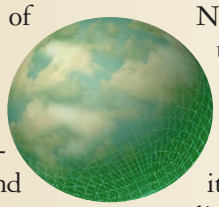
Paul Tyma

Why are we using JA

The practical answer you'll receive when asking anyone why they're programming in Java is based in the ultimate dream of platform-independence. Write an application in one place and it can run on any machine under any operating system (at least that's the idea). It certainly is a powerful statement, implying that we can lay down our architecture and operating system "bigotries" and allow performance to guide our computer purchases. No longer would one choose a certain system over another because "it has more applications," since all applications would run everywhere. This is not something I'd want to hear if I owned a company that had a monopoly on the CPU or operating system market.

Platform Independence

Platform independence in Java really takes two forms. The popular notion is that we write our code, compile it and then never worry about having to port it to new machines (write once, run everywhere). The more overlooked side of platform independence is Java's rather fantastic abstraction of many programming paradigms.



Productivity

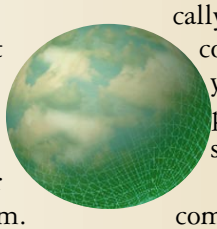
Java's automatic memory management, operating system abstractions (through its APIs), and familiar low-level syntax (at least for C programmers) make it a curiously productive environment. Most new Java programmers are surprised at how they are able to get something (anything) to work almost immediately. Before you're required to learn all there is about object-oriented programming, multithreading, remote method invocation, and the like, you can take a prebuilt piece of code, intuitively modify it, and get something up and running quite rapidly.

New C programmers inevitably write to some wayward pointer or commit some other fiendish memory-violating act that causes anything from "General Protection Faults" to system lockups. The guarded memory model in Java makes crashing the system much more difficult (although not impossible). Needless to say, crashing the system is a highly unproductive act.

The entire interaction with memory is worry-free in Java. When you want some memory you take it, when you're done with it, you walk away from it. This abstraction relieves the programmer of system "details." An alternative view (and one popular with C programmers during their transition to Java) is that Java is extremely limiting in terms of memory control. Compared to C, this is certainly true. However often, the word *flexibility* is synonymous with *responsibility*. Humans inevitably make mistakes and often fail to be responsible with memory management (dangling pointers, memory leaks, and so forth). Java relies on the fact that

VA AGAIN?

algorithms don't make mistakes and controls memory for you (at the cost of some run-time performance). It's been my experience that most hardcore C programmers eventually embrace this productive paradigm in spite of their lost flexibility.



Java programmers need only learn one set of APIs for all operations. Writing network code is the same regardless of which platform they're using. In C and C++ environments there are a plethora of choices for networking APIs—in fact, this is the problem. Two C programmers writing networking programs might not even be able to describe their respective projects to each other. Java programmers are always on the same level, regardless of their underlying environments. The number of APIs is vast, but if you learn them once, that's all you need to know (until the next release, of course).

Plug-in Components

The true dream of OO programming is the development of reusable software components. This includes classic data structures such as binary trees and hash tables, and very often user-interface components such as buttons, checkboxes, and scrollbars. These are various types of components that many diverse programs can use. If you buy a library of these types of components then you can speed your development by just "plugging in" what you need into your own code. Java's buzzword for its formalization of reusable software components is the Java "bean."

Often non-OO programmers argue that they can already reuse their code, just a cut-and-paste here or there and voilà! Of course, this means you need access to the source code which isn't always available. It's likely that reusing a piece of code will require slightly different modifications each time it is reintegrated. Beans (and other component models) formalize the process, giving programmers solid rules on how to interface to a given component. In fact, things are so well laid out that the integration can be done graphically in a builder tool that allows drag-and-drop code development. By connecting a few beans, you can literally say, "When this button is pushed, start this timer ticking," without ever seeing the code for the button or the timer.

Java's beans are designed to interact with components built for other systems. Soon you'll have thousands of beans to choose from, allowing you to build the core pieces of your Java application and use the beans to set up the details. This will increase productivity by leaps and bounds.

