CAFEBABE

Brian Hayes

A reprint from

American Scientist

the magazine of Sigma Xi, the Scientific Research Society

Volume 85, Number 4
July–Augustl, 1997
pages 304–308

CAFEBABE

Brian Hayes

An unfulfilled dream of computer science is the one true programming language, equally suited to all programs, all programmers and all computers. In years past committees and coalitions have tried to create or legislate such a language. Algol-60 was one of the first attempts; its greatest success, ironically, was in spawning dozens of new languages. A few years later PL/1 had the powerful backing of IBM. And the language called Ada had an even more imposing sponsor—the U.S. Department of Defense. And yet these languages too failed to conquer all.

The latest candidate for a computer *Ursprache* is a language called Java. This one is not the creation of a committee or an international standards board. Its origins are wonderfully humble. Java began as a language for programming "set-top boxes," the gadgets that are supposed to make TV interactive. So far, set-top boxes haven't made much of a splash, but Java has become an extraordinary marketing phenomenon, with the kind of promotion and product tie-ins you might expect of a newly released *Star Wars* movie.

Do I exaggerate? Well, maybe McDonald's will never give away Java trinkets, but Starbucks might. Barely two years after the language was introduced, there are Java magazines, Java conferences, Java videos. Usenet has a dozen Java newsgroups, with hundreds of messages every day. Java books have become an industry in their own right. (I review a few of them on page 389 of this issue of *American Scientist*). Java businesses are springing up everywhere—startups and spin-offs and new divisions of established companies. And as far as I know, Java is the only programming language ever to have its own venture-capital fund (initial capitalization: $100 million).

Java's ambitions extend beyond becoming the one and only programming language. Java is being proposed as a new "computing platform," which could supplant the various alliances of hardware and operating-system software that now dominate the world of desktop computing. In this vision Java would elbow aside not only C, Pascal, Lisp and other programming languages but would also replace Windows and Unix and the Macintosh operating system. It would even be built into the "embedded" computers in cellular telephones and home thermostats (not to mention set-top boxes).

Before presenting my thoughts on the Java phenomenon, I have a couple of disclaimers to put on the record. In the first place, I have a favorite programming language of my own, and it is not Java. I prefer to code in a dialect of Lisp called Scheme. Anyone who has followed the computer-language wars of recent decades will immediately know that I come from an enemy camp. To make matters worse, I don't drink coffee! I am therefore unmoved by all the subtle appeals to caffeine craving that turn up in the names of Java products: Roaster, JavaBeans, C@fé, Mocha, etc. I will try to keep my prejudices in check, but the reader should bear in mind that the following comments come from someone who doesn't know his latte from his cappuccino, and who sometimes dreads the prospect of a world overrun by undrinkable beverages and unthinkable languages.

Platform Agnosticism

The key to Java's widespread appeal is its promise of "platform-neutral" computing. As the Java ads put it (in a trademarked slogan): "Write Once, Run Anywhere."

The issue of adapting software to multiple computing platforms is a difficult and important one. Today, a program intended to reach a broad audience needs versions for the Macintosh, Windows 95, Windows 3.1, Windows NT, OS/2, at least a few varieties of Unix, and perhaps other systems as well. For the programmer, it's a logistic nightmare.

And it's not just a programmer's problem. For the ordinary consumer, too, the multiplicity of platforms has costs and risks. It means the software you buy today may have to be replaced if you switch platforms next year. Files you create on one system may not be readable on another. And, inevitably, some programs are simply not available for all platforms.

The compatibility quandary has been with us from the beginning of computing—or at least from the moment the *second* computer was booted up. But lately the problem has come to be seen as more urgent. The reason is the Internet and the client-server model of computing.

Brian Hayes is a former editor of American Scientist. *Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@amsci.org.*

If you are browsing the World Wide Web, it takes only a click of the mouse to transfer a page of text from a remote machine (the server) to your own computer (the client). Downloading an image is equally easy, or a sound file, an animation, a video clip. So why not click on a link to have the browser download and run a program? After all, the machines at both ends of the communications channel are computers. Running programs is what they do best. If you can read words, look at pictures, listen to music and watch TV over the Internet, surely you should be able to run programs too.

