals** (now part of the Grid Forum). See **<http://www.gridform.org>**.

Key Contacts

- Geoffrey C. Fox, Academic Coordinator, Florida State University
- John Reynders and Sia Zadeh, Sun Microsystems

Exhibit 4-8.

*Traditional Programming
Languages vs. JVM
Approach*

Words about the JVM

JVM interprets Java bytecodes stack-based registers for local variables each method invocation has its own frame, which contains private copies of

- operand stack
- local variables
- max size limitations for each

Data in instructions, stack, registers are all 32 bits.

- long and double values take up two stack positions or two registers

Exhibit 4-9.
*JVM Code and 80x86:
Side by Side*

Compare (80x86)

```
mov AX, 10      ; store the integer 10 in register AX
```

with (JVM)

```
bipush 10      ; store the integer constant 10 onto the stack  
istore_1      ; store the integer on top of the stack in local  
variable 1
```

or compare (80x86)

```
mov AX, 5      ; put the number 5 in AX  
mov BX,10     ; put the number 10 in BX  
add           ; add the numbers, the result is left in AX
```

to (JVM)

```
bipush 5      ; push 5 onto the stack  
bipush 10     ; push 10 onto the stack  
iadd         ; add top two numbers on the stack, leave result on  
stack  
istore_1     ; pop the result off the stack and store in local  
variable 1
```

This is not a performance comparison, just an illustration of

how JVM code is written compared to how it is done on commodity CPUs such as Intel 80x86.

Below is a table of JVM Instructions:

TABLE 0-1. Java Virtual Machine Instructions

Category	Numberof instruc- tions	Examples
arithmetic operations	24	iadd, lsub, frem
logical operations	12	iand, lor, ishl
numeric conversions	15	int2short, f2l, d2I
pushing constants	20	bipush, sipush, ldc, iconst_0, fconst_0
stack manipulation	9	pop, pop2, dup, dup2
flow control instruc- tions	28	goto, ifne, ifge, if_null, jsr, ret
managing local vari- ables	52	astore, istore, aload, iload, aload_0
manipulating arrays	17	aastore, bastore, aaload, baload
creating objects and arrays	4	new, newarray, anewarray, multi- anewarray

TABLE 0-1. Java Virtual Machine Instructions

Category	Number of instructions	Examples
object manipulation	6	getfield, putfield, getstatic, putstatic
method call and return	10	invokevirtual, invokestatic, areturn
miscellaneous	5	throw, monitorenter, breakpoint, nop

The JVM is similar to many assembler languages in its appearance.

Some observations:

- Instructions are generally typed with few exceptions: int (i), long (l), float (f), double (d), byte (b), char (c), short (s), reference or array (a)
- Instruction set is somewhat of a departure from a RISC-like model. Many instructions perform operations on arrays and objects, which are not supported (typically) on most hardware. Key implication: Looking at an instruction gives you little indication of a bound on its execution time (taken in isolation).
- You might want to add them up for later discussion.

Exhibit 4-10.
JVM Basic Architecture

The following illustrates the JVM structure:

