Chapter 1

An Introduction to Java

- Java as a Programming Tool
- Advantages of Java
- The Java "White Paper" Buzzwords
- Java and the Internet
- A Short History of Java
- Common Misconceptions about Java

For a long time, to open a computer magazine that did not have a feature article on Java seemed impossible. Even mainstream newspapers and magazines like *The New York Times, The Washington Post,* and *Business Week* have run numerous articles on Java. It gets better (or worse, depending on your perspective): can you remember the last time National Public Radio ran a 10-minute story on a computer language? Or, a $100,000,000 venture capital fund is set up solely for products produced using a *specific* computer language? CNN, CNBC, you name the mass medium, it seems everyone was and to a certain extent is still talking about how Java will do this or Java will do that.

However, we decided to write this book for serious programmers, and because Java is a serious programming language, there's a lot to tell. So, rather than immediately getting caught up in an analysis of the Java hype and trying to deal with the limited (if still interesting) truth behind the hype, we will write, in some detail, about Java as a programming language (including, of course, the features

added for its use on the Internet that started the hype). After that, we will try to separate current fact from fancy by explaining what Java can and cannot do.

In the early days of Java, there was a huge disconnect between the hype and the actual abilities of Java. As Java is maturing, the technology is becoming a lot more stable and reliable, and expectations are coming down to reasonable levels. As we write this, Java is being increasingly used for "middleware" to communicate between clients and databases and other server resources. While not glitzy, this is an important area where Java, primarily due its portability and multithreading and networking capabilities, can add real value. Java is making great inroads in embedded systems, where it is well positioned to become a standard for hand-held devices, Internet kiosks, car computers, and so on. However, early attempts to rewrite familiar PC programs in Java were not encouraging—the applications were underpowered and slow. With the current version of Java, some of these problems could be overcome, but still, users don't generally care what programming language was used to write their applications. We think that the benefits of Java will come from new kinds of devices and applications, not from rewriting existing ones.

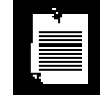
Java as a Programming Tool

As a computer language, Java's hype is overdone: Java is certainly a *good* programming language. There is no doubt that it is one of the better languages available to serious programmers. We think it could *potentially* have been a great programming language, but it is probably too late for that. Once a language is out in the field, the ugly reality of compatibility with existing code sets in. Moreover, even in cases where changes are possible without breaking existing code, it is hard for the creators of a language as acclaimed as Java to sit back and say, "Well, maybe we were wrong about X, and Y would be better." In sum, while we expect there to be some improvements over time, basically, the structure of the Java language tomorrow will be much the same as it is today.

Having said that, the obvious question is, Where did the dramatic improvements of Java come from? The answer is that they didn't come from changes to the underlying Java programming language, they came from *major changes in the Java libraries*. Over time, Sun Microsystems changed everything from the names of many of the library functions (to make them more consistent), to how graphics worked (by changing the event handling model and rewriting parts from scratch), to adding important features like printing that were not part of Java 1.0. The result is a far more useful programming tool, a tool moreover that, while not yet completely robust, is far less flaky than were earlier versions of Java.

But *every* version of Java has made major changes to the libraries of its predecessor, and the current version is no exception. In fact, the changes made in the

libraries are by far the most extensive in Java 1.2—there is roughly a ***doubling of the number of library functions*** over Java 1.1.

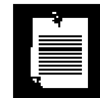


NOTE: Microsoft is producing a product called J++ that shares a family relationship with Java. Like Java, J++ is interpreted by a virtual machine that is compatible with the Java Virtual Machine for executing Java bytecodes, but there are substantial differences when interfacing with external code. The basic language syntax is almost identical to Java. However, Microsoft added language constructs that are of doubtful utility except for interfacing with the Windows API. In addition to Java and J++ sharing a common syntax, their foundational libraries (strings, utilities, networking, multithreading, math, and so on) are essentially identical. However, the libraries for graphics, user interfaces, and remote object access are completely different. Beyond the common syntax that is discussed in Chapters 3–6, we do not cover J++ in this book.

Advantages of Java

One obvious advantage is a run-time library that hopes to provide platform independence: you are supposed to be able to use the same code on Windows 95/98. NT, Solaris, Unix, Macintosh, and so on. This is certainly necessary for Internet programming. (However, implementations on other platforms usually lag behind those on Windows and Solaris. For example, as we write this, Java 1.2 is not even in beta for the Mac.)

Another programming advantage is that Java has a syntax similar to that of C++, which makes it economical without being absurd. Then again, Visual Basic (VB) programmers will probably find the syntax annoying and miss some of the nicer syntactic VB constructs like Select Case.



NOTE: If you are coming from a language other than C++, some of the terms used in this section will be less familiar—just skip those sections. You will be comfortable with all of these terms by the end of Chapter 6.