It's important to be clear about the nature of the transaction proposed here. Many common events on the Web cause a program to be run—but it runs on the server. For example, searching for a topic on AltaVista or one of the other Web index sites initiates a database query on the server machine. Sometimes it would be better to execute a program on the client computer. Running an interactive game or simulation on the server might overtax both the server itself and the communications channel. Downloading the program code and running it locally could make the software much more efficient and responsive.

The problem, of course, is figuring out *what* program code to download. Every client platform requires an entirely different instruction format; code for an Intel Pentium processor is nonsense to a PowerPC. Even if multiple versions of the software were kept on hand—like spare parts for various makes of automobiles—the server might not be able to identify the client, and so wouldn't know which version to send.

Another potential problem with downloading executable software is security. A program on a malicious Web site might offer to make you a millionaire but actually clean out your bank account. Prudent computers don't accept programs from strangers these days.

Java addresses both of these issues. Whereas programs in most other languages are compiled into instructions for a specific processor, Java programs are translated into platform-independent "byte codes." The byte codes are then interpreted by a "virtual machine," which generally takes the form of software running on a conventional computer. The same sequence of byte codes can run on any computer for which there is a virtual machine.

The safety of a Java program is assured in two ways. First, the virtual machine is constructed as a kind of padded room, isolated from the rest of the system, where even a program run amok cannot do much harm. Second, the incoming byte codes are examined by a "verifier" before they are executed, and any program found to break the rules is rejected.