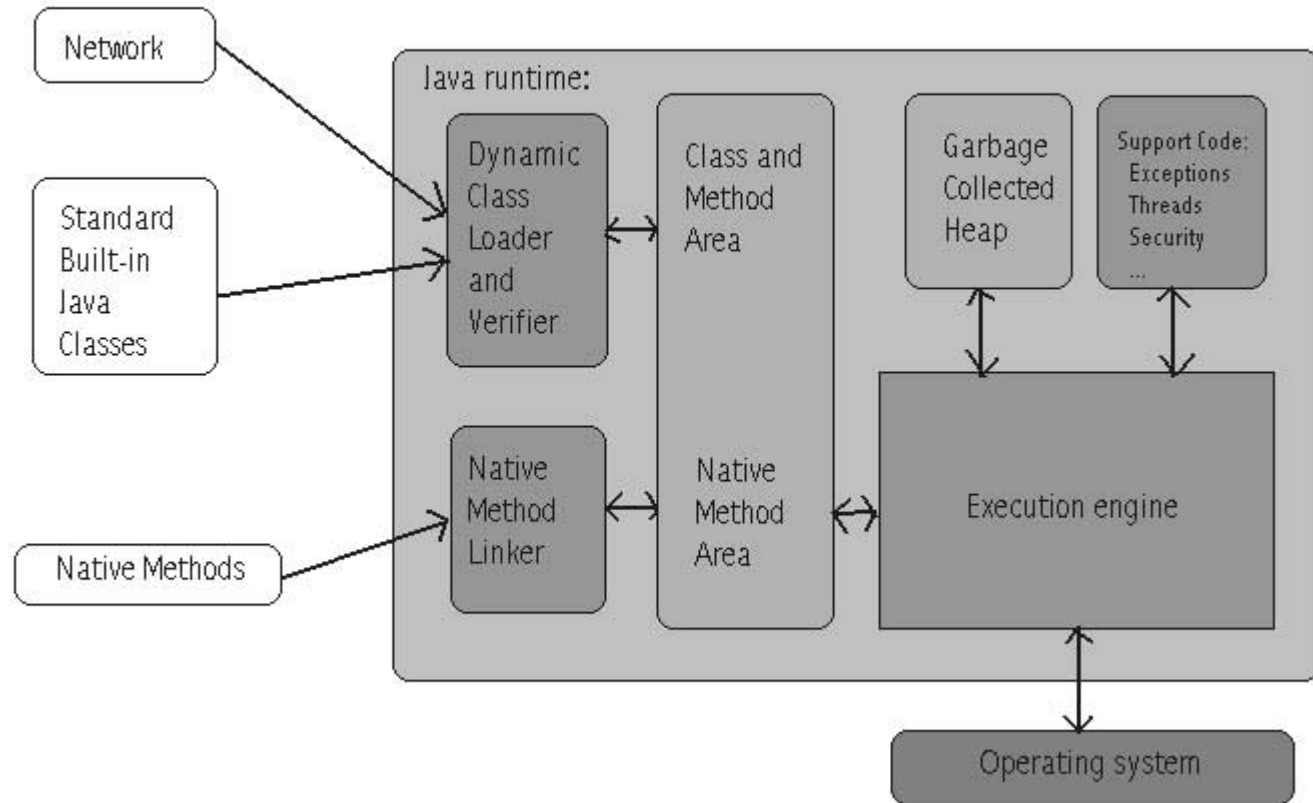


TABLE 0-2. Summary of JVM Components

Component	Description
Execution engine	runs the show
Memory manager	manages the heap (creation of objects, garbage collection)
Error and exception manager	Responsible for processing/propagating exceptions
Native method support	Allows you to exit the JVM to make calls to C and other language modules
Threads interface	Ability to access native platform concurrency support for lightweight processes.
Class loader	Similar to dynamic link libraries. Allows any Java class (code) to be loaded on demand.
Security manager	Interface to security policies. It is an access control framework.
Bytecode Verifier	Verifies all classes loaded (except trusted ones, such as the system class files) Verification involves making sure there are no improper bytecodes, each operation has an appropriate number of operands, etc. By verifying code up front, can remove some runtime checks, thus speeding up operation:

Exhibit 4-11.*Fundamental Issues/Problems with JVM Design*

The JVM has its share of fundamental problems that affect its scalability and performance.

