Published on **ONJava.com** (http://www.onjava.com/)
See this if you're having trouble printing code examples

What Is Java

by Chris Adamson
03/08/2006

Java
: "Java" generally refers to a combination of three things: the Java programming language (a high-level, object-oriented programming language); the Java Virtual Machine (a high-performance virtual machine that executes bytecodes on a specific computing platform, typically abbreviated *JVM*); and the Java platform, a JVM running compiled Java bytecodes, usually calling on a set of standard libraries such as those provided by Java Standard Edition (SE) or Enterprise Edition (EE). Though coupled by design, the language does not imply the JVM, and vice versa.

In this Article

We recently received an email asking for a "What Is Java" entry on the O'Reilly Network What Is site. Who could possibly not know what Java is in 2006? After ten years of books, websites, and conferences, doesn't everyone know what Java is? Apparently not.

After all, things have *changed*.

For every dusty definition that speaks of applets and Just-In-Time compilers, there are new directions and new realities that have settled in, understood by many, yet not always completely documented. Java used to mean:

- Applets
- Bytecode interpretation
- Slow performance
- A "cargo cult" awaiting drops from Sun

Today, it means:

- Web applications, web services, SOAs, etc.
- Hotspot dynamic compilation
- High-performance
- An open source community, increasingly independent of Sun

The old slogan "Write Once, Run Anywhere" still holds true--but what's being written and where and how it's being run are changing.

The Java Programming Language

Java, the language, is a high-level object-oriented programming language, influenced in various ways by C, C++, and Smalltalk, with ideas borrowed from other languages as well (see O'Reilly's History of Programming Languages). Its syntax was designed to be familiar to those familiar with C-descended "curly brace" languages, but with arguably stronger OO principles than those found in C++, static typing of objects, and a fairly rigid system of exceptions that require every method in the call stack to either handle exceptions or declare their ability to throw them. Garbage collection is assumed, sparing the developer from having to free memory used by obsolete objects.

One of Java's more controversial aspects--widely accepted at the time of its release but increasingly criticized today--is its incomplete object-orientation. Specifically, Java *primitives* such as `int`, `char`, `boolean`, etc. are not objects, and require a completely different treatment from the developer: as `int` is not a class, you cannot subclass and declare new methods on it, cannot pass it to a method that expects a generic `Object`, and so on. The inclusion of primitives increases Java performance, but at the arguable expense of code clarity, as anyone who's had to work with the so-called "wrapper classes" (`Integer`, `Character`, and `Boolean`) will attest. Java 5.0 introduces an "autoboxing" scheme to eliminate many uses of the wrapper classes, but in some ways it obscures what is really going on.

Philosophically, Java is a "fail early" language. Because of its syntactic restrictions, many programming failures are simply not possible in Java. With no direct access to pointers, pointer-arithmetic errors are non-existent. Using an object as a different type than what it was originally declared to be requires an explicit cast, which gives the compiler an opportunity to reject illogical programming, like calling a `String` method on an `Image`.

Many Java enterprise frameworks require the use of configuration files or deployment descriptors, typically written in XML, to specify functionality: what class handles a certain HTTP request, the order of steps to execute in a rule engine, etc. In effect, they have to go beyond the language to implement their functionality. Critics point out that this has the perverse effect of not only escaping Java's compiler checks, but also that a developer can no longer determine how a program will operate just by looking at its source code. Java 5.0 adds *annotations* to the language, which allows the tagging of methods, fields, and classes with values that can then be inspected and operated on at runtime, usually through reflection. Many programmers like annotations because they simplify tasks that might otherwise be addressed by deployment descriptors or other means. But again, they can make it difficult to understand Java code, as the presence or absence of an annotation may affect how the code is executed, in ways that are in no way obvious from the annotation.

Despite these criticisms, Java is generally understood to be the most popular general-purpose computing language in use today. It is a widely used standard in enterprise programming, and in 2005, it replaced C++ as the language most used by projects on SourceForge. What it has going for it is immense: free tools (on multiple platforms: Linux, Windows, Solaris, and Mac can all compile and execute Java apps), a vast base of knowledge, and a large pool of readily available developers.

The Java language hits a specific point in the tradeoff between developer productivity and code performance: CPU cycles keep getting cheaper, developers largely don't, so it is perhaps inevitable to accept another layer of abstraction between the developer and the execution of CPU opcodes, if it allows the developer to create better software faster. In fact, critics of Java's productivity, such as Bruce Tate in Beyond Java, may simply be observing this trend continuing past Java to a new sweet spot that further trades performance for developer productivity.

The Java Platforms

Java is generally thought of in terms of three *platforms*: Standard Edition (SE), Enterprise Edition (EE), and Micro Edition (ME). Each describes the combination of a language version, a set of standard libraries, and a virtual machine (see below) to execute the code. EE is a superset of SE--any EE application can assume the existence of all of the SE libraries--and EE's use of the language is identical to SE's.

Because of the limitations of small devices like phones and set-top boxes, Java Micro Edition differs significantly from its siblings. It is not a subset of SE (as SE is of EE), as some of its libraries exist only in Micro Edition. Moreover, ME eliminates some language features, such as the `float` primitive and `Float` class, reflecting the computing limitations of the platforms it runs on. Requiring different tools than SE and EE, and with profound differences in devices that makes code portability far less realistic in the micro space, many Java developers see ME as utterly alien.