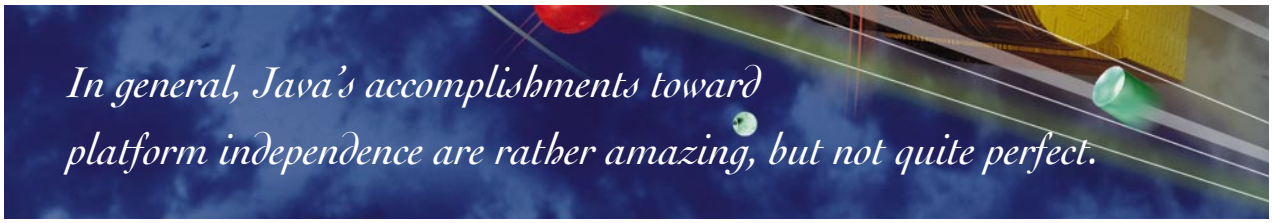
Write Once, Run Everywhere

The other side of platform independence is unexpectedly the darker side. This is the "write once, run everywhere" end-all solution. We're talking about one executable program running on any popular architecture and operating system. A Java executable is a set of instructions in Java bytecode. This is the machine language for the Java Virtual Machine (JVM) [1]. JVMs don't just happen to exist on every known platform; it takes serious work to create a JVM for each and every operating system. Every JVM is then expected to accept a validly compiled Java program and execute it in exactly the same way as any other JVM. This sounds like an insurmountable task, and so it is.

Platform independence at this level is not yet a total reality. There are many subtle idiosyncrasies to be found with every new operating system and a few major problems, too. For one thing, many Java programmers have quickly learned the difference between preemptive and non-preemptive multitasking operat-

ing systems. In preemptive operating systems (Windows 95/NT) processes can be stopped so that other processes have a chance to run. In other words, no single process should be able to hog the entire CPU. In non-preemptive operating systems (Mac OS) a process only gives up the CPU when it wants to. So Java programmers can write a multithreaded program that runs well in one OS and does not run at all in another. A Java programmer can write programs to ensure they run on both types of operating systems (in fact, today it's a requirement) but it can be argued that platform-independent programs should be either right or wrong (either they always run or they never run), not somewhere in between.

Another noticeable thorn in the side of this platform-independence idea is the design of user-interfaces. Java takes on the persona of whatever machine it is on: Run a Java program utilizing buttons and scrollbars on a Macintosh and it looks like a Mac application; run it under Windows and it looks like a Windows application. Unfortunately, every windowing system seems to treat its buttons and such just a little differently. It's not uncommon to set up an entire user interface on one machine only to see it turn out incorrectly when run on another.



In general, Java's accomplishments toward platform independence are rather amazing, but not quite perfect. As a shining example, when Sun Microsystems announced the 1.0 release of its Java development environment "Java Workshop," it was touted as "completely written in Java" [3]. It turns out that Java Workshop now only runs on Windows and Solaris platforms (not exactly platform independent). Apparently, even Sun discovered that programming straight Java couldn't get the job done completely.

The worst thing about architecture independence is that it keeps you independent of the architecture—meaning that Java needs to assume the lowest common denominator of available resources. Java supported only one mouse button because Macs only have one mouse button. Java can only assume very generic things: it assumes a system has a CPU, memory, and a graphics subsystem. It also assumes the operating system is multithreaded, which negates the idea of Java on Windows 3.1 (IBM subsequently released alphaworks, adding the required support). Joysticks, special keypads, and the like can't be accessed from Java. Only in the latest releases are we seeing the ability to print. To get system-specific, you need to write some (maybe just a little) C code and interface it with Java.

The problem is that if your program is even 1% C code, you lose Java's platform independence and its solid security model. C code must be compiled for a specific machine and the Java run time has no governing authority over what that C code can do (sure, it might "say" it's just going to check the joystick…). There are definite applications in which C and Java fit together, however, not in 100% platform-independent products.

A New Programming Language

Another possible reason Java is so popular is that maybe we just needed a new language. After all, other popular languages are at least 10 years old. That's about 250 years in "computer age" (presuming a "dog's age" is 7:1; a "computer's age" is more like 25:1). The rest of the computer industry has updated hardware, software, and so forth yearly, if not monthly, while our languages of choice have evolved but have not fundamentally changed. There's an inherent resistance to changing computer languages, since polished programmers must scrap their syntactical understanding of creating a program and start again. There's an old adage in the computing industry: "We have no idea what our primary computing language will look like in 20 years, but we know it will be called Fortran." How true it is, or at least, how true it was.