Java also is fully object oriented—even more so than C++. Everything in Java, except for a few basic types like numbers, is an object. (Object-oriented design has replaced earlier structured techniques because it has many advantages for dealing with sophisticated projects. If you are not familiar with Object-Oriented Programming (OOP), Chapters 4 through 6 provide what you need to know.)

However, having yet another, somewhat improved, dialect of C++ would not be enough. The key point is this: ***It is far easier to turn out bug-free code using Java than using C++.***

Why? The designers of Java thought hard about what makes C++ code so buggy. They added features to Java that eliminate the ***possibility*** of creating code with

the most common kinds of bugs. (Some estimates say that roughly every 50 lines of C++ code have at least one bug.)

- The Java designers eliminated manual memory allocation and deallocation.

  Memory in Java is automatically garbage collected. You *never* have to worry about memory leaks.

- They introduced true arrays and eliminated pointer arithmetic.

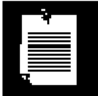
  You *never* have to worry about overwriting a key area of memory because of an off-by-one error when working with a pointer.

- They eliminated the possibility of confusing an assignment with a test for equality in a conditional.

  You cannot even compile `if (ntries = 3). . . .` (VB programmers may not see the problem, but, trust us, this is a common source of confusion in C/C++ code.)

- They eliminated multiple inheritance, replacing it with a new notion of *interface* that they derived from Objective C.

  Interfaces give you what you want from multiple inheritance, without the complexity that comes with managing multiple inheritance hierarchies. (If inheritance is a new concept for you, Chapter 5 will explain it.)



NOTE: The Java language specification is public. You can find it on the Web by going to the Java home page and following the links given there. (The Java home page is at `http://java.sun.com`.)

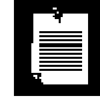
The Java "White Paper" Buzzwords

The authors of Java have written an influential White Paper that explains their design goals and accomplishments. Their paper is organized along the following eleven buzzwords:

Simple	Portable
Object Oriented	Interpreted
Distributed	High Performance
Robust	Multithreaded
Secure	Dynamic
Architecture Neutral	

We touched on some of these points in the last section. In this section, we will:

- Summarize via excerpts from the White Paper what the Java designers say about each buzzword, and
- Tell you what we think of that particular buzzword, based on our experiences with the current version of Java.



NOTE: As we write this, the White Paper can be found at `http://java.sun.com/doc/language_environment`. (If it has moved, you can find it by marching through the links at the Java home page.)

Simple

> *We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. So even though we found that C++ was unsuitable, we designed Java as closely to C++ as possible in order to make the system more comprehensible. Java omits many rarely used, poorly understood, confusing features of C++ that, in our experience, bring more grief than benefit.*

The syntax for Java is, indeed, a cleaned-up version of the syntax for C++. There is no need for header files, pointer arithmetic (or even a pointer syntax), structures, unions, operator overloading, virtual base classes, and so on. (See the C++ notes interspersed throughout the text for more on the differences between Java and C++.) The designers did not, however, *improve* on some stupid features in C++, like the `switch` statement. If you know C++, you will find the transition to Java's syntax easy.

If you are a VB programmer, you will not find Java simple. There is much strange syntax (though it does not take long to get the hang of it). More importantly, you must do a lot more programming in Java. The beauty of VB is that its visual design environment provides a lot of the infrastructure for an application almost automatically. The equivalent functionality must be programmed manually, usually with a fair bit of code, in Java. (See the following description of how Sun Microsystems is bringing the component-based, "glue" model to Java programming via "JavaBeans," a specification for developing plug-and-play components.)

At this point, the Java language is pretty simple, as object-oriented languages go. But there are many subtle points that you need to know to solve real-world problems. As time goes by, more and more of these details will be farmed off to libraries and development environments. Products like Sun's Java WorkShop, Symantec's Visual Café, and Inprise JBuilder have form designers that can make designing the interface of your programs easier. They are far from perfect but are a step forward. (Designing forms with nothing but the JDK is tedious at best

and unbearable at worst.) An ever-increasing supply of third-party class libraries and a lot of code samples (including many libraries) are actually freely available on the Net.

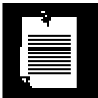
> *Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K bytes; adding the basic standard libraries and thread support (essentially a self-contained microkernel) adds an additional 175K.*

This is a great achievement. Note, however, that the GUI libraries are significantly larger.