- 8-bit instruction opcode, so only 256 instructions
 - results in non-orthogonal instruction set -- some data types (shorts, bytes, chars) are relegated to second-class status.
- Hard to extend
 - for example, if new datatypes (96 bit floating point, e.g.) come along, cannot add instructions
 - could use one of the unused instructions as an 'escape' code
- No parse tree
 - bytecode representation is flat -- e.g., no representation of lexical blocks in original code makes it harder for JIT compilers to optimize

Virtual machine overview

- Virtual machine
 - programs compiled into an instruction set for a virtual processor

- program executed on a real processor by a program which interprets the virtual machine instructions

Exhibit 4-12.
*Contrasting the JVM
 Approach with other
 approaches*

Traditional language implementation techniques range from interpreted to fully (and statically) compiled:

TABLE 0-3.

Approach	Detail
Fully Compiled	source text--> compiler --> executable (runs directly by CPU on the machine) Examples: C, C++, FORTRAN, COBOL, etc. Fastest Need compiler for each source language and machine.

 Contrasting the JVM Approach with other approaches

TABLE 0-3.

Approach	Detail
Interpreted	<p>source-text --> interpreter (directly executes)</p> <p>Examples: Tcl, JavaScript</p> <p>Can be 100X slower than compiled</p>
Semi-Compiled	<p>source-text --> compiler --> Intermediate code --> interpreter (executes)</p> <p>Examples: Java, VisualBasic, Smalltalk</p> <p>intermediate code portable to wide variety of machines (just need the interpreter)</p> <p>only need one compiler (if the compiler is written in the language itself, it is portable)</p> <p>Can be 10X or more slower than compiled</p> <p>Techniques such as Just-In-Time (JIT) compiling can produce performed close to compiled</p>

Exhibit 4-13.

*A Brief History of Byte
Cods and Virtual
Machines*

They're not all that new:

- IBM -- VM (1959)
- Smalltalk (1970s) -- could save a session on an HP running Unix and resume on a Mac
- P-Code (late 1970s) -- Pascal compiled to pseudo-code
- Emulation modes (e.g. PowerMac's 68000 emulation mode, or Windows on xxx)

Advantages

- Portability
- Safety -- interpreter provides a layer between program and underlying machine, can perform sanity and safety checks

Disadvantages

- Hard to build the VM interpreter
- Speed, speed, speed

Why is it going to work this time around?

A Brief History of Byte Cods and Virtual Machines

- Verifier
- Web
- C++
- Bill Gates. He is now building his own Java, C# and .NET

Exhibit 4-14.

Some words about the VM structure

Data Types

- Different levels of support for some types (as compared to Java)
- Word-oriented (32-bit), so doubles, longs need special treatment
- int, long, float, double have full support
- float, double require IEEE 754 -- need emulation if not native
- long, double require two stack entries or registers
- byte, short, char
 - converted to int upon retrieval storage
 - truncation ops (i2s, i2b, i2c)
- boolean -- 0 = false, 1 = true, also converted to int
- byte, short, char, boolean are *storage types* -- JVM manipulates data as ints only, but can store shorter values in fields and arrays
- reference = pointer.

Constant pool

- Every Java class and interface has a constant pool in its class file.
- Loaded by JVM and manipulated by it.
- Similar to symbol tables in shared libraries and executables.
- class constants are ordered, indexed by #, first constant has index 1.
- a reference to a constant is by its index #
 - references encoded in instructions as either 8-bit or 16-bit unsigned integers
 - thus 65535 is max # entries
- Used for all literal constants: numbers, quoted strings, methods, fields, classes.
- Ops: ldc, ldc_w, ldc2_w

Entry types

TABLE 0-4.

Entry type	Purpose	Contains	Used by
Class	identifies a java class	full name of class	new, instanceof, ...
Fieldref	identifies a Java field	class, name, and type signature	getfield, putfield
Methodref	identifies a Java method	class, name, and type signature	invokevirtual, invoke nonvirtual and invokestatic
InterfaceMethodref	identifies a Java interface method	ditto	invokeinterface
String	used for constant java.lang.String objects	Utf8 or Unicode string	ldc, ldc_w
Int	used for constant int value	int value	ldc, ldc_w
Float			ldc, ldc_w
Double			ldc2_w
Long			ldc2_w
NameAndType	identifies an identifier and type	name and type signature	Used by other constant pool entries
Utf8	gives bytes of strings in UTF8 format	byte array	Used by other constant pool entries

