Part 1

Foundations of Java

In this section:

Chapter 1

Introducing Java

ONCE UPON A TIME there was an island in Indonesia known by the name of Java. The people on the island lived a peaceful existence, going about their business and their daily routines for thousands of years. One day in the early 1990s, a company by the name of Sun Microsystems created a new programming language called Oak. Since Oak was internal to Sun, the name didn't matter much, and the people on Java continued to go on their merry way, ignorant of what was in store for them.

A few years later, Sun was ready to make Oak a real product. Unfortunately, Oak's name didn't pass the trademark test. Some other choices, like Silk, Ruby, and WRL (for WebRunner Language), didn't cut the mustard, either, but eventually the name Java won out and made its way through a legal review. (Apparently, having conflicting names for a programming language and an island doesn't matter. What can you expect from a company named after a star?) Then on May 23, 1995, Java was launched at Sun's annual SunWorld conference, as the 12th of about 20 different announcements. Little did those Indonesians know what was going to happen on their little island.

Java, the platform, began to evolve, and Microsoft thought it important enough to incorporate into their Internet Explorer browser. Back on the island of Java, the volcanoes were getting restless. Then in 1997, Sun sued Microsoft over some Java incompatibilities, and Krakatoa just blew. (Okay, so Java has about 10 major volcanic eruptions each year; it may have just been a coincidence.) As the lawsuit progressed further and further, life on the island got bad. Economic unrest in Jakarta created calls for President Suharto to step down, and East Timor demanded political autonomy. Near bloody battles ensued on the Sun Java front, with an eventual $20 million dollar settlement with Microsoft in 2001. Coincidence? Just ask then President Wahid what *he* has to say.

Java History 101

Okay, enough of the convoluted history of Java the island and Java the platform. And yes, that *is* Java, *the platform.* What began in 1995 as just another programming language is now formally known as a platform. Beginning as a platform-neutral, Internet-friendly development language, Java has evolved into a means of creating programs for just about anything. Let's step back to the beginning and see how it came about.

Back in January 1991, a project by the name of Green began. The Green project's purpose was to come up with a way of controlling set-top boxes, those smart cable TV access boxes. Because the hardware in the consumer devices was always changing, it became apparent to James Gosling, the "father" of Java, that C++ wasn't appropriate for the job. As a result, the language called Oak was created; it was less susceptible to bugs and wouldn't crash the whole system. It initially ran on a Hammer technology device called *7 (Star7). Cavorting across the touch screen was this little digital character named Duke, a helper agent, who would go off and do various tasks. (He later turned into the Java mascot.)

NOTE *If you've never seen Duke before, he looks like a cross between a Star Trek communicator and an upside-down tooth with a witch's hat, with a big red nose and Screen Bean–style arms. (Screen Beans are those clip art characters you frequently see running all over the place in PowerPoint presentations.)*

Oak was used to bid on a project for a TV set–top operating system. The bid failed, and Sun refused a purchase offer from Trip Hawkins (then CEO of Gamemaker/3DO), so Sun had all this leftover stuff it didn't know what to do with. Thankfully, the Internet revolution had begun with the release of Mosaic, the predecessor to the Netscape Navigator Web browser. Oak was repurposed in 1994 for the Web, and a browser called WebRunner (later renamed HotJava) was created to show off the newfound technology of what eventually became Java.

By now it was time to take Java public. At the now infamous SunWorld conference, Sun's John Gage and Netscape's Marc Andreesen announced Java to the world. Netscape committed to incorporate Java into the next release of their browser, and by the end of the year companies such as IBM, Oracle, Borland, Adobe, Macromedia, Lotus, Spyglass, and Intuit had incorporated Java into their products. Even Microsoft licensed Java, committing themselves to incorporating Java into their products, operating systems, and development tools.

This initial version of Java was so small it fit on a floppy. The core interpreter was about 100KB. The math library added 20KB. The code to support integration with C libraries was just 50KB. The majority of the class libraries fit in 375KB, leaving just the platform-specific graphics libraries that varied in size. Overall, Java was just around a megabyte and could run in a machine with 1MB of ROM and 1MB of RAM. (*7 had 4MB, though.)

Jump ahead six years, and Java has evolved considerably. No longer confined to run in the browser, you'll find Java programs everywhere—from Web servers, to jewelry with flashing lights, to smarter toasters, and even in the originally planned set-top boxes. The core libraries have grown considerably, too, with the latest release (Java 2 Standard Edition, version 1.4) occupying 30 to 40MB just for its runtime environment. Sure, Java can do more now, and that is essentially why you have the Java Platform, which is the combination of the programming language, the standard libraries, and the runtime environment. And, in this book, it's that platform you'll learn about.

