Chapter One

Introduction to Java's Architecture

At the heart of Java technology lies the Java Virtual Machine--the abstract computer on which all Java programs run. Although the name "Java" is generally used to refer to the Java programming language, there is more to Java than the language. The Java Virtual Machine, Java API, and Java class file work together with the Java language to make the Java phenomenon possible.

The first four chapters of this book (Part I. "Java's Architecture") show how the Java Virtual Machine fits into the big picture. They show how the virtual machine relates to the other components of Javaís architecture: the class file, API, and language. They describe the motivation behind--and the implications of--the overall design of Java technology.

This chapter gives an introduction to Java as a technology. It gives an overview of Java's architecture, discusses why Java is important, and looks at Javaís pros and cons.

Why Java?

Over the ages people have used tools to help them accomplish tasks, but lately their tools have been getting smarter and interconnected. Microprocessors have appeared inside many commonly used items, and increasingly, they have been connected to networks. As the heart of personal computers and workstations, for example, microprocessors have been routinely connected to networks. They have also appeared inside devices with more specific functionality than the personal computer or the workstation. Televisions, VCRs, audio components, fax machines, scanners, printers, cell phones, personal digital assistants, pagers, and wrist-watches--all have been enhanced with microprocessors; most have been connected to networks.

Given the increasing capabilities and decreasing costs of information processing and data networking technologies, the network is rapidly extending its reach. The emerging infrastructure of smart devices and computers interconnected by networks represents a new environment for software--an environment that presents new challenges and offers new opportunities to software developers.

Java technology is a tool well suited to help you meet the challenges and seize the opportunities presented by the emerging computing environment. Java was designed for networks. Its suitability for networked environments is inherent in its architecture, which enables secure, robust, platform-independent programs to be delivered across networks and run on a great variety of computers and devices.

The Challenges and Opportunities of Networks

One challenge presented to developers by a networked computing environment is the wide range of devices that networks interconnect. A typical network usually has many different kinds of attached devices, with diverse hardware architectures, operating systems, and purposes. Java addresses this challenge by enabling the creation of platform-independent programs. A single Java program can run unchanged on a wide range of computers and devices. Compared with programs compiled for a specific hardware and operating system, platform-independent programs written in Java can be easier and cheaper to develop, administer, and maintain.

Another challenge the network presents to software developers is security. In addition to their potential for good, networks represent an avenue for malicious programmers to steal or destroy information, steal computing resources, or simply be a nuisance. Virus writers, for example, can place their wares on the network for unsuspecting users to download. Java addresses the security challenge by providing an environment in which programs downloaded across a network can be run with customizable degrees of security. A downloaded program can do anything it wants inside the boundaries of the secure environment, but canít read or write data outside those boundaries.

One aspect of security is simple program robustness. Java's architecture guarantees a certain level of program robustness by preventing certain types of pernicious bugs, such as memory corruption, from ever occurring in Java programs. This establishes trust that downloaded code will not inadvertently (or intentionally) crash, but it also has an important benefit unrelated to networks: it makes programmers more productive. Because Java prevents many types of bugs from ever occurring, Java programmers need not spend time trying to find and fix them.

One opportunity created by an omnipresent network is online software distribution. Java takes advantage of this opportunity by enabling the transmission of binary code in small pieces across networks. This capability can make Java programs easier and cheaper to deliver than programs that are not network-mobile. It can also simplify version control. Because the most recent version of a Java program can be delivered on-demand across a network, you neednít worry about what version your end-users are running. They will always get the most recent version each time they use your program.

Platform independence, security, and network-mobility--these three facets of Javaís architecture work together to make Java suitable for the emerging networked computing environment. Because Java programs are platform independent, network-delivery of software is more practical. The same version of a program can be delivered to all the computers and devices the network interconnects. Java's built-in security framework also helps make network-delivery of software more practical. By reducing risk, the security framework helps to build trust in a new paradigm of network-mobile code.

The Architecture

Javaís architecture arises out of four distinct but interrelated technologies, each of which is defined by a separate specification from Sun Microsystems:

- the Java programming language
- the Java class file format
- the Java Application Programming Interface
- the Java Virtual Machine

When you write and run a Java program, you are tapping the power of these four technologies. You express the program in source files written in the Java programming language, compile the source to Java class files, and run the class files on a Java Virtual Machine. When you write your program, you access system resources (such as I/O, for example) by calling methods in the classes that implement the Java Application Programming Interface, or Java API. As your program runs, it fulfills your programís Java API calls by invoking methods in class files that implement the Java API. You can see the relationship between these four parts in Figure 1-1.