Object Oriented

> *Simply stated, object-oriented design is a technique for programming that focuses on the data (= objects) and on the interfaces to that object. To make an analogy with carpentry, an "object-oriented" carpenter would be mostly concerned with the chair he was building, and secondarily with the tools used to make it; a "non-object-oriented" carpenter would think primarily of his tools. The object-oriented facilities of Java are essentially those of C++.*

Object orientation has proven its worth in the last 30 years, and it is inconceivable that a modern programming language would not use it. Indeed, the object-oriented features of Java are comparable to C++. The major difference between Java and C++ lies in multiple inheritance, for which Java has found a better solution, and in the Java metaclass model. The reflection mechanism (see Chapter 5) and object serialization feature (see Volume 2) make it much easier to implement persistent objects and GUI builders that can integrate off-the-shelf-components.



NOTE: If you do not have any experience with OOP languages, you will want to carefully read Chapters 4 through 6. These chapters explain what OOP is and why it is more useful for programming sophisticated projects than traditional, procedure-oriented languages like BASIC or C.

Distributed

> *Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system.*

We have found the networking capabilities of Java to be both strong and easy to use. Anyone who has tried to do Internet programming using another language will revel in how simple Java makes onerous tasks like opening a socket connec-

tion. Java even makes common gateway interface (CGI) scripting easier, and an elegant mechanism, called servlets, makes server-side processing in Java extremely efficient. Many popular web servers support servlets. (We will cover networking in Volume 2 of this book.) The remote method invocation mechanism enables communication between distributed objects (also covered in Volume 2).

Robust

> *Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (run-time) checking, and eliminating situations that are error-prone.… The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.*

This feature is also very useful. The Java compiler (in both its original incarnation and in the various improved versions in third-party implementations) detects many problems that, in other languages, would only show up at run time (or, perhaps, not even then). As for the second point, anyone who has spent hours chasing memory corruption caused by a pointer bug will be very happy with this feature of Java.

If you are coming from a language like VB that doesn't explicitly use pointers, you are probably wondering why this is so important. C programmers are not so lucky. They need pointers to access strings, arrays, objects, even files. In VB, you do not use pointers for any of these entities, nor do you need to worry about memory allocation for them. On the other hand, if you implement some of the fancier data structures in VB that require pointers using class modules, you need to manage the memory yourself. Java gives you the best of both worlds. You do not need pointers for everyday constructs like strings and arrays. You have the power of pointers if you need it, for example, for linked lists. And you always have complete safety, since you can never access a bad pointer, make memory allocation errors, or have to protect against memory leaking away.

Secure

> *Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.*

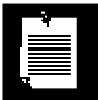
In the first edition of *Core Java* we said: "Well, one should 'never say never again,'" and we turned out to be right. A group of security experts at Princeton University found the first bugs in the security features of Java 1.0—not long after the JDK 1.0 was shipped. Moreover, they and various other people have continued to find other bugs in the security mechanisms of all subsequent versions of Java. All we can suggest is that you check

1. The URL for the Princeton group:
   `http://www.cs.princeton.edu/sip/`

2. The `comp.risks` newsgroup

for opinions from outside experts on the current status of Java's security mechanisms.

The good side is that the Java team has said that they will have a "zero tolerance" for security bugs and will immediately go to work on fixing any bugs found in the applet security mechanism (the browser companies go to work immediately as well). In particular, by making public the internal specifications of how the Java interpreter works, Sun is making it far easier for people to find any bugs in Java's security features—essentially enlisting the outside community in the ever-so-subtle security bug detection. This makes one more confident that security bugs will be found as soon as possible. In any case, Java makes it extremely difficult to outwit its security mechanisms. The bugs found so far have been very subtle and (relatively) few in number.



NOTE: Sun's URL for security-related issues is currently at:
`http://java.sun.com/sfaq/`

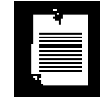
Here is a sample of what Java's security features are supposed to keep a Java program from doing:

1. Overrunning the run-time stack, like the infamous Internet worm did
2. Corrupting memory outside its own process space
3. Reading or writing local files when invoked through a security-conscious class loader, like a Web browser that has been programmed to forbid this kind of access

All of these features are in place and for the most part seem to work as intended. Java is certainly the most secure programming language to date. But, caution is always in order: though the bugs found in the security mechanism to date were not trivial to find and full details are often kept secret, still it may be impossible to *prove* that Java is secure. So, all we can do is repeat what we said before with even more force attached to it:

> *"Never say never again."*

Regardless of whether Java can ever be proved secure, Java 1.1 now has the notion of signed classes (see Volume 2). With a signed class, you can be sure of who wrote it. Once a signing mechanism is in place, any time you trust the author of the class, the class can be allowed more privileges on your machine.



NOTE: A competing code delivery mechanism from Microsoft based on its ActiveX technology relies on digital signatures alone for security. Clearly this is not sufficient—as any user of Microsoft's own products can confirm, programs from well-known vendors do crash and in so doing, create damage. Java has a far stronger security model than ActiveX since it controls the application as it runs and stops it from wreaking havoc. (For example, you can ensure that local file input and output is forbidden even for signed classes.)