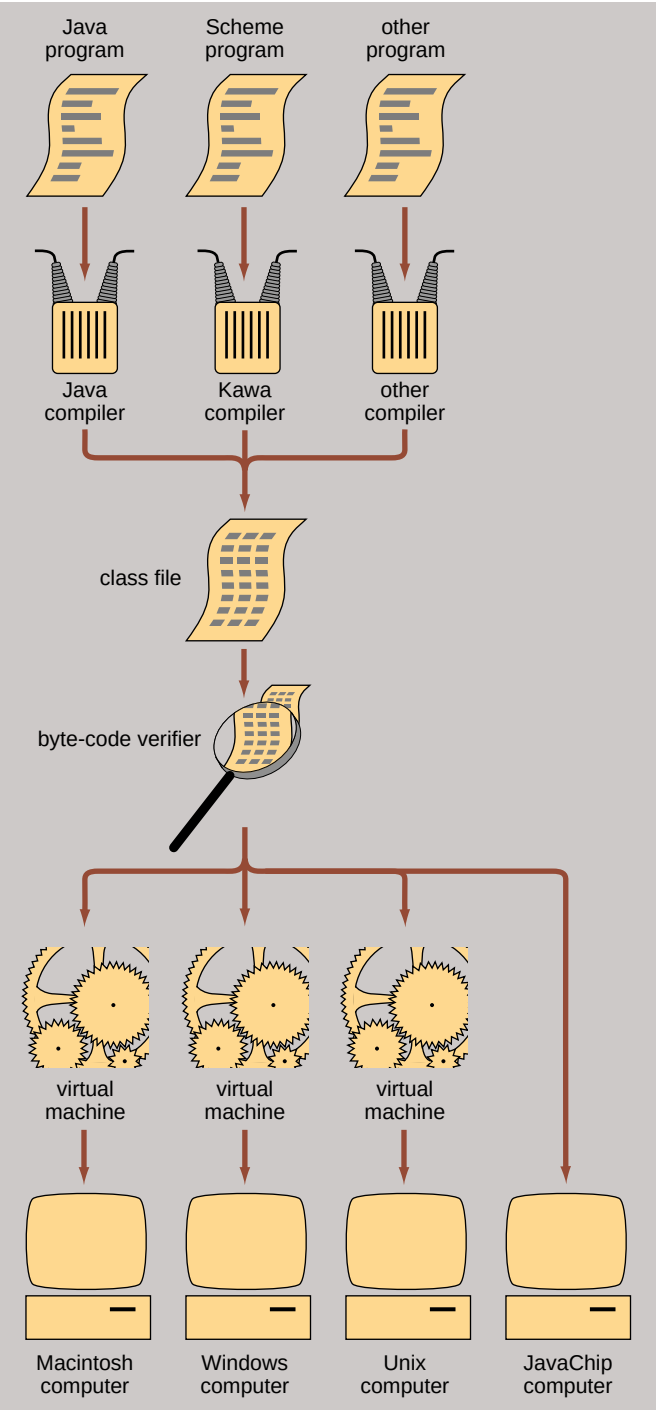

Figure 1. The Java language promises "platform-neutral" computing.

From Tiny Acorns …

Java was a programming language long before the World Wide Web was world wide, or even a web. The project that became Java began in 1990, when a group at Sun Microsystems, the leading maker of Unix workstations, set out to explore ways the company might enter the consumer-electronics market. A key member of this group was James Gosling, now a Sun vice president, who was probably best known at the time as the author of a version of the Emacs text editor. Gosling quickly put together the first version of a programming language meant for networked electronic appliances, such as set-top boxes or video games. The language was initially called Oak; the name was later changed to avoid a trademark conflict.

The consumer-electronics group at Sun had some hard years, which is not surprising since the market they aimed to serve is still inchoate. Then the Web came along, and suddenly their solution had found its problem. In 1994 two Sun engineers wrote a Web browser in Java, a predecessor of the HotJava browser available today. Then in May of 1995 Sun publicly released Java as a language for the Internet.

I don't mean to give the impression that Sun stumbled into the Java bonanza by blind luck. The company has a tradition of visionary thinking in computer networking. Indeed, Sun's corporate slogan was "The network *is* the computer" at a time when I for one didn't have a clue what that was supposed to mean.

… Mighty Coffee Beans Grow

Java is a descendant of the C programming language, which itself rose to celebrity from modest origins. C was developed in the 1970s by Dennis Ritchie of Bell Laboratories. At the outset it was closely associated with the Unix operating system; Unix was written in C, and the first C compilers ran under Unix. Soon the language spread to other systems. By now it has become so much a part of the mainstream that few remember just how idiosyncratic C once seemed. It is a language with a strict type system—the programmer must declare in advance what kind of data each variable can hold—and yet it allows many operations that other languages prohibit as reckless stunts. A prime example is the direct manipulation of "pointers," which represent the addresses of data items.

C has a distinctive, terse syntax, heavy on punctuation marks and symbols from the top row of the typewriter keyboard. Figure 2 gives an annotated example of what a very small C program looks like.

The main successor of C is C++, a language created in the 1980s by Bjarne Stroustrup, also of Bell Laboratories. The name is meant to suggest "C incremented," or "a little more than C." In fact it's a lot more—not just C + 1 but maybe C × 2. The major addition is a facility for object-oriented programming, a technique first explored years earlier in some very different programming languages, such as Simula and Smalltalk.

Two main ideas lie behind object-oriented programming. First, programs are assembled not from separate data structures and procedures but from software "objects" that encapsulate both data and procedures. For example, a triangle object might include as data the coordinates of its three vertices and as procedures the methods for finding the triangle's center, area and altitude. The second idea is inheritance: Classes of objects are organized into a hierarchy extending from the general to the specific. The triangle might be a member of the class of polygons, from which it would inherit properties common to all polygons. At the same time the triangle class could be further specialized in subclasses for right triangles, equilateral triangles and so on.

On the surface, a C++ program looks just like one in C. But this similarity is deceptive; at the semantic level, algorithms in C++ require a thorough rethinking. Going from C to Java is similarly tricky. On the surface, again, Java also looks like the same old C, but underneath all is changed.

Like C++, Java is an object-oriented language. Indeed, the object methodology is enforced in Java, whereas the C++ programmer can fall back into standard C. On the other hand, Java is not a "pure" object language in the way that Smalltalk is. In Smalltalk, *everything* is an object, but in Java certain elementary data types, such as numbers and characters, do not have object status—they are not members of a class hierarchy.

