Chapter 1

An Introduction to Java

- Java As a Programming Platform
- The Java "White Paper" Buzzwords
- Java Applets and the Internet
- A Short History of Java
- Common Misconceptions about Java

The first release of Java in 1996 generated an incredible amount of excitement, not just in the computer press, but in mainstream media such as *The New York Times, The Washington Post,* and *Business Week*. Java has the distinction of being the first and only programming language that had a ten-minute story on National Public Radio. A $100,000,000 venture capital fund was set up solely for products produced by use of a *specific* computer language. It is rather amusing to revisit those heady times, and we give you a brief history of Java in this chapter.

Java As a Programming Platform

In the first edition of this book, we had this to write about Java:

"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."

Our editor got a lot of flack for this paragraph from someone very high up at Sun Microsystems who shall remain unnamed. But, in hindsight, our prognosis seems accurate. Java has a lot of nice language features—we examine them in detail later in this chapter. It has its share of warts, and newer additions to the language are not as elegant as the original ones because of the ugly reality of compatibility.

But, as we already said in the first edition, Java was never just a language. There are lots of programming languages out there, and few of them make much of a splash. Java is a whole *platform*, with a huge library, containing lots of reusable code, and an execution environment that provides services such as security, portability across operating systems, and automatic garbage collection.

As a programmer, you will want a language with a pleasant syntax and comprehensible semantics (i.e., not C++). Java fits the bill, as do dozens of other fine languages. Some languages give you portability, garbage collection, and the like, but they don't have much of a library, forcing you to roll your own if you want fancy graphics or networking or database access. Well, Java has everything—a good language, a high-quality execution environment, and a vast library. That combination is what makes Java an irresistible proposition to so many programmers.

The Java "White Paper" Buzzwords

The authors of Java have written an influential White Paper that explains their design goals and accomplishments. They also published a shorter summary that is organized along the following 11 buzzwords:

Simple	Portable
Object Oriented	Interpreted
Network-Savvy	High Performance
Robust	Multithreaded
Secure	Dynamic
Architecture Neutral	

In this section, we will

- Summarize, with excerpts from the White Paper, what the Java designers say about each buzzword; and
- Tell you what we think of each 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/docs/white/langenv/`. The summary with the 11 buzzwords is at `http://java.sun.com/docs/overviews/java/java-overview-1.html`.

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, attempt to fix all of the clumsy features of C++. For example, the syntax of the `switch` statement is unchanged in Java. If you know C++, you will find the transition to the Java syntax easy.

If you are used to a visual programming environment (such as Visual Basic), you will not find Java simple. There is much strange syntax (though it does not take long to get the hang of it). More important, you must do a lot more programming in Java. The beauty of Visual Basic is that its visual design environment almost automatically provides a lot of the infrastructure for an application. The equivalent functionality must be programmed manually, usually with a fair bit of code, in Java. There are, however, third-party development environments that provide "drag-and-drop"-style program development.

> *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 was a great achievement at the time. Of course, the library has since grown to huge proportions. There is now a separate Java Micro Edition with a smaller library, suitable for embedded devices.

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 those of C++. The major difference between Java and C++ lies in multiple inheritance, which Java has replaced with the simpler concept of interfaces, and in the Java metaclass model (which we discuss in Chapter 5).

NOTE: If you have no experience with object-oriented programming languages, you will want to carefully read Chapters 4 through 6. These chapters explain what object-oriented programming is and why it is more useful for programming sophisticated projects than are traditional, procedure-oriented languages like C or Basic.

Network-Savvy

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 connection. (We cover networking in Volume II of this book.) The remote method invocation mechanism enables communication between distributed objects (also covered in Volume II).

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 (runtime) 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 detects many problems that, in other languages, would show up only at runtime. 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 Visual Basic 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, and even files. In Visual Basic, 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, many data structures are difficult to implement in a pointerless language. 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, because 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. Not long after the first version of the Java Development Kit was shipped, a group of security experts at Princeton University found subtle bugs in the security features of Java 1.0. Sun Microsystems has encouraged research into Java security, making publicly available the specification and implementation of the virtual machine and the security libraries. They have fixed all known security bugs quickly. In any case, Java makes it extremely difficult to outwit its security mechanisms. The bugs found so far have been very technical and few in number.