Architecture Neutral

> *The compiler generates an architecture neutral object file format—the compiled code is executable on many processors, given the presence of the Java run time system. The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.*

This is not a new idea. More than twenty years ago, the UCSD Pascal system did the same thing in a commercial product, and even before that, Niklaus Wirth's original implementation of Pascal used the same approach. With the use of bytecodes, performance takes a major hit (but just-in-time compilers mitigate this, in many cases). The designers of Java did an excellent job developing a bytecode instruction set that works well on today's most common computer architectures. And the codes have been designed to translate easily into actual machine instructions.

Portable

> *Unlike C and C++, there are no "implementation-dependent" aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.*

For example, an `int` in Java is always a 32-bit integer. In C/C++, `int` can mean a 16-bit integer, a 32-bit integer, or any other size that the compiler vendor likes. The only restriction is that the `int` type must have at least as many bytes as a `short int` and cannot have more bytes than a `long int`. Having a fixed size of number types eliminates a major porting headache. Binary data is stored in a fixed format, eliminating the "big endian/little endian" confusion. Strings are saved in a standard Unicode format.

> *The libraries that are a part of the system define portable interfaces. For example, there is an abstract Window class and implementations of it for Unix, Windows, and the Macintosh.*

As anyone who has ever tried knows, it is an effort of heroic proportions to write a program that looks good on Windows, the Macintosh, and 10 flavors of Unix. Java 1.0 made the heroic effort, delivering a simple toolkit that mapped common user-interface elements to a number of platforms. Unfortunately, the result was a library that, with a lot of work, could give barely acceptable results on different systems. (And there were often *different* bugs on the different platform graphics implementations.) But it was a start. There are many applications in which portability is more important than the nth degree of slickness, and these applications did benefit from early versions of Java. By now, the user interface toolkit has been completely rewritten so that it no longer relies on the host user interface. The result is far more consistent and, we think, more attractive than in earlier versions of Java.

Interpreted

> *The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. Since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.*

Perhaps this is an advantage while developing an application, but it is clearly overstated. In any case, we have found the Java compiler to be quite slow. If you are used to the speed of VB's or Delphi's development cycle, you will likely be disappointed.

High Performance

> *While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at run time) into machine code for the particular CPU the application is running on.*

If you use the standard Java interpreter to translate the bytecodes, "high performance" is not the term that we would use ("middling to poor" is probably more accurate). While it is certainly true that the speed of the interpreted bytecodes can be acceptable, it isn't fast. (At best, Java is only slightly faster than VB4, according to our tests, and is not as fast as later versions of VB.) On the other hand, you will want to run many Java programs through a true compiler and not restrict yourself to interpreting the bytecodes. For example, you will almost certainly want to do so for any program that is designed to be a stand-alone application on a specific machine. Ultimately, you will want compilers for every platform.

Some native code compilers for Java are available, for example, from Asymetrix, Symantec and IBM. There is also another form of compilation, the *just-in-time* (JIT) compilers. These work by compiling the bytecodes into native code once, caching the results, and then calling them again if needed. This approach speeds up loops tremendously since one has to do the interpretation only once. Although still

slightly slower than a true native code compiler, a just-in-time compiler can give you a 10- or even 20-fold speedup for some programs and will almost always be significantly faster than the Java interpreter. This technology is being improved continuously and may eventually yield results that cannot be matched by traditional compilation systems. For example, a just-in-time compiler can monitor which code is executed frequently and optimize just that code for speed.

Multithreaded

> *[The] benefits of multithreading are better interactive responsiveness and real-time behavior.*

If you have ever tried to do multithreading in another language, you will be pleasantly surprised at how easy it is in Java. Threads in Java also have the capacity to take advantage of multiprocessor systems if the base operating system does so. On the downside, thread implementations on the major platforms differ widely, and Java makes no effort to be platform independent in this regard. Only the code for calling multithreading remains the same across machines; Java offloads the implementation of multithreading to the underlying operating system. (Threading will be covered in Volume 2.) Nonetheless, the ease of multithreading is one of the main reasons why Java is such an appealing language for server-side development.

Dynamic

> *In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment. Libraries can freely add new methods and instance variables without any effect on their clients. In Java, finding out run time type information is straightforward.*

This is an important feature in those situations where code needs to be added to a running program. A prime example is code that is downloaded from the Internet to run in a browser. In Java 1.0, finding out run-time type information was anything but straightforward, but current versions of Java give the programmer full insight into both the structure and behavior of its objects. This is extremely useful for systems that need to analyze objects at run time such as Java GUI builders, smart debuggers, pluggable components, and object databases.