Perhaps the most important change in going from C or C++ to Java is the abolition of pointers. A C program steps through the elements of an array by taking a pointer to the first element—the pointer is effectively the array's address in memory—and repeatedly incrementing it. Java has a special form for accessing array elements without pointer arithmetic. Linked data structures in C also rely on pointers; in a list structure, for example, each item in the list includes a pointer to the next item, so that a program can step through the list by following the chain of pointers. Java provides for linked object references without exposing the machinery of pointers or addresses to the programmer.

Another major innovation in Java is automatic storage reclamation, otherwise known as garbage collection. If a C program allocates memory to a data structure, it had better remember to release the space once the structure is no longer needed, or all of the computer's memory could be clogged with waste. Java programs automatically clean up after themselves. If an object is no longer needed, the garbage collector sweeps it up. (I speak of this mechanism as an innovation because it is new to the world of C-like languages, but Lisp systems have had garbage collection since about 1960.)

Two more notable features of Java are exceptions and threads. Exceptions are a means for gracefully handling run-time errors and other unexpected events. In many languages a simple routine for reading a disk file can become incomprehensible because the normal outcome (where the file is read successfully) is hidden in a thicket of statements needed to check for possible errors. (What if the file doesn't exist? What if it can't be opened? What if it ends prematurely?) Java programs clear up the clutter by "throwing" an exception to another routine, which "catches" it. (Just for the record, "catch" and "throw" are also Lisp concepts.)

Threads are a mechanism for dividing a program into multiple concurrent processes. For example, a program searching through a large database might start several threads, each one looking for a different key. Of course the threads will truly run at the same time only on a computer with multiple processors, but the Java virtual machine's scheduling algorithm simulates concurrency even on a single processor. Java's implementation of threads is admirably lucid.

A few cosmetic improvements in Java are also commendable. Java programs are not limited to the ASCII character set but are written in Unicode, which accommodates a broader selection of the world's (human) languages. Hence a constant can be named π instead of pi. Also, comments documenting Java programs can employ the tags of Hypertext Markup Language, so that the programs are easily formatted for readability with a Web browser.

Figure 3 gives the Java equivalent of the C program in Figure 2. It still has the same algorithmic core, but now that core is wrapped in a thick blanket of object-oriented fur.

All in all, Java seems a distinct improvement over C, and yet it is not the programming language I would choose to be stranded on a desert island with. Its strengths are directed to the needs of the professional software engineer; it is not so well adapted to the kind of exploratory or experimental programming whose aim is not to build a software product but merely to answer a question or calculate a value. (I suspect that a lot of computing in the sciences is of this nature.) Java also seems to me less than ideal as a medium for reasoning about algorithms, and for teaching some of the fundamental ideas of computer science. Not that it can't be used for these purposes; it simply would not be my first choice.

A world where Java is the only programming language is therefore not a vision I greet warmly. But the prospect is not one that keeps me awake nights. One reason is that even if Java achieves its most grandiose ambitions—if it becomes the universal language of computing—it holds within itself the seed of a new confusion of tongues. That seed is the Java virtual machine.

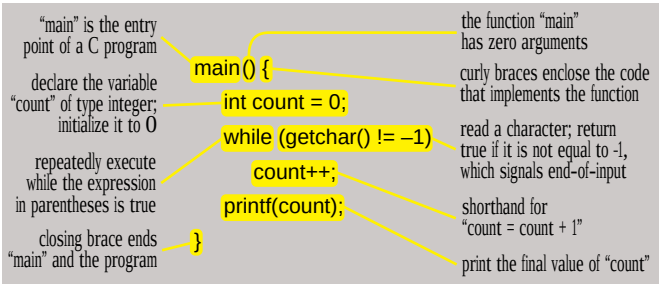

Figure 2. A program in C counts characters typed at the keyboard.