Examining the Java Architecture

According to the original Java white paper (available at `http://java.sun.com/people/jag/OriginalJavaWhitepaper.pdf`), the Java design goals were to be "a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded, dynamic language." By looking into what these buzzwords represent, we can see what Java was trying to prove then and what still applies now.

Simple

At the first C++ conference, Bill Joy, cofounder and now chief scientist and CEO of Sun, said that "C++ is too complicated, what I want is C++ ++ −−." In Java, that is what he and everyone else gets. Java incorporates new tasks such as automatic garbage collection (the ++) and removes confusing and rarely used aspects of C++ such as operator overloading (the −−).

Another aspect of Java's simplicity is that nothing in Java is really new. If you look through the feature set of Java alongside the history of computing, you'll find that everything came from somewhere else.

- Classes came from C++ and Smalltalk, though they're limited in Java to single implementation inheritance.
- Interfaces came from Objective-C, providing Java with multiple interface inheritance.
- Packages originated in Modula, adding to Java a hierarchical name space and logical development units.
- Concurrency was in Mesa, and in Java brings built-in multithreading support.
- Exception handling was from Modula-3, adding to Java methods that are declarative of what they throw.
- Dynamic linking and automatic garbage collection come from Lisp, offering to Java the ability to load additional classes as needed and to free memory when it's not needed.

And as new features get added into Java, they too will derive from tried and true practices. For instance, still missing from the 1.4 release is *parameterized type support,* also known as *templates* or *generics.* Once added, this feature will add support for type-safe collections, a feature that has been requested for some time. However, the powers that be don't want to add in just anything; they're trying to make sure that anything added into Java proper is done correctly. Expect to find generics support in Java 1.5, assuming everything is decided by then.

Object-Oriented

People call Java *object-oriented,* but what exactly does that mean? Essentially, object-oriented programming (OOP) is a way of developing software by describing problems through the use of elements or objects from the problem space, rather than a sequential set of steps to execute in the computer. Good design then leads to reusable, extensible, and maintainable components. These components are flexible enough to handle changes in the environment over time, since the primary job of these objects is just to send messages back and forth to each other.

Essentially, the Core Java APIs are a collection of these prebuilt components, called *class libraries*. Instead of reinventing the wheel with Java, you get to start with the standard libraries. These standard libraries have evolved over time, gaining more prebuilt components with each Java release.

For more depth on object-oriented programming, see Chapter 3.

Distributed

Since its inception in 1982, Sun's vision has been "The network is the computer." With Java, they wanted a programming language that was network savvy and that supported access to distributed objects as easily as local ones. This goal, while present from the beginning, has changed as far as Java's capabilities are concerned.

Initially, all Java could do was access distributed objects through standard TCP/IP-based protocols such as HTTP. With the latest release of the Java Platform, you can invoke methods as easily and invisibly on a remote machine as you can in your same execution space, through such common protocols as CORBA (Common Object Request Broker Architecture) and RMI (Remote Method Invocation), as well as the recently added Web Services. For each protocol, the system deals with all the translation and transport automatically for you.

Interpreted

Java programs are *interpreted.* Instead of getting compiled into native executables, Java code gets translated into platform-neutral byte codes. These byte codes can then be transferred onto any platform having a Java Runtime Environment (JRE), which consists of a Java Virtual Machine (JVM), and thus be executed without recompiling or re-linking. Such a process may suggest that Java is inherently slow, but—as explained in the upcoming "Performance" section—that isn't necessarily the case. The platform-neutral byte codes actually contain additional information that can be used to optimize execution at runtime, based on decisions that are impossible to make at compile time.

Robust

Robustness is a measure of a program's reliability. Java has several built-in features that improve the reliability of a program:

- Java is a strongly typed language. The compiler and class loader ensure the correctness of all method calls, preventing implicit type mismatches and versioning incompatibilities.
- Java has no pointers. You cannot reference a pointer into memory, accidentally corrupting memory or walking past the end of an array. In Java, there are references instead of pointers, and they cannot be dereferenced to directly manipulate the memory space. (The exception is video memory, which can be more directly controlled starting with Java 1.4.)
- Java performs automatic garbage collection. Programmers can't accidentally forget to free up memory and don't have to worry about figuring out where memory must be released.
- Java encourages the use of interfaces rather than classes. As you'll learn in Chapter 5, interfaces define a set of behaviors, and classes implement those behaviors. Instead of passing around classes, you pass around interfaces, thus hiding the implementations. If the implementation changes, as usually happens, then as long as the new class implements the old interface, everything else will still work fine.