Java and the Internet

The idea here is simple: users will download Java bytecodes from the Internet and run them on their own machines. Java programs that work on Web pages are called *applets.* (Actually, it is the bytecodes, rather than the source file, that you download and then run.) To use an applet, you need a Java-enabled Web browser, which will interpret the bytecodes for you. Because Sun is licensing the Java source code and insisting that there be no changes in the language and basic library structure, you can be sure that a Java applet will run on any browser that is advertised as Java

enabled. Note that Netscape 2.x and Netscape 3.x are only *Java 1.02 enabled*, as is Internet Explorer 3.0. Netscape 4 and Internet Explorer 4 run different subsets of Java 1.1. This sorry situation made it increasingly difficult to develop applets that take advantage of the most current Java version. To remedy this problem, Sun has developed the *Java Plug-in*, a tool that makes the newest Java run-time environment available to both Netscape and Internet Explorer (see Chapter 10).

We suspect that, ultimately, most of the initial hype around Java stemmed from the lure of making money from special-purpose software. You have a nifty "Will Writer" program. Convert it to an applet, and charge people per use—presumably, most people would be using this kind of program infrequently. Some people predict a time when everyone downloads software from the Net on a per-use basis. This might be great for software companies, but we think it is absurd, for example, to expect people to download and pay for a spell-checker applet each time they send an e-mail message.

Another early suggested use for applets was so-called content and protocol handlers that allow a Java-enabled Web browser to deal with new types of information dynamically. Suppose you invent a nifty fractal compression algorithm for dealing with humongous graphics files and want to let someone sample your technology before you charge them big bucks for it. Write a Java content handler that does the decompression and send it along with the compressed files. The HotJava browser by Sun Microsystems supports this feature, but neither Netscape nor Internet Explorer ever did.

Applets can also be used to add buttons and input fields to a Web page. But downloading those applets over a dialup line is slow, and you can do much of the same with Dynamic HTML, HTML forms, and a scripting language such as JavaScript. And, of course, early applets were used for animation: the familiar spinning globes, dancing cartoon characters, nervous text, and so on. But animated GIFs can do much of this and Dynamic HTML combined with scripting can do even more of what Java applets were first used for.

As a result of the browser incompatibilities and the inconvenience of downloading applet code through slow net connections, applets on Internet Web pages have not become a huge success. The situation is entirely different on *intranets*. There are typically no bandwidth problems, so the download time for applets is no issue. And in an intranet, it is possible to control which browser is being used or to use the Java Plug-in consistently. Employees can't misplace or misconfigure programs that are delivered through the Web with each use, and the system administrator never needs to walk around and upgrade code on client machines. Many corporations have rolled out programs such as inventory checking, vacation planning, travel reimbursement, and so on, as applets that use the browser as the delivery platform.

Applets at Work

This book includes a few sample applets; ultimately, the best source for applets is the Web itself. Some applets on the Web can only be seen at work; many others include the source code. When you become more familiar with Java, these applets can be a great way to learn more about Java. A good Web site to check for Java applets is Gamelan—it is now hosted as part of the developer.com site, but you can still reach it through the URL `http://www.gamelan.com`. (By the way, *gamelan* also stands for a special type of Javanese musical orchestra. Attend a gamelan performance if you have a chance—it is gorgeous music.)

To place an applet onto a Web page, you need to know or work with someone who knows hypertext markup language (HTML). The number of HTML tags needed for a Java applet are few and easy to master (see Chapter 10). Using general HTML tags to design a Web page is a design issue—it is not a programming problem.

As you can see in Figure 1-1, when the user downloads an applet, it works much like embedding an image in a Web page. (For those who know HTML, we mean one set with an IMG tag.) The applet becomes a part of the page, and the text flows around the space used for the applet. The point is, the image is *alive.* It reacts to user commands, changes its appearance, and sends data between the computer viewing the applet and the computer serving it.

Figure 1-1 shows a good example of a dynamic web page that carries out sophisticated calculations, an applet to simulate genetic variations in the fruit fly

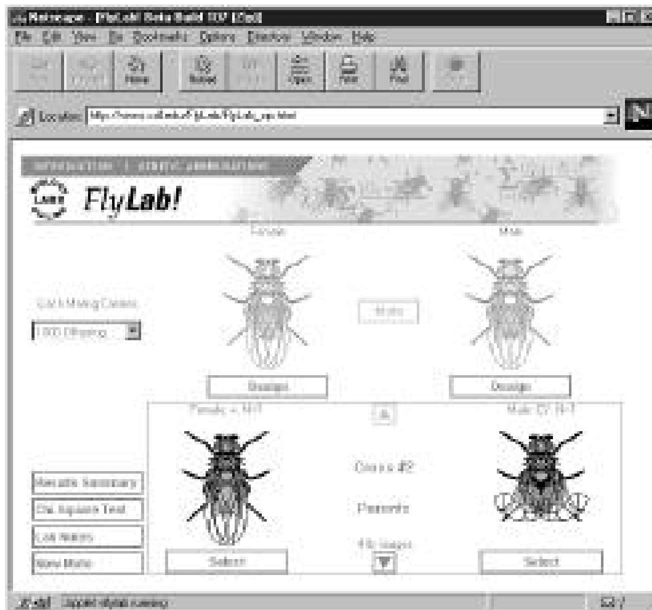


Figure 1-1: FlyLab Applet (`http://www.cdl.edu/FlyLab`)