Stack frames

- Each invocation/activation of a method has frame allocated on the Java stack
- Each thread has its own Java stack
- Frame holds:
 - operand stack
 - local vars
 - program counter
 - pointer to class of current method
 - pointer to invoker's frame
- Execution engine keeps track of:
 - current method,
 - current class,
 - current constant pool

Registers

- No directly accessible general purpose registers

Some words about the VM structure

Operand stack

- 32 bit values
- per method activation
- has a size limit per method

Local vars

- 32 bit values
- per method activation
- has a size limit per method

Exhibit 4-15.
The Class Loader

Class loading

- Before the JVM can run a program, it has to load the classes that make up the program.
- Loading = the process of obtaining the bytes of a Java class file that defines a class.
 - read a file from disk
 - read a file over the network
 - read from a database
 - dynamically generate a byte array
- Two mechanisms for loading
 - system class loader
 - user defined class loaders
- Each runtime system has a "primordial" or "system" class loader -- a built-in class loader to load
 - all the standard Java classes (in java.* packages)
 - classes named on the command line to the Java executable.

The Class Loader

- System loader typically searches CLASSPATH.
- User can subclass `java.lang.ClassLoader` to define custom methods for locating and loading class files.
- `ClassLoader` resources are protected by the `SecurityManager`
 - the `SecMgr` in most Web browsers won't allow an applet to create a custom class loader.
- The JVM remembers how each class is loaded. When a class loaded by a `ClassLoader` tries to access another class, the same `ClassLoader` is invoked to obtain the new class.

Effectively, each `ClassLoader` defines its own namespace -- so classes that were loaded through one class loader are not necessarily available in others.

Exhibit 4-16.

Phases of a Class

- *Loading* - obtain the bytes that make up a class file
- *Linking* - convert those bytes to runtime form for JVM.
 - May cause recursive load/link of other classes (e.g. superclass)
 - verification happens now
- *Verification* -- Run the bytecode verifier
 - in Sun's JDK, only on classes loaded by a `ClassLoader`.
 - Classes loaded by internal system class loader assumed safe.
 - java flags:
 - `-verifyremote` -- only on classes loaded over the network
 - `-verify` -- verify all classes
 - `-noverify` -- verify no classes
- *Preparation* -- static fields given initial values, make sure no abstract methods on a concrete class, ...
- *Initialization* -- make sure all superclasses have been initialized, then run any static initializers (gathered in a method named `<clinit>`).

The Class Loader

- *Resolution* -- during execution, first time references go through a resolution process.
- *Finalization* -- before unloading, run the static void `classFinalize(){ }` method if it exists.

Constant pool resolution

- When JVM encounters a use of a constant pool entry for the first time (e.g., when you first use `new` to create a new object of a class, or in the first use of `getField` to get a field), the constant pool entry is resolved.
- Resolution =
 - check that the item you are trying to access exists (possibly loading or creating it if it doesn't already exist)
 - check permission to access (private, etc.)
- Classes:
 - When the JVM encounters a use of an unresolved entry tagged `CONSTANT_Class`, it checks if the named class has been loaded and setup already. If not, tries to load it (see above). If okay, then check privacy.
- Fieldrefs

The Class Loader

- A `CONSTANT_Fieldref` entry contains pointers to two other entries: a `CONSTANT_Class` entry (resolve first as above) indicating the class that the field is for, and a `CONSTANT_NameAndType` entry giving the name and type descriptor. Verify existence of the field in the class, verify type, verify access, verify storage type (static or not)
- `Methodref` and `InterfaceMethodref`
 - similar
- `Strings`
 - Construct an instance, maintain it in a hashtable: `String.intern()`
- Other types -- no special handling.