The sluggishness of change is by no means the fault of computer scientists; it's difficult to find a copy of the proceedings from an ACM conference on programming languages that does not introduce some new language. Many even sport catchy names like Java; previously, there have been ALLOY, SISAL, COOL, PIZZA, SELF, and many more.

Many of these even implemented the same ideas as Java: A straightforward object-orientation model, multithreading, and platform independence. In fact, the well-read computer scientist can tell you that Java invented nothing at all. Its object model borrowed interfaces from Objective-C, single-ancestor inheri-

tance hierarchy from Smalltalk, and other bits from Self and C++. Multithreading has been found in many different libraries in C and C++ for years. The synchronization model was invented in the early 1970s (that's about 625 years old in "computer age") by Per Brinch Hansen and/or Edsgar Dijkstra (they still argue, we still don't know).

So why is every college student and major corporation ignoring all those other shiny, new languages and trying to use Java? For academics, a major reason is that Java packages all the previously mentioned features together. In many universities, Java has replaced C++ as the first language freshman students learn. There's good reason for this. It has a "simpler" object model than C++ and most of what's learned helps students learn C or C++ later if they choose. Also, its standard library of APIs allows students to get results quickly.

Industry has largely moved in this direction for the promise of platform independence. This idea did exist before Java, it's just that the other great unsung languages never got the ear of industry (or the marketing of Sun). The bandwagon effect is taking hold and industries are willing to bet on a new language if they believe their peers already have.

It would seem that everybody is happy. Academics are by no means disappointed in Java. It takes the best ideas developed over the past 10 years and incorporates them into one powerful and highly supported language. Industries don't mind either; they can develop their applications in a platform-independent and robust manner. So, with all these wonderful reasons supporting its use, you would think the world would be filled with Java programs. So where are they? Java has been out for well over two years and very few applications have made it to market.

In the early days of Java (approximately 50 "computer years" ago), the major thrust was "Web applets." Everyone saw the potential of Web pages with dancing bears on them. After Netscape licensed Java, we jumped in realizing that not only would this be cool, but it was going to have backing.

Unfortunately, the idea of slick "Java-powered" Web pages stalled. Instead of moving like greased lightning, it turned out that Java loaded into Web pages with kind of a clunk—the ubiquitous gray box sitting there while the browser loaded, checked, and fired up the applet. Animations written in Java quickly disappeared when everyone realized that browsers already supported the GIF89a format (good thing too). The death of the "dancing bear" applets wasn't necessarily bad because it evolved Java's focus into the idea of "thin clients" and the ability to make Web-enabled applications.

The notion of creating full-blown platform-independent Java applications also has hit a few speed bumps. Fifty "computer years" is a long time, you'd think by now that full-blown Java applications would be running amok. The idea is stellar—write a word-processor application in Java once, and everyone on every platform could run it immediately. In fact, plug-in modules or even updates could be downloaded from the Web to keep the program up-to-date. This is cool. Where is this?

See Java Crawl

Many companies are currently working on improving the slow performance as well as the word-processor problems. But everyone woke up to some interesting facts hiding behind the whole Java paradigm. The plain truth is: Java is slow. Java isn't just slow, it's *really* slow, *surprisingly* slow. It is "you get to watch the buttons being drawn on your toolbar" slow. There are a lot of good reasons (hiding a lot of hard problems) for this, but that doesn't make buyers of word processors any happier.

The reason for Java's speed can be simplified in one word: abstraction. To OO/high-level language designers, abstraction is the goal. To the computer, any abstraction puts the code another step away from what it understands. It is also the computer's job to decode those steps down to its machine language so it can actually run the program.

Machine language coding is the lowest level we can code in (and even that is an abstraction). Above that, we can abstract machine code to assembly code. Then the computer geniuses invented the high-level language abstraction (Cobol, Fortran, and so forth). Java puts the object-orientation abstraction on top of this (Sun's new Hotspot product [2] is doing a lot to help remove this layer to increase performance). Finally, Java adds the platform-independence abstraction (Java's stack-based architecture is typically mapped to a register-based one). That's a lot of abstraction and it takes a lot of compilers, optimizers, and smart run-time environments to remove them all, to get our program back down to machine code so it can actually run.