From the beginning, Java was designed to make certain kinds of attacks impossible, among them:

- Overrunning the runtime stack—a common attack of worms and viruses
- Corrupting memory outside its own process space
- Reading or writing files without permission

A number of security features have been added to Java over time. Since version 1.1, Java has the notion of digitally signed classes (see Volume II). With a signed class, you can be sure who wrote it. 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 create damage. Java has a far stronger security model than that of ActiveX because it controls the application as it runs and stops it from wreaking havoc.

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 runtime 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 30 years ago, both Niklaus Wirth's original implementation of Pascal and the UCSD Pascal system used the same technique.

Of course, interpreting bytecodes is necessarily slower than running machine instructions at full speed, so it isn't clear that this is even a good idea. However, virtual machines have the option of translating the most frequently executed bytecode sequences into machine code, a process called just-in-time compilation. This strategy has proven so effective that even Microsoft's .NET platform relies on a virtual machine.

The virtual machine has other advantages. It increases security because the virtual machine can check the behavior of instruction sequences. Some programs even produce bytecodes on the fly, dynamically enhancing the capabilities of a running program.

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 for number types eliminates a major porting headache. Binary data is stored and transmitted in a fixed format, eliminating confusion about byte ordering. 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 ten 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 user interface 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.

Incremental linking has advantages, but its benefit for the development process is clearly overstated. Early Java development tools were, in fact, quite slow. Today, the bytecodes are translated into machine code by the just-in-time compiler.

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 runtime) into machine code for the particular CPU the application is running on.

In the early years of Java, many users disagreed with the statement that the performance was "more than adequate." Today, however, the just-in-time compilers have become so good that they are competitive with traditional compilers and, in some cases, even outperform them because they have more information available. For example, a just-in-time compiler can monitor which code is executed frequently and optimize just

that code for speed. A more sophisticated optimization is the elimination (or "inlining") of function calls. The just-in-time compiler knows which classes have been loaded. It can use inlining when, based upon the currently loaded collection of classes, a particular function is never overridden, and it can undo that optimization later if necessary.

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 can 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 or a thread library. 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 runtime type information is straightforward.*

This is an important feature in those situations in which 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 runtime 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 runtime, such as Java GUI builders, smart debuggers, pluggable components, and object databases.

NOTE: Shortly after the initial success of Java, Microsoft released a product called J++ with a programming language and virtual machine that was almost identical to Java. At this point, Microsoft is no longer supporting J++ and has instead introduced another language called C# that also has many similarities with Java but runs on a different virtual machine. There is even a J# for migrating J++ applications to the virtual machine used by C#. We do not cover J++, C#, or J# in this book.

Java Applets 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*. To use an applet, you only need a Java-enabled web browser, which will execute the bytecodes for you. You need not install any software. Because Sun licenses the Java source code and insists that there be no changes in the language and standard library, a Java applet should run on any browser that is advertised as Java-enabled. You get the latest version of the program whenever you visit the web page containing the applet.

Most important, thanks to the security of the virtual machine, you need never worry about attacks from hostile code.

When the user downloads an applet, it works much like embedding an image in a web page. 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 presenting the applet and the computer serving it.

Figure 1–1 shows a good example of a dynamic web page that carries out sophisticated calculations. The Jmol applet displays molecular structures. By using the mouse, you can rotate and zoom each molecule to better understand its structure. This kind of direct manipulation is not achievable with static web pages, but applets make it possible. (You can find this applet at `http://jmol.sourceforge.net`.)

When applets first appeared, they created a huge amount of excitement. Many people believe that the lure of applets was responsible for the astonishing popularity of Java. However, the initial excitement soon turned into frustration. Various versions of Netscape and Internet Explorer ran different versions of Java, some of which were seriously outdated. This sorry situation made it increasingly difficult to develop applets that took advantage of the most current Java version. Today, most web pages simply use JavaScript or Flash when dynamic effects are desired in the browser. Java, on the other hand, has become the most popular language for developing the server-side applications that produce web pages and carry out the backend logic.