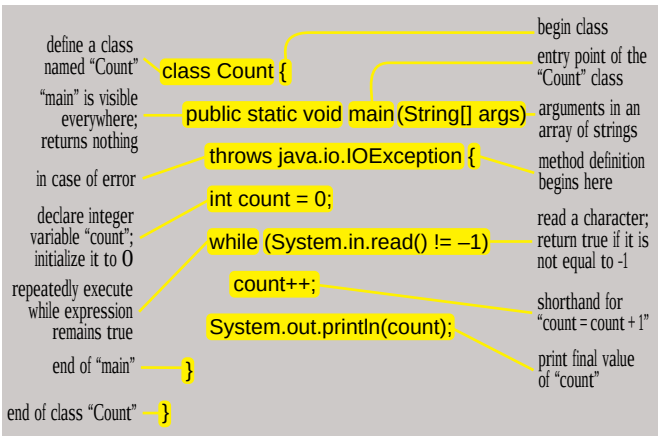

Figure 3. The character-counting program translated into Java.

The Virtual Machine

The idea of a virtual machine is hardly new to Java. It goes back to the very origins of computer science and is one of the many ingenious inventions of Alan M. Turing. The idea is that any sufficiently powerful computer can emulate, or mimic, any other computer. Such emulation is not just a theoretical toy. Practical emulators allow a Macintosh or a Unix box to dream it is a PC. Likewise, an emulator allows just about any computer to act as a Java virtual machine.

The virtual-machine strategy has a simple combinatorial advantage. Writing N programs for M platforms calls for an amount of labor proportional to $N \times M$. With a virtual machine the work needed is $N + M$. The N operations are needed to write one version of each program; the M operations consist of building the virtual machine for each platform. In the 1970s this approach to software portability was tried in the P-code system, developed at the University of California at San Diego. P-code was intended to be a universal intermediate language. Compilers for many high-level languages could generate P-code, which would be run by interpreters on various computers.

In the case of Java, the intermediate language consists of byte codes, which make up the instruction set of the virtual machine. Because there are just 256 eight-bit bytes, the machine's repertory of actions is limited to no more than 256 instructions. The architecture of the virtual machine is centered on a "pushdown stack," where values are stored while operations are pending. Consider the sequence of three instructions *iload0*, *iload1* and *iadd*, which Java happens to encode in the bytes whose decimal values are 26, 27 and 96. The two *iload* instructions push two local variables onto the top of the stack. Then *iadd* pops the two numbers off the stack, adds them and pushes the sum on the stack in their place. The *i* prefixed to each instruction indicates that the operands must be integers; there are equivalent instructions for other data types, such as floating-point numbers.

When a Java program is compiled, the output, called a class file, is not just a stream of byte codes. The file format includes several additional fields, structures and markers. For example, every valid class file must begin with a magic number, 3405691582. (The number seems less arbitrary when it is written in hexadecimal notation, where the 16 digits run from 0 to 9 and A to F. Converted to base 16, the magic number is CAFEBABE.)

The byte-code verifier ensures that a class file has the right format, and it also runs many checks on the byte codes themselves. In analyzing the three-byte program fragment given above, the verifier would make sure that both of the operands are integers, and it would prove that the stack cannot overflow or underflow. These checks enhance the reliability of Java programs, since type mismatches

and stack failures are errors that would likely cause the program to crash. The same checks are also the main line of defense against malicious software. (Java's armor against hostile programs has been found to have a few chinks, but so far most of them have been flaws of implementation, not design.)

Interestingly, one thing the byte-code verifier cannot verify is that a class file was actually generated by a Java compiler, rather than coming from some other source. Since the format of the class file has been spelled out in complete detail, a compiler for another language can emit byte codes that will be executed by the Java virtual machine just as if they were authentic Java. Note that these cuckoo-egg byte codes are subject to the same defenses against malicious programs, since the ersatz class file has to pass through the verifier. In effect, the Java language and the Java virtual machine are completely decoupled. Programs written in any language can be compiled into byte codes and run on the Java virtual machine; they thus gain the benefits of platform independence. Conversely, Java programs could be compiled for platforms other than the virtual machine.

Hijacking the Java virtual machine in this way is not just a hypothetical possibility. Per Bothner of Cygnus Solutions has written a compiler called Kawa that translates Scheme—my own pet language—into Java byte codes. Furthermore, the Kawa compiler is itself written in Java, so that it will run on any platform that has a Java virtual machine. Other languages, including Ada, are being grafted into Java in the same way.