Secure

Java has many built-in security features, adding even more standard ones with the 1.4 release. These features are at work in several different areas. Many of the things that make Java robust also ensure the security of the executing program—for instance, no memory pointers, so no corrupting memory. Other security features are built into the execution model: the byte code verifier, class loader, and security manager. These ensure the safety of the executing code, and prevent untrusted code from performing trusted operations such as reading your hard drive. The final aspect of security has to do with authentication, authorization, and encryption, for protecting the privacy and ensuring the integrity of data.

The *byte code verifier* provides the first level of defenses on the security front. Essentially, it says "Don't trust the compiler." Before a class is loaded into the execution environment, its byte codes must be verified. If someone used a byte-level editor on the file and modified the bytes in an attempt to access something that shouldn't be, the verifier will detect this and prevent the loading of the class and execution of the program. In fact, programs that run in the JRE don't even have to be written with the Java programming language.

Next in the security category comes the *class loader.* As its name implies, the class loader is responsible for loading classes. On the security front, the class loader is responsible for loading classes and keeping them separated from untrustworthy peers. First, the class loader keeps local classes and network-loaded classes in different name spaces, ensuring that network-loaded classes cannot replace local system classes. Secondly, classes loaded from different network locations are also kept apart. This ensures that a nonsystem class doesn't pretend to be a system class; nor can it be friendly to a class loaded from a third-party Web site.

The third security component of the runtime environment is the *security manager,* which enforces the security policy for the runtime environment. All tasks requiring permission to do anything pass through the security manager. If the task can't get permission from the security manager, then the task doesn't execute. This is especially important for code that is loaded over the Internet from untrusted Web sites. Such code that executes in the browser is called applets. They run in an area of the browser dubbed the *sandbox* (a safe place to run but with no access to the client's environment).

Starting with Java 1.1 and up to and including Java 1.4, more security capabilities have been added to the Core Java API set. There is support for developers to sign Java classes with a digital signature, permitting users to trust their integrity enough to break out of the sandbox. There is support for generating digests for messages to ensure their integrity. You'll find support for certificate management, access control lists, SSL/TLS (Secure Sockets Layer/Transport Layer Security), and even an exportable encryption library.

All of these security-related features are present to ensure successful construction and execution of distributed, networked programs.

Architecture-Neutral

"Architectural neutrality" refers to Java's platform-neutral byte codes. Rather than being compiled into platform-specific binaries, Java programs are made to execute anywhere without recompilation or re-linking. If a company develops new hardware, it doesn't have to throw away its software investment, but only needs to place a JRE on the new platform. And, if a brand-new company develops completely new hardware or a new operating system, it doesn't have to start from ground zero with

no software on the product. With the addition of just the JRE, the newly designed platform can run all the Java programs out there.

In fact, this is exactly what a company called SavaJe (`www.savaje.com`) did. SavaJe created a new operating system for the Compaq iPAQ and other 32-bit StrongARM-based information appliances. Instead of having to train developers to create applications for the new operating system, they were able to take existing Java-based applications and simply put them on the device, and they ran fine (though they probably had a smaller-than-expected screen).

Portable

You may or may not have heard the Java mantra: Write Once, Run Anywhere (WORA). Java's goal is to be able to take the same program and run it on any architecture. This *portability* isn't achieved by just the platform-neutral byte codes, though. Rather than relying on implementation-specific details, such as how big an `int` is, all numerical representations in size, byte order, and manipulation are defined in the *Java Language Specification* (`http://java.sun.com/docs/books/jls/`).

High-Performance

You might think that including the words *interpreted* and *high-performance* in the same sentence is an oxymoron. However, the platform-neutral byte codes can in fact be translated at runtime to CPU-specific machine code, executing nearly as fast if not faster than natively compiled C and C++ code. Two such runtime translation tools included with Java do this for you automatically: The first-generation tool is called a Just-in-Time (JIT) compiler. Sun's second-generation tool is called HotSpot. Essentially, HotSpot and the JIT compiler do the same thing: runtime translation to the native instruction set. HotSpot, however, also monitors the code as it executes and performs speed optimizations on the frequently executed blocks, instead of on everything.

NOTE *For a comparison of various Java runtime platforms, be sure to examine the Volano Report (*`www.volano.com/report/`*).*

Multithreaded

You can think of a *thread* as an execution context. For example, when surfing online, your browser will display a new Web page while continuing to download images in the background. Each of these tasks can be done in a separate thread. When your program runs multiple tasks or threads simultaneously, your program is said to be multithreaded.