species Drosophila Melanogaster. This is part of the Center for Distributed Learning of the California State University. Students design parent flies by setting features such as eye color or wing shape, and the Java program determines the possible features of their offspring.

A Short History of Java

This section gives a short history of Java's evolution. It is based on various published sources (most importantly, on an interview with Java's creators in the July 1995 issue of *SunWorld's* on-line magazine).

Java goes back to 1991, when a group of Sun engineers, led by Patrick Naughton and Sun Fellow (and all-around computer wizard) James Gosling, wanted to design a small computer language that could be used for consumer devices like cable TV switchboxes. Since these devices do not have a lot of power or memory, the language had to be small and generate very tight code. Also, because different manufacturers may choose different central processing units (CPUs), it was important not to be tied down to any single architecture. The project got the code name "Green."

The requirements for small, tight code led the team to resurrect the model that a language, called UCSD Pascal, tried in the early days of PCs and that Niklaus Wirth had pioneered earlier. What Wirth pioneered and UCSD Pascal did commercially, and the Green project engineers did as well, was to design a portable language that generated intermediate code for a hypothetical machine. (These are often called *virtual machines*—hence, the Java Virtual Machine or JVM.) This intermediate code could then be used on any machine that had the correct interpreter. Intermediate code generated with this model is always small, and the interpreters for intermediate code can also be quite small, so this solved their main problem.

The Sun people, however, come from a UNIX background, so they based their language on C++ rather than Pascal. In particular, they made the language object oriented rather than procedure oriented. But, as Gosling says in the interview, "All along, the language was a tool, not the end." Gosling decided to call his language "Oak." (Presumably because he liked the look of an oak tree that was right outside his window at Sun.) The people at Sun later realized that Oak was the name of an existing computer language, so they changed the name to Java.

In 1992, the Green project delivered its first product, called "*7." It was an extremely intelligent remote control. (It had the power of a SPARCstation in a box that was 6" by 4" by 4".) Unfortunately, no one was interested in producing this at Sun, and the Green people had to find other ways to market their technology. However, none of the standard consumer electronics companies were interested. The group then bid on a project to design a cable TV box that could deal

with new cable services such as video on demand. They did not get the contract. (Amusingly, the company that did was led by the same Jim Clark who started Netscape—a company that did much to make Java successful.)

The Green Project (with a new name of "First Person Inc.") spent all of 1993 and half of 1994 looking for people to buy its technology—no one was found. (Patrick Naughton, one of the founders of the group and the person who ended up doing most of the marketing, claims to have accumulated 300,000 air miles in trying to sell the technology.) First Person was dissolved in 1994.

While all of this was going on at Sun, the World Wide Web part of the Internet was growing bigger and bigger. The key to the Web is the browser that translates the hypertext page to the screen. In 1994, most people were using Mosaic, a noncommercial Web browser that came out of the supercomputing center at the University of Illinois in 1993. (Mosaic was partially written by Marc Andreessen for $6.85 an hour as an undergraduate student on a work-study project. He moved on to fame and fortune as one of the cofounders and the chief of technology at Netscape.)

In a *SunWorld* interview, Gosling says that in mid-1994, the language developers realized that "We could build a real cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we'd done: architecture neutral, real-time, reliable, secure—issues that weren't terribly important in the workstation world. So we built a browser."

The actual browser was built by Patrick Naughton and Jonathan Payne and evolved into the HotJava browser that we have today. The HotJava browser was written in Java to show off the power of Java. But the builders also had in mind the power of what are now called applets, so they made the browser capable of interpreting the intermediate bytecodes. This "proof of technology" was shown at SunWorld '95 on May 23, 1995, and inspired the Java craze that continues unabated today.

The big breakthrough for widespread Java use came in the fall of 1995, when Netscape decided to make the next release of Netscape (Netscape 2.0) Java enabled. Netscape 2.0 came out in January of 1996, and it has been (as have all subsequent releases) Java enabled. Other licensees include IBM, Symantec, Inprise, and many others. Even Microsoft has licensed and supports Java. Internet Explorer is Java enabled, and Windows ships with a Java virtual machine. (Note that Microsoft does not support the most current version of Java, however, and that its implementation differs from the Java standard.)