The Java Virtual Machine

At some point, Java source needs to become platform-native executable code. This typically requires a two-step process: the developer compiles his or her source into Java bytecode, and then a Java Virtual Machine (JVM) converts this into native code for the host platform. This latter step originally was performed by interpretation--taking each JVM instruction and converting it on the fly to one or more native instructions. Later, just-in-time (JIT) compilers converted all of a Java program from JVM bytecode to native code as the program started up. In the modern era, there are multiple approaches. Sun's HotSpot compiler starts by interpreting code and profiling it at runtime, compiling and optimizing those parts that are found to be most critical to the program's operation. The "mixed mode interpreter" of IBM's JVMs works much the same way. These approaches avoid the startup performance hit entailed by JITing the entire program, but means that performance arrives over time, as critical code sections are located and optimized. Long-running server processes are well-served by this approach, client applications less so.

As is the case with primitives, the two-step compile cycle of Java now starts to look like a premature optimization to some critics. If you're going to wait until runtime to compile from Java bytecode to native, they ask, why not save the developer a step by interpreting not Java bytecode, but Java *source*? As Tate notes in *Beyond Java*, "Java is not the simplest of languages. Nor is it friendly to very short iterations ... Other languages let you move from one change to the next without a cumbersome compile/deploy cycle."

The JVM Without Java

Indeed, one of Tate's key criteria in finding potential successors to Java's success is the idea that "the next commercially successful language should have a version that runs in the JVM. That would help a language overcome many obstacles, both political and technical." He points out that a VM approach gives you security ("if you can secure the virtual machine, it's much easier to secure the language"), portability, interoperability, and extensibility. With the JVM having effectively solved these problems, a new language wouldn't need its own VM if it can simply run in the JVM that is already on millions of computers.

In many ways, this is already happening. Writing interpreters for scripting languages in Java effectively brings these languages to the JVM, like Rhino for JavaScript, Jython for Python, or JRuby for Ruby.

But it's also possible to bypass the Java language altogether and go straight to the JVM level. There are already C-to-JVM bytecode compilers, such as the commercial Axiomatic Multi-Platform C, which provides a subset of ANSI C. Furthermore, the growth of Java bytecode manipulation with tools such as ASM and Apache BCEL allow Java applications to create executable classes at runtime. This is no longer Java, but effectively a form of assembly language programming for the JVM.

Perhaps appreciating the desire to run non-Java code on the JVM, a new JSR, "Supporting Dynamically Typed Languages on the JavaTM Platform" (JSR 292), has recently been introduced, specifying a new bytecode that would make the JVM better suited to running languages without static type information.

Java Without the JVM

You can also turn the tables the other way, and run Java without a JVM. After all, at some point, Java source becomes bytecode, which in turn becomes native code--and nobody said you can't do it all at once. The GNU Compiler for Java (GCJ) allows for a one-time, up-front compilation of Java source into an executable for a single platform. While incomplete--it does not support the Abstract Windowing Toolkit (AWT), thus making it unsuitable for AWT or Swing GUI programming--there's enough there to compile server-side and command-line applications.

This process has one obvious downside: cross-platform code becomes bound to a single platform in one step. Moreover, the static compilation is not an automatic trump of HotSpot's dynamic compilation--the author once worked on a project where the performance gain from GCJ was found to be less than five percent beyond the HotSpot version. Still, GCJ can solve important problems, like deploying a runnable Java application without having to worry whether a JVM is available or running a specific version.

The Java Community Process

Beyond language, libraries, and VM, there is the Java community. Despite the massive amount of open source software written in Java, there continues to be an open and obvious friction between the Java community and the open source community at large. Largely, this can be traced to Sun's unwillingness to release its Java implementation under a suitable open-source license, although the source is available under a variety of Sun-specified licenses.

Some say this conflict is utterly misguided. Developer Bruno Souza, speaking on a recent episode of O'Reilly's Distributing the Future podcast, said that this anti-Sun argument profoundly misunderstands the nature of Java, as the language, libraries, and virtual machine are all standards set by the largely open and transparent Java Community Process:

> All of the Java standards that are there can be implemented as open source. This distinction between if the Java standard is run or not by an organization outside of Sun, I don't think that matters that much. The most important thing is that the rules of the JCP are very clear ... The JCP is a very open standards organization. Of course, it's not perfect. But I think that one very important thing is that the standards the JCP generates, you can implement open source implementations of the Java standards. And that's extremely important, because that's the combination we want ....
>
> You say Java's not open source ... that's a totally meaningless statement. Because it means nothing to say that Java is or isn't open source. It's like saying HTTP is or isn't open source; it doesn't mean anything.

Indeed, the Apache Harmony project is developing what it intends to be a "world-class, certified J2SE implementation," available under the Apache License V2, all of which is permitted and encouraged by the JCP.

Beyond the JCP

Moving from the metaphorical cathedral to the bazaar, a massive range of Java projects exists outside of the standards body of the JCP. As noted before, Java is the top language for projects on SourceForge, and still more open source Java projects can be found at java.net, the Apache Jakarta Project, Javalobby's Javaforge, OpenSymphony, and countless independent sites. Many of these projects have grown to rival official JCP standards in mindshare, most obviously lightweight enterprise frameworks like the Spring framework, which has lured many developers frustrated with "official" specs like EJB 2.1. Independent projects have also been quick to adapt to changes outside of Java and bring in their best features, such as the Rails-inspired Trails, or the AJAX-simplifying Direct Web Remoting (DWR) project.

Conclusions

What a difference ten years and a few million developers make! It is time to sweep away old assumptions about Java: the coupling of language to VM, mischaracterizations of its status vis-a-vis the open source world, and predictable cheap shots about its performance or lack thereof. The next ten years of Java look to be entirely different, possibly forking not the language, but the focus, with many developers continuing to work in an evolving Java language that is run in various environments, while others run many different languages on the VM. Soon, asking "What is Java" may beg the question "which Java--the language or the VM?"

Chris Adamson is an author, editor, and developer specializing in iPhone and Mac.

Return to ONJava.com.