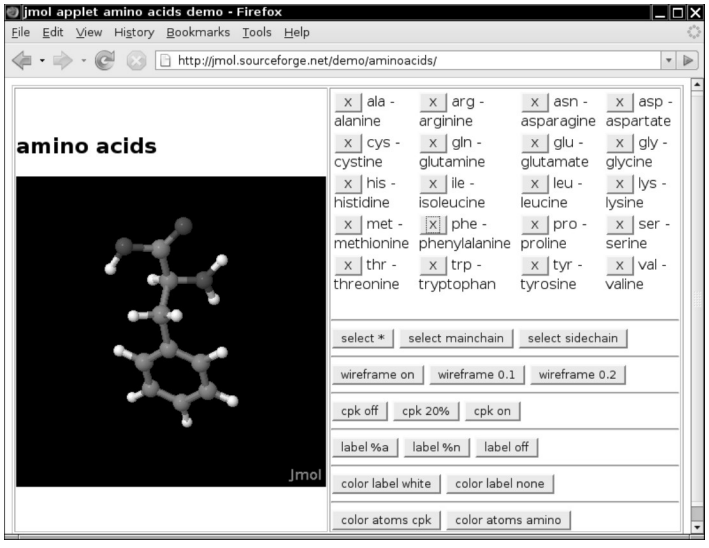

Figure 1–1 The Jmol applet

A Short History of Java

This section gives a short history of Java's evolution. It is based on various published sources (most important, 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. Because 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 that the language not be tied to any single architecture. The project was code-named "Green."

The requirements for small, tight, and platform-neutral code led the team to resurrect the model that some Pascal implementations tried in the early days of PCs. Niklaus Wirth, the inventor of Pascal, had pioneered the design of 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. The Green project engineers used a virtual machine as well, 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. This turned out to be an inspired choice.

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 inches by 4 inches by 4 inches.) 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 the *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. 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 executing code inside web pages. This "proof of technology" was shown at SunWorld '95 on May 23, 1995, and inspired the Java craze that continues today.

Sun released the first version of Java in early 1996. People quickly realized that Java 1.0 was not going to cut it for serious application development. Sure, you could use Java 1.0 to make a nervous text applet that moved text randomly around in a canvas. But you couldn't even *print* in Java 1.0. To be blunt, Java 1.0 was not ready for prime time. Its successor, version 1.1, filled in the most obvious gaps, greatly improved the reflection capability, and added a new event model for GUI programming. It was still rather limited, though.

The big news of the 1998 JavaOne conference was the upcoming release of Java 1.2, which replaced the early toylike GUI and graphics toolkits with sophisticated and scalable versions that come a lot closer to the promise of "Write Once, Run Anywhere"™ than its predecessors. Three days after (!) its release in December 1998, Sun's marketing department changed the name to the catchy *Java 2 Standard Edition Software Development Kit Version 1.2*.

Besides the Standard Edition, two other editions were introduced: the Micro Edition for embedded devices such as cell phones, and the Enterprise Edition for server-side processing. This book focuses on the Standard Edition.

Versions 1.3 and 1.4 of the Standard Edition are incremental improvements over the initial Java 2 release, with an ever-growing standard library, increased performance, and, of course, quite a few bug fixes. During this time, much of the initial hype about Java applets and client-side applications abated, but Java became the platform of choice for server-side applications.

Version 5.0 is the first release since version 1.1 that updates the Java *language* in significant ways. (This version was originally numbered 1.5, but the version number jumped to 5.0 at the 2004 JavaOne conference.) After many years of research, generic types (which are roughly comparable to C++ templates) have been added—the challenge was to add this feature without requiring changes in the virtual machine. Several other useful language features were inspired by C#: a "for each" loop, autoboxing, and metadata. Language changes are always a source of compatibility pain, but several of these new language features are so seductive that we think that programmers will embrace them eagerly.

Version 6 (without the .0 suffix) was released at the end of 2006. Again, there are no language changes but additional performance improvements and library enhancements.

Table 1–1 shows the evolution of the Java language and library. As you can see, the size of the application programming interface (API) has grown tremendously.

Table 1–1 Evolution of the Java Language

Version	Year	New Language Features	Number of Classes and Interfaces
1.0	1996	The language itself	211
1.1	1997	Inner classes	477
1.2	1998	None	1,524
1.3	2000	None	1,840
1.4	2004	Assertions	2,723
5.0	2004	Generic classes, "for each" loop, varargs, autoboxing, metadata, enumerations, static import	3,279
6	2006	None	3,777

Common Misconceptions about Java

We close this chapter with a list of some common misconceptions about Java, along with commentary.

Java is an extension of HTML.

