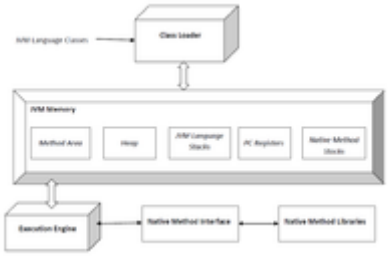
Java virtual machine

A **Java virtual machine** (**JVM**) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode. The JVM is detailed by a specification that formally describes what is required in a JVM implementation. Having a specification ensures interoperability of Java programs across different implementations so that program authors using the Java Development Kit (JDK) need not worry about idiosyncrasies of the underlying hardware platform.

The JVM reference implementation is developed by the OpenJDK project as open source code and includes a JIT compiler called HotSpot. The commercially supported Java releases available from Oracle Corporation are based on the OpenJDK runtime. Eclipse OpenJ9 is another open source JVM for OpenJDK.

Java virtual machine

Designer	Sun Microsystems
Bits	32-bit
Introduced	1994
Version	15.0.2[1]
Type	Stack and register–register
Encoding	Variable
Branching	Compare and branch
Endianness	Big
Open	Yes
Registers	
General purpose	Per-method operand stack (up to 65535 operands) plus per-method local variables (up to 65535)



Overview of a Java virtual machine (JVM) architecture based on The Java Virtual Machine Specification Java SE 7 Edition

Contents

JVM specification

The Java virtual machine is an abstract (virtual) computer defined by a specification. The garbage-collection algorithm used and any internal optimization of the Java virtual machine instructions (their translation into machine code) are not specified. The main reason for this omission is to not unnecessarily constrain

implementers. Any Java application can be run only inside some concrete implementation of the abstract specification of the Java virtual machine.[2]

Starting with Java Platform, Standard Edition (J2SE) 5.0, changes to the JVM specification have been developed under the Java Community Process as JSR 924.[3] As of 2006, changes to specification to support changes proposed to the class file format (JSR 202)[4] are being done as a maintenance release of JSR 924. The specification for the JVM was published as the *blue book*,[5] The preface states:

> We intend that this specification should sufficiently document the Java Virtual Machine to make possible compatible clean-room implementations. Oracle provides tests that verify the proper operation of implementations of the Java Virtual Machine.

One of Oracle's JVMs is named HotSpot, the other, inherited from BEA Systems is JRockit. Clean-room Java implementations include Kaffe, OpenJ9 and Skelmir's CEE-J (https://skelmir.com). Oracle owns the Java trademark and may allow its use to certify implementation suites as fully compatible with Oracle's specification.

Class loader

One of the organizational units of JVM byte code is a class. A class loader implementation must be able to recognize and load anything that conforms to the Java class file format. Any implementation is free to recognize other binary forms besides *class* files, but it must recognize *class* files.

The class loader performs three basic activities in this strict order:

1. Loading: finds and imports the binary data for a type
2. Linking: performs verification, preparation, and (optionally) resolution
   - Verification: ensures the correctness of the imported type
   - Preparation: allocates memory for class variables and initializing the memory to default values
   - Resolution: transforms symbolic references from the type into direct references.
3. Initialization: invokes Java code that initializes class variables to their proper starting values.

In general, there are two types of class loader: bootstrap class loader and user defined class loader.

Every Java virtual machine implementation must have a bootstrap class loader, capable of loading trusted classes, and an extension class loader or application class loader. The Java virtual machine specification doesn't specify how a class loader should locate classes.

Virtual machine architecture

The JVM operates on primitive values (integers and floating-point numbers) and references. The JVM is fundamentally a 32-bit machine. `long` and `double` types, which are 64-bits, are supported natively, but consume two units of storage in a frame's local variables or operand stack, since each unit is 32 bits. `boolean`, `byte`, `short`, and `char` types are all sign-extended (except `char` which is zero-extended) and operated on as 32-bit integers, the same as `int` types. The smaller types only have a few type-specific instructions for loading, storing, and type conversion. `boolean` is operated on as 8-bit `byte` values, with 0 representing `false` and 1 representing `true`. (Although `boolean` has been treated as a type since *The*

Java Virtual Machine Specification, Second Edition clarified this issue, in compiled and executed code there is little difference between a `boolean` and a `byte` except for name mangling in method signatures and the type of boolean arrays. `boolean`s in method signatures are mangled as `Z` while `byte`s are mangled as `B`. Boolean arrays carry the type `boolean[]` but use 8 bits per element, and the JVM has no built-in capability to pack booleans into a bit array, so except for the type they perform and behave the same as `byte` arrays. In all other uses, the `boolean` type is effectively unknown to the JVM as all instructions to operate on booleans are also used to operate on `byte`s.)