The one nagging doubt about this ruse for fooling the virtual machine has to do with efficiency. The architecture of the virtual machine was designed to be a good match for typical Java programs; it is probably less than optimal for very different languages such as Scheme. But efficiency is a troublesome issue even for the "100% Pure Java" that Sun advocates. Compiling a program into byte codes, rather than into the "native code" of a specific processor, interposes a layer of interpretation that inevitably slows execution. This penalty may be acceptable for the occasional Java "applet" downloaded from a Web site and run once or twice; it will be intolerable if the major applications that people work with every day are rewritten in Java. (Corel Corporation has announced plans to publish Corel Office for Java, a suite of Java programs including a word processor and a spreadsheet.)

Sun's answer to the efficiency problem is the JavaChip—a microprocessor whose native instruction set consists of Java byte codes. Thus the virtual machine becomes real, and the overhead of interpretation is eliminated. But with this vision Java has come full circle. It is no longer a bridge between platforms but a new platform competing with all the others.

Meanwhile, Java has not quite reached the promised land of platform-independence even among the existing platforms. The small Java program of Figure 3 is one of the first examples given in *The Java Tutorial*, by Mary Campione and Kathy Walrath. The source code is the same for all platforms, but the tutorial's instructions for running the program are different for Unix, Windows and Macintosh computers. What's worse, the program also produces different results for each platform! (The source of these differences is that the program counts characters typed at the keyboard, and line-ends are encoded differently by the three operating systems.)

Babeling On

Talk of a universal language inevitably brings to mind that unfortunate incident of the tower. We tend to read the Babel story as a myth about hubris: The tower builders were out of their depth in trying to erect a structure as high as the heavens; they had undertaken a project beyond their abilities. But the wording of Genesis chapter 11 allows a different interpretation: "And Yahweh said, 'If this is how they have started to act, while they are one people with a single language for all, then nothing that they may presume to do will be out of their reach. Let me, then, go down and confound their speech there, so that they shall not understand one another's talk.'" In other words, the problem was not that the architects were incapable of raising such a tower; on the contrary, they had to be stopped precisely because they would have succeeded, and then doubtless gone on to even greater glories. Thus the story seems to be about the hidden dangers of good engineering practice. I'm not sure what that portends for the future of Java.

Bibliography

Arnold, Ken, and James Gosling. 1996. *The Java Programming Language*. Reading, Mass.: Addison-Wesley.

Campione, Mary, and Kathy Walrath. 1996. *The Java Tutorial: Object-Oriented Programming for the Internet*. Reading, Mass.: Addison-Wesley.

Corel Office for Java. <http://officeforjava.corel.com/>.

Gamelan: The Official Directory for Java. <http://www.gamelan.com/>.

The JavaSoft Home Page. <http://java.sun.com/>.

Kawa, the Java-based Scheme system. <http://www.cygnus.com/~bothner/ kawa.html>.

Kernighan, Brian W., and Dennis M. Ritchie. 1978. *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall.

Lindholm, Tim, and Frank Yellin. 1996. *The Java Virtual Machine Specification*. Reading, Mass.: Addison-Wesley.

McGraw, Gary. 1997. The Java security hotlist. <http://www.rstcorp.com/javasecurity/links.html>.

O'Connell, Michael. 1995. Java: The inside story. *SunWorld Online*. <http://www.sun.com/sunworldonline/swol-07-1995/swol-07-java.html>.

Speiser, E. A. (translator). 1964. *The Anchor Bible: Genesis*. Garden City, N.Y.: Doubleday and Co.

Stroustrup, Bjarne. 1986. *The C++ Programming Language*. Reading, Mass.: Addison-Wesley.

Sun Microelectronics. 1996. Picojava I microprocessor core architecture. <http://www.sun.com/sparc/ whitepapers/wpr-0014-01/>.

Yellin, Frank. 1996. Low level security in Java. http://www.javasoft.com/sfaq/verifier.html