Java is a programming language; HTML is 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.

I use XML, so I don't need Java.

Java is a programming language; XML is a way to describe data. You can process XML data with any programming language, but the Java API contains excellent support for XML processing. In addition, many important third-party XML tools are implemented in Java. See Volume II for more information.

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 *libraries.* The Java libraries contain thousands of classes and interfaces, and tens of thousands of functions. Luckily, you do not need to know every one of them, but you do need to know surprisingly many to use Java for anything realistic.

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, many applications, already working perfectly well on desktops, 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 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.

Java is just another programming language.

Java is a nice programming language; most programmers prefer it over C, 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 Visual Basic, have been wildly successful.

Why? The success of a programming language is determined far more by the utility of the *support system* surrounding it than by the elegance of its syntax. Are there useful, convenient, and standard libraries for the features that you need to implement? Are there tool vendors that build great programming and debugging environments? Does the language and the toolset integrate with the rest of the computing infrastructure? Java is successful because its class libraries let 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 factors are not the source of its success.

Now that C# is available, Java is obsolete.

C# took many good ideas from Java, such as a clean programming language, a virtual machine, and garbage collection. But for whatever reasons, C# also left some good stuff behind, in particular security and platform independence. If you are tied to Windows, C# makes a lot of sense. But judging by the job ads, Java is still the language of choice for a majority of developers.

Java is proprietary, and it should therefore be avoided.

Sun Microsystems licenses Java to distributors and end users. Although Sun has ultimate control over Java through the "Java Community Process," they have involved many other companies in the development of language revisions and the design of new libraries. Source code for the virtual machine and the libraries has always been freely available, but only for inspection, not for modification and redistribution. Up to this point, Java has been "closed source, but playing nice."

This situation changed dramatically in 2007, when Sun announced that future versions of Java will be available under the General Public License, the same open source license that is used by Linux. It remains to be seen how Sun will manage the governance of Java in the future, but there is no question that the open sourcing of Java has been a very courageous move that will extend the life of Java by many years.

Java is interpreted, so it is too slow for serious applications.

In the early days of Java, the language was interpreted. Nowadays, except on "micro" platforms such as cell phones, the Java virtual machine uses a just-in-time compiler. The

"hot spots" of your code will run just as fast in Java as they would in C++, and in some cases, they will run faster.

Java does have some additional overhead over C++. Virtual machine startup time is slow, and Java GUIs are slower than their native counterparts because they are painted in a platform-independent manner.

People have complained for years that Java applications are too slow. However, today's computers are much faster than they were when these complaints started. A slow Java program will still run quite a bit better than those blazingly fast C++ programs did a few years ago. At this point, these complaints sound like sour grapes, and some detractors have instead started to complain that Java user interfaces are ugly rather than slow.

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 most Java programs are stand-alone applications that run outside of a web browser. In fact, many Java programs run on web servers and produce the code for web pages.

Most of the programs in this book are stand-alone programs. Sure, applets can be fun. But stand-alone Java programs are more important and more useful in practice.

Java programs are a major security risk.

In the early days of Java, there were some well-publicized reports of failures in the Java security system. Most failures were 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 surprisingly little criticism of the weaknesses of the attacked platform. Also, the ActiveX mechanism in Internet Explorer would be a fertile ground for abuse, but it is so boringly obvious how to circumvent it that few researchers 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 hostile Java applets is perhaps comparable to the risk of dying from a plane crash; the risk of being infected by opening Word documents 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. JavaScript is more tightly integrated with browsers than Java applets are. In particular, a JavaScript program can modify the document that is being displayed, whereas an applet can only control the appearance of a limited area.

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

When Java was first released, some people bet big that this was going to happen. Ever since the first edition of this book, we have believed it is absurd to think that home users are going to give up a powerful and convenient desktop for a limited machine with no local storage. We found the Java-powered network computer a plausible option for a "zero administration initiative" to cut the costs of computer ownership in a business, but even that has not happened in a big way.

On the other hand, Java has become widely distributed on cell phones. We must confess that we haven't yet seen a must-have Java application running on cell phones, but the usual fare of games and screen savers seems to be selling well in many markets.

TIP: For answers to common Java questions, turn to one of the Java FAQ (frequently asked question) lists on the Web—see `http://www.apl.jhu.edu/~hall/java/FAQs-and-Tutorials.html`.