The JVM has a garbage-collected heap for storing objects and arrays. Code, constants, and other class data are stored in the "method area". The method area is logically part of the heap, but implementations may treat the method area separately from the heap, and for example might not garbage collect it. Each JVM thread also has its own call stack (called a "Java Virtual Machine stack" for clarity), which stores frames. A new frame is created each time a method is called, and the frame is destroyed when that method exits.

Each frame provides an "operand stack" and an array of "local variables". The operand stack is used for operands to computations and for receiving the return value of a called method, while local variables serve the same purpose as registers and are also used to pass method arguments. Thus, the JVM is both a stack machine and a register machine.

Bytecode instructions

The JVM has instructions for the following groups of tasks:

Load and store · Arithmetic · Type conversion · Object creation and manipulation · Operand stack management (push / pop) · Control transfer (branching) · Method invocation and return · Throwing exceptions · Monitor-based concurrency

The aim is binary compatibility. Each particular host operating system needs its own implementation of the JVM and runtime. These JVMs interpret the bytecode semantically the same way, but the actual implementation may be different. More complex than just emulating bytecode is compatibly and efficiently implementing the Java core API that must be mapped to each host operating system.

These instructions operate on a set of common abstracted data types rather the native data types of any specific instruction set architecture.

JVM languages

A JVM language is any language with functionality that can be expressed in terms of a valid class file which can be hosted by the Java Virtual Machine. A class file contains Java Virtual Machine instructions (Java byte code) and a symbol table, as well as other ancillary information. The class file format is the hardware- and operating system-independent binary format used to represent compiled classes and interfaces.[6]

There are several JVM languages, both old languages ported to JVM and completely new languages. JRuby and Jython are perhaps the most well-known ports of existing languages, i.e. Ruby and Python respectively. Of the new languages that have been created from scratch to compile to Java bytecode, Clojure, Apache Groovy, Scala and Kotlin may be the most popular ones. A notable feature with the JVM languages is that they are compatible with each other, so that, for example, Scala libraries can be used with Java programs and vice versa.[7]

Java 7 JVM implements *JSR 292: Supporting Dynamically Typed Languages*[8] on the Java Platform, a new feature which supports dynamically typed languages in the JVM. This feature is developed within the Da Vinci Machine project whose mission is to extend the JVM so that it supports languages other than Java.[9][10]

Bytecode verifier

A basic philosophy of Java is that it is inherently safe from the standpoint that no user program can crash the host machine or otherwise interfere inappropriately with other operations on the host machine, and that it is possible to protect certain methods and data structures belonging to trusted code from access or corruption by untrusted code executing within the same JVM. Furthermore, common programmer errors that often led to data corruption or unpredictable behavior such as accessing off the end of an array or using an uninitialized pointer are not allowed to occur. Several features of Java combine to provide this safety, including the class model, the garbage-collected heap, and the verifier.

The JVM verifies all bytecode before it is executed. This verification consists primarily of three types of checks:

- Branches are always to valid locations
- Data is always initialized and references are always type-safe
- Access to private or package private data and methods is rigidly controlled

The first two of these checks take place primarily during the verification step that occurs when a class is loaded and made eligible for use. The third is primarily performed dynamically, when data items or methods of a class are first accessed by another class.

The verifier permits only some bytecode sequences in valid programs, e.g. a jump (branch) instruction can only target an instruction within the same method. Furthermore, the verifier ensures that any given instruction operates on a fixed stack location,[11] allowing the JIT compiler to transform stack accesses into fixed register accesses. Because of this, that the JVM is a stack architecture does not imply a speed penalty for emulation on register-based architectures when using a JIT compiler. In the face of the code-verified JVM architecture, it makes no difference to a JIT compiler whether it gets named imaginary registers or imaginary stack positions that must be allocated to the target architecture's registers. In fact, code verification makes the JVM different from a classic stack architecture, of which efficient emulation with a JIT compiler is more complicated and typically carried out by a slower interpreter.