Sun released the first version of Java in early 1996. It was followed by Java 1.02 a couple of months later. People quickly realized that Java 1.02 was not going to cut it for serious application development. Sure, you could use Java 1.02 to make a

nervous text applet. (It moves text randomly around in a canvas) But you couldn't even *print* in Java 1.02. To be blunt, Java 1.02 was not ready for prime time.

The big announcements about Java's future features trickled out over the first few months of 1996. Only at the JavaOne conference held in San Francisco in May of 1996 did the bigger picture of where Java was going become clearer. At JavaOne the people at Sun Microsystems outlined their vision of the future of Java with a seemingly endless stream of improvements and new libraries. We were, to say the least, suspicious that all this would take *years* to come to pass. We are happy to report that while not everything they have outlined has come to pass, surprisingly much was incorporated into Java 1.1 in an equally surprisingly short amount of time.

The big news of the 1998 JavaOne conference was the upcoming release of Java 1.2, which replaces the early toy-like GUI and graphics toolkits with sophisticated and scalable versions that come a lot closer to the promise of "write once, run anywhere" than their predecessors. Again we were, to say the least, suspicious that all this would take *years* to come to pass. We are happy to report again that while not everything they have outlined has come to pass, surprisingly much was incorporated into Java 1.2 in an equally surprisingly short amount of time.

Common Misconceptions About Java

In summary, what follows is a list of some common misconceptions about Java, along with commentary.

Java is an extension of HTML.

Java is a programming language; HTMLis a way to describe the structure of a Web page. They have nothing in common except that there are HTML extensions for placing Java applets on a Web page.

Java is an easy programming language to learn.

No programming language as powerful as Java is easy. You always have to distinguish between how easy it is to write toy programs and how hard it is to do serious work. Also, consider that only four chapters in this book discuss the Java language. The remaining chapters of both volumes show how to put the language to work, using the Java *library.* The Java library contains over 150 classes and interfaces. Just listing every possible function and constant in the library along with a one- or two-sentence description of them takes more than 250 pages. (The annotated listing of all the Java 1.02 library, which is quite a bit smaller than the Java 1.1 library, included a reasonable number of code snippets, but it took more than 1500 printed pages!) Luckily, you do not need to know every function and constant listed in the 250 pages of the (hypertext-based) Java Application Programming User's Guide, but you do need to know surprisingly many of them in order to use Java for anything realistic.

Java is an easy environment in which to program.

The native Java development environment is not an easy environment to use—except for people who swear by 1970s command-line tools. There are integrated development environments that bring Java development into the modern era of VB-style drag-and-drop form designers combined with decent debugging facilities, but they can be somewhat complex and daunting for the newcomer. They also work by generating what is often hundreds of lines of code. We don't think you are well served when first learning Java by starting with hundreds of lines of computer-generated UI code filled with comments that say DO NOT MODIFY or the equivalent. We have found in teaching Java that using your favorite text editor is still the best way to learn Java, and that is what we will do.

Java will become a universal programming language for all platforms.

This is possible, in theory, and it is certainly the case that every vendor but Microsoft seems to want this to happen. However, there are many applications, already working perfectly well on desktops, that would not work well on other devices or inside a browser. Also, these applications have been written to take advantage of the speed of the processor and the native user-interface library and have been ported to all of the important platforms anyway. Among these kinds of applications are word processors, photo editors, and Web browsers. They are typically written in C or C++, and we see no benefit to the end user in rewriting them in Java. And, at least in the short run, there would be significant disadvantages since the Java version is likely to be slower, less powerful, and to use incompatible file formats.

Java is just another programming language.

Java is a nice programming language; most programmers prefer it to C or C++. But there have been hundreds of nice programming languages that never gained widespread popularity, whereas languages with obvious flaws, such as C++ and VB, have been wildly successful.

Why? Well we feel that the success of a programming language is determined more by the utility of the *support system* surrounding it, not by the elegance of its syntax. Are there useful, convenient, and standard libraries for the features that you need to implement? Are there tools vendors that build great programming and debugging environments? Does the language and the tool set integrate with the rest of the computing infrastructure? Java is successful on the server because its class library lets you easily do things that were hard before, such as networking and multithreading. The fact that Java reduces pointer errors is a bonus and so programmers seem to be more productive with Java, but these are not the source of its success. Whether Java's ability to make portable user interfaces will be important remains to be seen—the necessary support libraries were not there in versions of Java before Java 1.2.

This is an important point that one vendor in particular—who sees portable user interfaces as a threat—tries to ignore, by labeling Java "just a programming language" and by supplying a system that uses a derivative of the Java syntax and a proprietary and nonportable library. The result may well be a very nice language that is a direct competitor to VB but has little to do with Java.