Multithreaded programs share memory and must communicate among the threads. The Java programming language and standard libraries include many facilities to assist you with this communication process, ensuring thread safety. You'll learn more about creating thread-safe, multithreaded programs in Chapter 8.

Dynamic Language

The last of the Java characteristic buzzwords says that Java is a *dynamic* language. This doesn't mean that the language is constantly evolving; the phrase has more to do with the libraries. Specifically, it means that as libraries change over time, programs do not need to be re-linked. The earlier,

platform-neutral byte codes will continue to work after the new libraries are released, either because of a new Java version from Sun or a new third-party library release from some other company. As long as the used parts of the library aren't dropped, the programs will still work, even if new pieces get added to the libraries. This dynamic nature is also true of Java's preference of interfaces over classes, as mentioned in the discussion of robustness.

Understanding Microsoft's View

A book on Java seems to require some statement about where Java fits into Microsoft's plans. Basically, beyond a duplicitous attempt to move Visual J++ developers forward, it doesn't. The *Sun v. Microsoft* lawsuit was settled in January 2001. As part of the settlement, Microsoft can continue to ship Microsoft's latest Java Runtime Environment, if it so chooses. Three issues have arisen: First, the latest environment supplied by Microsoft is based on Java 1.1.4—yes, three releases ago, and it's even older if you count the fact that Sun's latest 1.1 release is 1.1.8. Second, as of Windows XP, Microsoft's Java runtime does not ship with the operating system anymore. Third, Microsoft offers Visual J#.NET, a Java 1.1.4–based tool that supports the Java programming language but not the platform, supporting Microsoft's .NET architecture instead.

Does it matter? Technically speaking, it depends. Sure, it was nice to have *any* runtime environment on every client machine. However, using an old *and* incompatible version of the JRE caused many problems and headaches. Sun will continue to provide the latest runtime environment for Windows platforms for free. Users just have to download it separately, unless Sun can convince computer hardware vendors to preinstall it. (For Compaq and some other hardware vendors, this has already been announced; Compaq will ship Sun's runtime with all XP boxes.) Microsoft's business decision was to simply abandon support of standard Java on the Windows platform. If you liked their Visual J++ extensions and want tight Windows integration, you're better off moving to C# than jumping on their latest Java derivative.

NOTE *C#, pronounced "C-sharp," is Anders Hejlsberg's creation for working with Microsoft's .NET platform.*

As far as Java on the server goes, the lack of a Microsoft runtime is a nonissue. Application servers tend to come with their own runtime environment, and it is easy to upgrade if you don't like the one they provide.

Moving On...

In the summer of 2001, DevX published an analysis on the state of Java after six years of existence (`www.devx.com/judgingjava/articles/sixyears`). As you travel on the path toward the realm of productive Java developer, this article should prove useful. You can read summaries of some of DevX's findings, based on the polling of over 2,600 Java developers.

First off, tools are an important component of Java development. However, nobody is really completely happy with what is out there. In nearly all cases, less than 50 percent of developers were overly satisfied with any of the tools. If you like to follow the crowd, however, the six most frequently used tools with high satisfaction as of the time of the article were JBuilder from Borland, WebLogic from BEA, VisualAge for Java from IBM, WebGain Studio/Visual Café from WebGain, CodeWarrior

from MetroWorks, and Forte for Java from Sun. Beyond these, there's a definite gap in the market share of the remaining satisfactory tools.

The second interesting statistic from DevX's report is about technology/language trends. Of all applications built, DevX says 43 percent were reportedly built using Java technologies. Visual Basic came in second with 25 percent, and C++ around 22 percent. (C# came in at 2.8 percent for planned future development projects, but none current.) I'd like to think Java's 43 percent is a number that should encourage people to learn Java and drop Visual Basic or C++, but I really have to think hard about the value of this number given the statistical sample. On first thought, the developers polled were already Java developers. Their company at least had already decided to take the plunge. This means the companies with no Java development projects probably had no survey respondents. In comparison, though, Bloor Interactive reported in February 2000 that "Java overtakes C++ as most in-demand programming skill." In that report, they looked at want ads. Java and C++ ran neck and neck around 36 percent of about 350,000 Internet job ads gathered over a 12-month period, and VB was about 10 percent lower. So, given over a year's difference between the two reports, the DevX's 43 percent doesn't really seem that far out of whack.

The DevX report concludes with a list of areas with demand for skills and what the skills gap is. As you learn more about Java, consider acquiring a deeper knowledge in the hungrier areas of Java technology: data access, middleware, XML, and server-side scripting.