The original specification for the bytecode verifier used natural language that was incomplete or incorrect in some respects. A number of attempts have been made to specify the JVM as a formal system. By doing this, the security of current JVM implementations can more thoroughly be analyzed, and potential security exploits prevented. It will also be possible to optimize the JVM by skipping unnecessary safety checks, if the application being run is proven to be safe.[12]

Secure execution of remote code

A virtual machine architecture allows very fine-grained control over the actions that code within the machine is permitted to take. It assumes the code is "semantically" correct, that is, it successfully passed the (formal) bytecode verifier process, materialized by a tool, possibly off-board the virtual machine. This is designed to allow safe execution of untrusted code from remote sources, a model used by Java applets, and other secure code downloads. Once bytecode-verified, the downloaded code runs in a restricted "sandbox", which is designed to protect the user from misbehaving or malicious code. As an addition to the bytecode verification

process, publishers can purchase a certificate with which to digitally sign applets as safe, giving them permission to ask the user to break out of the sandbox and access the local file system, clipboard, execute external pieces of software, or network.

Formal proof of bytecode verifiers have been done by the Javacard industry (Formal Development of an Embedded Verifier for Java Card Byte Code[13])

Bytecode interpreter and just-in-time compiler

For each hardware architecture a different Java bytecode interpreter is needed. When a computer has a Java bytecode interpreter, it can run any Java bytecode program, and the same program can be run on any computer that has such an interpreter.

When Java bytecode is executed by an interpreter, the execution will always be slower than the execution of the same program compiled into native machine language. This problem is mitigated by just-in-time (JIT) compilers for executing Java bytecode. A JIT compiler may translate Java bytecode into native machine language while executing the program. The translated parts of the program can then be executed much more quickly than they could be interpreted. This technique gets applied to those parts of a program frequently executed. This way a JIT compiler can significantly speed up the overall execution time.

There is no necessary connection between the Java programming language and Java bytecode. A program written in Java can be compiled directly into the machine language of a real computer and programs written in other languages than Java can be compiled into Java bytecode.

Java bytecode is intended to be platform-independent and secure.[14] Some JVM implementations do not include an interpreter, but consist only of a just-in-time compiler.[15]

JVM in the web browser

At the start of the Java platform's lifetime, the JVM was marketed as a web technology for creating Rich Web Applications. As of 2018, most web browsers and operating systems bundling web browsers do not ship with a Java plug-in, nor do they permit side-loading any non-Flash plug-in. The Java browser plugin was deprecated in JDK 9.[16]

The NPAPI Java browser plug-in was designed to allow the JVM to execute so-called Java applets embedded into HTML pages. For browsers with the plug-in installed, the applet is allowed to draw into a rectangular region on the page assigned to it. Because the plug-in includes a JVM, Java applets are not restricted to the Java programming language; any language targeting the JVM may run in the plug-in. A restricted set of APIs allow applets access to the user's microphone or 3D acceleration, although applets are not able to modify the page outside its rectangular region. Adobe Flash Player, the main competing technology, works in the same way in this respect.

As of June 2015 according to W3Techs, Java applet and Silverlight use had fallen to 0.1% each for all web sites, while Flash had fallen to 10.8%.[17]

JavaScript JVMs and interpreters

As of May 2016, JavaPoly allows users to import unmodified Java libraries, and invoke them directly from JavaScript. JavaPoly allows websites to use unmodified Java libraries, even if the user does not have Java installed on their computer.[18]

Compilation to JavaScript

With the continuing improvements in JavaScript execution speed, combined with the increased use of mobile devices whose web browsers do not implement support for plugins, there are efforts to target those users through compilation to JavaScript. It is possible to either compile the source code or JVM bytecode to JavaScript.

Compiling the JVM bytecode, which is universal across JVM languages, allows building upon the language's existing compiler to bytecode. The main JVM bytecode to JavaScript compilers are TeaVM,[19] the compiler contained in Dragome Web SDK,[20] Bck2Brwsr,[21] and j2js-compiler.[22]

Leading compilers from JVM languages to JavaScript include the Java-to-JavaScript compiler contained in Google Web Toolkit, Clojurescript (Clojure), GrooScript (Apache Groovy), Scala.js (Scala) and others.[23]

Java Runtime Environment

The Java Runtime Environment (JRE) released by Oracle is a freely available software distribution containing a stand-alone JVM (HotSpot), the Java standard library (Java Class Library), a configuration tool, and—until its discontinuation in JDK 9—a browser plug-in. It is the most common Java environment installed on personal computers in the laptop and desktop form factor. Mobile phones including feature phones and early smartphones that ship with a JVM are most likely to include a JVM meant to run applications targeting Micro Edition of the Java platform. Meanwhile, most modern smartphones, tablet computers, and other handheld PCs that run Java apps are most likely to do so through support of the Android operating system, which includes an open source virtual machine incompatible with the JVM specification. (Instead, Google's Android development tools take Java programs as input and output Dalvik bytecode, which is the native input format for the virtual machine on Android devices.)

