Inside the Java Virtual Machine

Tim Lindholm and Frank Yellin

15.1 (Jan. 1997): p31.

Copyright: COPYRIGHT 1997 United Business Media LLC - UBM

<http://ubmtechnology.com/>

Abstract:

The Java Virtual Machine (JVM) functions as an interpreter to execute instructions encoded in Java bytecodes. After verifying their safety, JVM will step through the instructions in the file in a sequential manner. Sun Microsystems' implementation of the JVM verifies constraints at link time, thereby enhancing the interpreter's performance. The class file verifier runs four passes. The first pass ensures that a file is in the proper format for a Java class file. Additional verification is done on the second pass, after the class file is linked. The third pass performs a data flow analysis on each method, and the fourth is actually a virtual pass that executes according to specific JVM instructions. The JVM does, however, anticipate that most type checking has been done at compile time, and the JVM also contains explicit object support. Further, the JVM supports multiple threads of execution simultaneously. Each thread has its own program counter register.



Understanding the inner workings of the JVM will help you improve your Java application design.



The Java Virtual Machine (JVM) is the run-time module that executes the instructions encoded in Java bytecodes. As such, it serves as an interpreter of sorts whose design is conceptually simple.



All Java programs are collections of classes. These classes are compiled into bytecodes that are run by the JVM. When invoked, the JVM loads the bytecode flies and runs a verifier on them to ensure they are safe and properly formed; finally, the JVM steps through the file, executing instructions sequentially until the end of the program.



This article discusses the JVM's activities and aspects of the bytecodes themselves. To derive the most benefit from this discussion, you need some familiarity with the Java language and with the mechanics of program execution.

Verification Of class Files

Even though Java compilers attempt to produce only class files that satisfy all necessary static constraints, the Java Virtual Machine cannot verify that a file it is asked to load was generated by that compiler or is properly formed. Because of this, the JVM needs to verify for itself the hold the desired constraints have on the class files the JVM attempts to incorporate. A well-written JVM emulator could reject poorly formed instructions when a class file is loaded. Other constraints could be checked at run time. For example, a JVM implementation could tag run-time data and have each instruction ensure its operands are of the right type.

Instead, Sun's JVM implementation verifies that each class file it considers untrustworthy satisfies the necessary constraints at link time.

Link-time verification enhances the performance of the interpreter. Expensive checks that otherwise must be performed to verify constraints at run time for each interpreted instruction can be eliminated. The JVM can assume these checks already have been performed. For example, the JVM already will know the following:

\* There are no operand-stack overflows or underflows.

\* All local variable uses and stores are valid.

\* The arguments to all JVM instructions are of valid types.

Sun's class-file verifier is independent of any Java compiler. It should certify all code generated by Sun's current Java compiler; it also should certify code that other compilers can generate, as well as code that the current compiler could not possibly generate. Any class file that satisfies the structural criteria and static constraints will be certified by the verifier.

The class-file verifier also is independent of the Java language. Other languages can be compiled into the class format but will pass verification only if they satisfy the same constraints as a class file compiled from Java source.

The Verification Process

The class-file verifier operates in four passes.

Pass 1. When a prospective class file is loaded by the Java Virtual Machine, the JVM first ensures the file has the basic format of a Java class file. The first four bytes must contain the right magic number (OxCAFEBABE). All recognized attributes must be the proper length. The class file must not be truncated or have extra bytes at the end. The constant pool must not contain any superficially unrecognizable information.



While class-file verification properly occurs during class linking, this check for basic class-file integrity is necessary for any interpretation of the class-file contents, and can be considered a logical step in the verification process.

Pass 2. When the class file is linked, the verifier performs all additional verification that can be done without looking at the code attribute's code array. Cheeks performed by this pass include



\* ensuring that final classes are not subclassed and that final methods are not overridden

\* checking that every class (except Object) has a superclass

\* ensuring the constant pool satisfies the documented static constraints

\* checking that all field references and method references in the constant pool have valid names, valid classes, and valid type descriptors

Note that when the verifier looks at field and method references, it does not ensure the given field or method actually exists in the given class, nor does it cheek that the type descriptors given refer to real classes. It only checks that these items are wellformed. More detailed checking is delayed until passes 3 and 4.

Pass 3. During linking, the verifier cheeks the code attribute's code array for each method of the class file by performing data-flow analysis on each method. The verifier ensures the following at any given point in the program, no matter what code path is taken to reach that point:

\* The operand stack is always the same size and contains the same types of objects.

\* No local variable is accessed unless it is known to contain a value of an appropriate type.

\* Methods are invoked with the appropriate arguments.

\* Fields are assigned using only values of appropriate types.

\* All opcodes have appropriate type arguments on the operand stack and in the local variables.

Pass 4. For efficiency, certain tests that could in principle be performed in Pass 3 are delayed until the first time the code for the method is invoked. In doing so, Pass 3 of the verifier avoids loading class files unless it must.