Java is interpreted, so it is too slow for serious applications on a specific platform.

Many programs spend most of their time on things like user-interface interactions. All programs, no matter what language they are written in, will detect a mouse click in adequate time. It is true that we would not do CPU-intensive tasks with the interpreter supplied with the Java development kit. However, on platforms where a just-in-time compiler is available, all you need to do is run the bytecodes through it and most performance issues simply go away. Finally, Java is great for network-bound programs. Experience has shown that Java can comfortably keep up with the data rate of a network connection, even when doing computationally intensive work such as encryption. As long as Java is faster than the data that it processes, it does not matter that C++ might be faster still. Java is easier to program, and it is portable. This makes Java a great language for implementing network services.

All Java programs run inside a Web page.

All Java *applets* run inside a Web browser. That is the definition of an applet—a Java program running inside a browser. But it is entirely possible, and quite useful, to write stand-alone Java programs that run independently of a Web browser. These programs (usually called *applications*) are completely portable. Just take the code and run it on another machine! And because Java is more convenient and less error-prone than raw C++, it is a good choice for writing programs. It is an even more compelling choice when it is combined with database access tools like Sun's JDBC (see Volume 2). It is certainly the obvious choice for a first language in which to learn programming.

Most of the programs in this book are stand-alone programs. Sure, applets are interesting, and right now most useful Java programs are applets. But we believe that stand-alone Java programs will become extremely important, very quickly.

Java applets are a major security risk.

There have been some well-publicized reports of failures in the Java security system. Most have been in the implementation of Java in a specific browser. Researchers viewed it as a challenge to try to find chinks in the Java armor and to defy the strength and sophistication of the applet security model. The technical failures that they found have all been quickly corrected, and to our knowledge, no actual systems were ever compromised. To keep this in perspective, consider the literally millions of virus attacks in Windows executable files and Word

macros that cause real grief but, generally, little publicity. Also, the ActiveX mechanism in Internet Explorer would be a fertile ground for abuse, but it so boringly obvious how to circumvent it that few have bothered to publicize their findings.

Some system administrators have even deactivated Java in company browsers, while continuing to permit their users to download executable files, ActiveX controls, and Word documents. That is pretty ridiculous—currently, the risk of being attacked by a hostile Java applet is perhaps comparable to the risk of dying from a plane crash; the risk of the latter activities is comparable to the risk of dying while crossing a busy freeway on foot.

JavaScript is a simpler version of Java.

JavaScript, a scripting language that can be used inside Web pages, was invented by Netscape and originally called LiveScript. JavaScript has a syntax that is reminiscent of Java, but otherwise there are no relationships (except for the name, of course). A subset of JavaScript is standardized as ECMA-262, but the extensions that you need for real work have not been standardized, and as a result, writing JavaScript code that runs both in Netscape and Internet Explorer is an exercise in frustration.

Java eliminates the need for CGI scripting.

Not yet. With today's technology, CGI is still the most common mechanism for server-side processing. However, there are technologies on the horizon that greatly reduce the need for CGI scripts. Servlets give you the same execution environment on the server that applets have on the client. JDBC (see Volume 2) permits direct database manipulations by the client of information lying on the server.

Java will revolutionize client-server computing.

This is possible and it is where much of the best work in Java is being done. There are quite a few application servers such as Weblogic's Tengah that are built entirely in Java. The JDBC discussed in Volume 2 certainly makes using Java for client-server development easier. As third-party tools continue to be developed, we expect database development with Java to be as easy as the Net library makes network programming. Accessing remote objects is significantly easier in Java than in C++ (see Volume 2).

With Java, I can replace my computer with a $500 "Internet appliance."

Some people are betting big that this is going to happen. We believe it is pretty absurd to think that people are going to give up a powerful and convenient desktop for a limited machine with no local storage. However, if your Java-powered computer has enough local storage for the inevitable situation when your local intranet goes down, a Java-powered network computer is a viable option for a "zero administration initiative" to cut the costs of computer ownership in a business.

We also see an Internet appliance as a portable *adjunct* to a desktop. Provided the price is right, wouldn't you rather have an Internet-enabled *device* with a screen on which to read your e-mail or see the news? Because the Java kernel is so small, Java is the obvious choice for such a telephone or other Internet "appliance."

Java will allow the component-based model of computing to take off.

No two people mean the same thing when they talk about components. Regarding visual controls, like ActiveX components that can be dropped into a graphical user interface (GUI) program, Java 1.1 includes the JavaBeans initiative (see Volume 2). Java beans can potentially do the same sorts of things as ActiveX components *except* they are *automatically* cross-platform and *automatically* come with a security manager. All this shows that the success of the JavaBeans initiative will go far toward making Java successful.