Performance

The JVM specification gives a lot of leeway to implementors regarding the implementation details. Since Java 1.3, JRE from Oracle contains a JVM called HotSpot. It has been designed to be a high-performance JVM.

To speed-up code execution, HotSpot relies on just-in-time compilation. To speed-up object allocation and garbage collection, HotSpot uses generational heap.

Generational heap

The *Java virtual machine heap* is the area of memory used by the JVM for dynamic memory allocation.[24]

In HotSpot the heap is divided into *generations*:

- The *young generation* stores short-lived objects that are created and immediately garbage collected.
- Objects that persist longer are moved to the *old generation* (also called the *tenured generation*). This memory is subdivided into (two) Survivors spaces where the objects that survived the first and next garbage collections are stored.

The *permanent generation* (or *permgen*) was used for class definitions and associated metadata prior to Java 8. Permanent generation was not part of the heap.[25][26] The *permanent generation* was removed from Java 8.[27]

Originally there was no permanent generation, and objects and classes were stored together in the same area. But as class unloading occurs much more rarely than objects are collected, moving class structures to a specific area allowed significant performance improvements.[25]

Security

Oracle's JRE is installed on a large number of computers. End users with an out-of-date version of JRE therefore are vulnerable to many known attacks. This led to the widely shared belief that Java is inherently insecure.[28] Since Java 1.7, Oracle's JRE for Windows includes automatic update functionality.

Before the discontinuation of the Java browser plug-in, any web page might have potentially run a Java applet, which provided an easily accessible attack surface to malicious web sites. In 2013 Kaspersky Labs reported that the Java plug-in was the method of choice for computer criminals. Java exploits are included in many exploit packs that hackers deploy onto hacked web sites.[29] Java applets were removed in Java 11, released on September 25, 2018.

See also

- List of Java virtual machines
- Comparison of Java virtual machines
- Comparison of application virtual machines
- Automated exception handling
- Java performance
- List of JVM languages
- Java processor
- Common Language Runtime

References

1. "jdk-updates/jdk15u: tags" (https://hg.openjdk.java.net/jdk-updates/jdk15u/tags). *Oracle Corporation*. Retrieved 2021-01-27.
2. Bill Venners, *Inside the Java Virtual Machine (http://www.artima.com/insidejvm/ed2/index.html)* Chapter 5
3. "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 924" (http://www.jcp.org/en/jsr/detail?id=924). Jcp.org. Retrieved 2015-06-26.
4. "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 202" (http://www.jcp.org/en/jsr/detail?id=202). Jcp.org. Retrieved 2015-06-26.
5. *The Java Virtual Machine Specification (http://java.sun.com/docs/books/vmspec/)* (the first (http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html) and second (http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html) editions are also available online).
6. "The Java Virtual Machine Specification : Java SE 7 Edition" (http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf) (PDF). Docs.oracle.com. Retrieved 2015-06-26.
7. "Frequently Asked Questions - Java Interoperability" (http://www.scala-lang.org/old/faq/4). *scala-lang.org*. Retrieved 2015-11-18.
8. "The Java Community Process(SM) Program - JSRs: Java Specification Requests - detail JSR# 292" (https://jcp.org/en/jsr/detail?id=292). Jcp.org. Retrieved 2015-06-26.
9. "Da Vinci Machine project" (http://openjdk.java.net/projects/mlvm/). Openjdk.java.net. Retrieved 2015-06-26.