For example, if a method invokes another method that returns an instance of class A, and that instance is assigned only to a field of the same type, the verifier does not bother to check whether class A actually exists. However, if it is assigned to a field of type B, the definitions of both A and B must be loaded to ensure A is a subclass of B.

Pass 4 is a virtual pass whose checking is done by the appropriate JVM instructions. The first time an instruction that references a type is executed, the executing instruction



\* loads the definition of the referenced type, if it has not already been loaded

\* ensures the currently executing type is allowed to reference the type

\* initializes the class, if this has not already been done

The first time an instruction invokes a method, or accesses or modifies a field, the executing instruction

\* ensures the referenced method or field exists in the given class

\* checks that the referenced method or field has the indicated descriptor

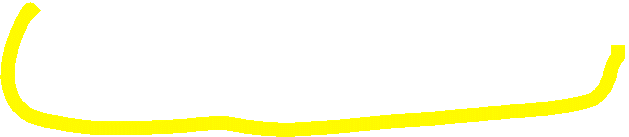
\* checks that the currently executing method has access to the referenced method or field

The Java Virtual Machine does not have to check the type of the object on the operand stack. That check is performed during Pass 3. Errors detected in Pass 4 cause instances of LinkageError to be thrown.

In Sun's JVM implementation, after verification has been performed, the instruction in the JVM code is replaced with an alternative form of the instruction. For example, the opcode new is replaced with new quick. This alternative instruction indicates that the verification needed by the instruction has occurred and does not need to be performed again. Thus, subsequent invocations of the method will be faster. It is illegal for these alternative instruction forms to appear in class files, and they should never be encountered by the verifier.

The Java Virtual Machine

To implement the Java Virtual Machine correctly after verification is complete, you need only be able to read the Java class-file format and correctly perform the operations specified therein.



Like the Java language, the JVM operates on two types: primitive types and reference types. There are, correspondingly, two values that can be stored in variables, passed as arguments, returned by methods, and operated upon: primitive values and reference values.

The Java Virtual Machine expects nearly all type checking to be done at compile time--not by the JVM itself. In particular, data need not be tagged or otherwise inspectable to determine types. Instead, the instruction set of the JVM distinguishes its operand types using instructions intended to operate on values of specific types. For instance, iadd, ladd, fadd, and dadd all are JVM instructions that add two numeric values, but they require operands whose types are int, long, float, and double, respectively. For a summary of type support in the Java Virtual Machine's instruction set, see Figure 1.



The JVM contains explicit support for objects. An object is either a dynamically allocated class instance or an array. A reference to an object is considered to have JVM type reference. Values of type reference can be thought of as pointers to objects. More than one reference to an object can exist. Although the Java Virtual Machine performs operations on objects, it never addresses them directly. Objects are always operated on, passed, and tested via values of type reference.

Primitive Types And Values

The primitive data types supported by the Java Virtual Machine are the numeric types and the returnAddress type. The numeric types consist of the integral types

\* byte, whose values are 8-bit, signed, two's-complement-integers

\* short, whose values are 16-bit, signed, two's-complement integers

\* int, whose values are 32-bit, signed, two's-complement integers

\* long, whose values are 64-bit, signed, two's-complement integers

\* char, whose values are 16-bit, unsigned integers representing Unicode version 1.1.5 characters and the floating-point types

\* float, whose values are 32-bit, IEEE 754, floating-point numbers

\* double, whose values are 64-bit, IEEE 754, floating-point numbers

The values of the returnAddress type are pointers to the opcodes of JVM instructions. Only the returnAddress type is not a Java language type.

Three kinds of reference types exist whose values are references to dynamically created class instances or arrays that implement interfaces: class types, interface types, and array types. A reference value also can be the special null reference, a reference to no object. The null reference initially has no run-time type but can be cast to any type.

[TABULAR DATA OMITTED]

This article is based on excerpts from The Java Virtual Machine Specification by Tim Lindholm and Frank Yellin (ISBN 0-201-63452-x). Copyright 1997 by Sun Microsystems Inc. Adapted with permission of Addison-Wesley Longman, 1 Jacob Way, Reading, MA 01867, (800) 822-6339. For more information on this book, access http://www.aw.com/cp/lindholmyellin.html.

**Source Citation**   (MLA 8th Edition)

Lindholm, Tim, and Frank Yellin. "Inside the Java Virtual Machine." *UNIX Review*, Jan. 1997, p. 31+. *Academic OneFile*, go.galegroup.com/ps/i.do?p=AONE&sw=w&u=uniwater&v=2.1&id=GALE%7CA18937758&it=r&asid=13401230a1c9542c1bbedbed67b91b97. Accessed 30 Oct. 2017.

**Gale Document Number:**GALE|A18937758