Everyone in the Java industry knows about Java's addiction to abstraction and many are trying to fix it. The compiler companies came out with their just-in-time (JIT) compilers. These programs (which are closer to assemblers than compilers) convert Java's stack-based intermediate representation (it's executable) into native machine code immediately prior to execution on your machine. That way, the Java program actually runs as a real executable (that is, straight in machine code).

Curiously, the term "just-in-time" implies that they actually met the deadline they tried for. In actuality, you are forced to wait for the JIT compiler to do its stuff. The JIT compiler only goes into action between the moment you say "run" and the time the program actually does run. The entire execution of a JIT compiler is waiting time to the user. In reality, a more appropriate name might be "Wait to the Last Minute Holding Everybody Up" compilers but I doubt that would get past marketing. In addition, the efficiency of the machine code these JIT compilers produce is questionable. There are a lot of ways to implement a JIT compiler—rushing it to market can often influence the design.

In any case, using a JIT compiler promises some significant speedup and there is no question that JIT compilers help, but that's not the whole story. Every Java program is really only running in Java for a percentage of its time. The Java environment seemingly knows everything about every architecture. Think about it—how does the Java system know how to draw a line on a screen for any computer on any operating system? In reality, it doesn't, but it does know who to ask. Every operating system Java runs on already has routines (written in C or C++) that do these things. Whether it involves drawing a button, starting a thread, printing to the screen, or opening a network connection, the best Java can do is ask the operating system to get it done.

Depending upon the application, a Java program might run anywhere from 20% to 90% in C. JIT compilers can only speed up the Java part; they can do nothing about the operating system calls. For something like a bubble sort (mostly Java), you might speed up by a factor of 6 where a graphics-intensive program might only speed up by a factor of 2 [5].

There is no argument—JIT compilers do speed up Java programs. But to be honest, after comparing Java programs that use JIT compilers to normal C and C++ applications, it's still slow. JIT compilers are also hard to make compared to interpretive environments (Java's first home) and they only exist for a few select platforms (Windows 95/NT and Mac). That hardly makes Java platform independent or fast does it?

The good news is that Java's compilers and run times are basically in their infancy. There is much room for improvement. The term "Java optimizing compiler" is so far an oxymoron. Current compilers (JDK 1.1) do almost no optimization (dead-code elimination, constant propagation, and method inlining to be exact). Java run times are also catching up. Sun's Hotspot run time shows significant promise to outperform current JIT compiler run times.

I keep reading that some given run time or compiler makes Java performance equivalent to C or C++. This cannot happen. Java performance could someday approach C or C++ performance, but could never equal (or better) it. The reasons are seemingly endless, but some glaring hurdles are: Java has asynchronous garbage collection; Java has run-time, array-bounds checking; Java has run-time type checking; Java is dynamically linked; to cite a few examples.

Java for the Future

In spite of the bad news I've just presented about "almost" platform independence and poor performance, the good news is that Java is getting better. The language is not easy to learn compared to Basic, but it is arguably easier than C++. Compilers exist to compile Java into native executable files, which allow us to use Java for the sake of Java, because it is an interesting and powerful language. Compilers also exist for other languages (Ada95) [5] that compile those languages to Java bytecode so platform independence can be achieved without using Java. This truly is the entry into open computing. Lesser-known machines and operating systems are bound to increase in popularity. The time it takes to get applications to market will speed up because Java is a productive environment and because no time will be wasted porting those applications to other interested architectures.

Java programmers are already in massive demand for projects that may require run times that don't yet exist. Java has certainly found its niche (which arguably is in the book-publishing business) and it doesn't look like it's going away soon. So welcome to Java, circa 1998: Go learn it, write a book about it, and go get some venture capital for your Java startup. It seems that's what everybody else is doing. C

References

1. Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.
2. Project HotSpot; www.javasoft.com/javaone/sessions/slides/TT01/tt01_4 3.htm.
3. Sunworld Online, May 1996; www.eu.sun.com/sunworldonline/swol-05-1996/swol-05-newproducts2.html#wk42.java-workshop.
4. Taft, S.T. *Programming the Internet in Ada 95.* Ada Europe '96.
5. Tyma, P. Tuning Java performance. *Dr. Dobbs J.* (Apr. 1996).

Paul Tyma (ptyma@preemptive.com) is Chief Scientist at PreEmptive Solutions in Euclid, OH.