10. "New JDK 7 Feature: Support for Dynamically Typed Languages in the Java Virtual Machine" (http://www.oracle.com/technetwork/articles/javase/dyntypelang-142348.html). Oracle.com. Retrieved 2015-06-26.
11. "The Verification process" (http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html#9766). *The Java Virtual Machine Specification*. Sun Microsystems. 1999. Retrieved 2009-05-31.
12. Freund, Stephen N.; Mitchell, John C. (1999). "A formal framework for the Java bytecode language and verifier". *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '99*. pp. 147–166. CiteSeerX 10.1.1.2.4663 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.4663). doi:10.1145/320384.320397 (https://doi.org/10.1145%2F320384.320397). ISBN 978-1581132380.
13. http://www-sop.inria.fr/everest/Lilian.Burdy/CBR02dsn.pdf
14. David J. Eck, *Introduction to Programming Using Java (http://math.hws.edu/javanotes/c1/s3.html)*, Seventh Edition, Version 7.0, August 2014 at Section 1.3 "The Java Virtual Machine"
15. *Oracle JRockit Introduction (http://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm) Archived (https://web.archive.org/web/20150906145705/http://docs.oracle.com/cd/E15289_01/doc.40/e15058/underst_jit.htm) 2015-09-06 at the Wayback Machine* Release R28 at 2. "Understanding Just-In-Time Compilation and Optimization"
16. "Oracle deprecates the Java browser plugin, prepares for its demise" (https://arstechnica.com/information-technology/2016/01/oracle-deprecates-the-java-browser-plugin-prepares-for-its-demise/). *Ars Technica*. 28 January 2016. Retrieved 15 April 2016.
17. "Historical yearly trends in the usage of client-side programming languages, June 2015" (http://w3techs.com/technologies/history_overview/client_side_language/all/y). W3techs.com. Retrieved 2015-06-26.
18. Krill, Paul (13 May 2016). "JavaPoly.js imports existing Java code and invokes it directly from JavaScript" (http://www.infoworld.com/article/3069995/java/new-javascript-library-brings-java-to-browsers-without-applets.html). *InfoWorld*. Retrieved 18 July 2016.
19. "TeaVM project home page" (http://teavm.org/). Teavm.org. Retrieved 2015-06-26.
20. "Dragome Web SDK" (http://www.dragome.com/). Dragome.com. Retrieved 2015-06-26.
21. "Bck2Brwsr - APIDesign" (http://wiki.apidesign.org/wiki/Bck2Brwsr). Wiki.apidesign.org. Retrieved 2015-06-26.
22. Wolfgang Kuehn (decatur). j2js-compiler (https://github.com/decatur/j2js-compiler) GitHub
23. "List of languages that compile to JS · jashkenas/coffeescript Wiki · GitHub" (https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js). Github.com. 2015-06-19. Retrieved 2015-06-26.
24. "Frequently Asked Questions about Garbage Collection in the Hotspot Java Virtual Machine" (http://java.sun.com/docs/hotspot/gc1.4.2/faq.html). Sun Microsystems. 6 February 2003. Retrieved 7 February 2009.
25. Masamitsu, Jon (28 November 2006). "Presenting the Permanent Generation" (https://blogs.oracle.com/jonthecollector/entry/presenting_the_permanent_generation). Retrieved 7 February 2009.
26. Nutter, Charles (11 September 2008). "A First Taste of InvokeDynamic" (http://blog.headius.com/2008/09/first-taste-of-invokedynamic.html). Retrieved 7 February 2009.
27. "JEP 122: Remove the Permanent Generation" (http://openjdk.java.net/jeps/122). Oracle Corporation. 2012-12-04. Retrieved 2014-03-23.
28. "What Is Java, Is It Insecure, and Should I Use It?" (https://lifehacker.com/5988800/what-is-java-is-it-insecure-and-should-i-use-it). Lifehacker.com. 2013-01-14. Retrieved 2015-06-26.

29. "Is there any protection against Java exploits? | Kaspersky Lab" (https://web.archive.org/web/20150404000409/https://www.kaspersky.com/about/news/virus/2013/is_there_any_protection_against_java_exploits). Kaspersky.com. 2013-09-09. Archived from the original (https://www.kaspersky.com/about/news/virus/2013/is_there_any_protection_against_java_exploits) on 2015-04-04. Retrieved 2015-06-26.

- *Clarifications and Amendments to the Java Virtual Machine Specification, Second Edition (http://java.sun.com/docs/books/vmspec/2nd-edition/jvms-clarify.html)* includes list of changes to be made to support J2SE 5.0 and JSR 45
- JSR 45 (http://www.jcp.org/en/jsr/detail?id=45), specifies changes to the class file format to support source-level debugging of languages such as JavaServer Pages (JSP) and SQLJ that are translated to Java

External links

- The Java Virtual Machine Specification (http://docs.oracle.com/javase/specs/jvms/se7/html/index.html)
- How to download and install prebuilt OpenJDK packages (http://openjdk.java.net/install/)
- How to Install Java? (JRE from Oracle) (https://java.com/en/download/help/download_options.xml)