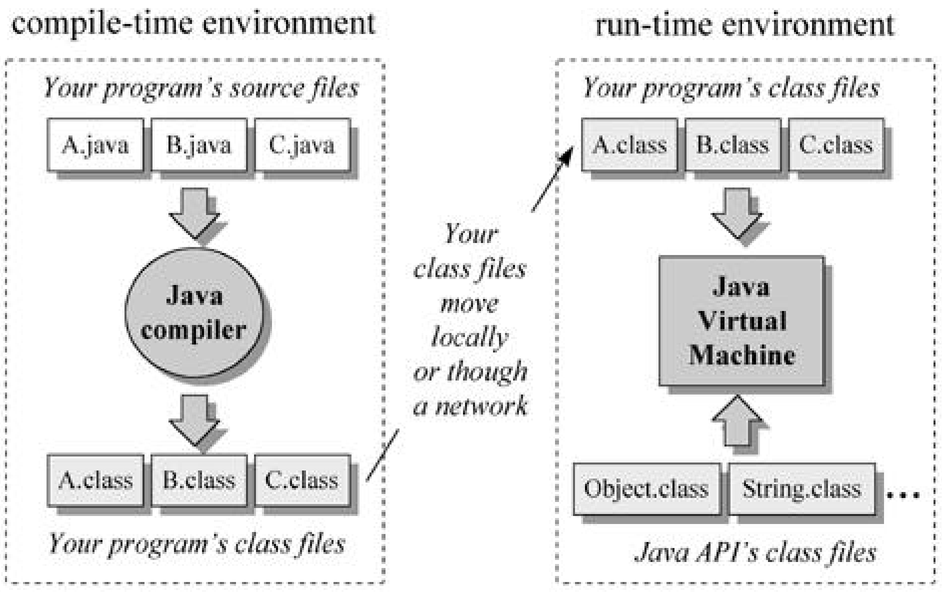

Figure 1-1. The Java programming environment.

Together, the Java Virtual Machine and Java API form a "platform" for which all Java programs are compiled. In addition to being called the *Java runtime system*, the combination of the Java Virtual Machine and Java API is called the *Java Platform*. Java programs can run on many different kinds of computers because the Java Platform can itself be implemented in software. As you can see in Figure 1-2, a Java program can run anywhere the Java Platform is present.

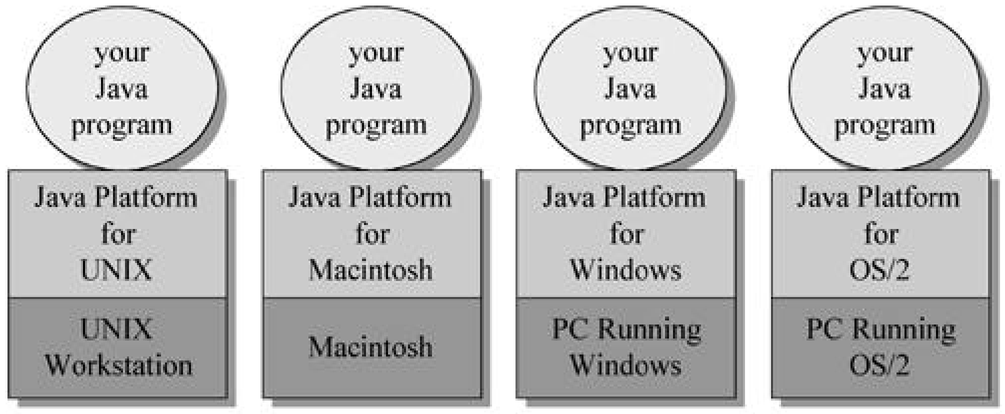

Figure 1-2. Java programs run on top of the Java Platform.

The Java Virtual Machine

At the heart of Java's network-orientation is the Java Virtual Machine, which supports all three prongs of Java's network-oriented architecture: platform independence, security, and network-mobility.

The Java Virtual Machine is an abstract computer. Its specification defines certain features every Java Virtual Machine must have, but leaves many choices to the designers of each implementation. For example, although all Java Virtual Machines must be able to execute Java bytecodes, they may use any technique to execute them. Also, the specification is flexible enough to allow a Java Virtual Machine to be implemented either completely in software or to varying degrees in hardware. The flexible nature of the Java Virtual Machine's specification enables it to be implemented on a wide variety of computers and devices.

A Java Virtual Machine's main job is to load class files and execute the bytecodes they contain. As you can see in Figure 1-3, the Java Virtual Machine contains a *class loader*, which loads class files from both the program and the Java API. Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine. The bytecodes are executed in an *execution engine*, which is one part of the virtual machine that can vary in different implementations. On a Java Virtual Machine implemented in software, the simplest kind of execution engine just interprets the bytecodes one at a time. Another kind of execution engine, one that is faster but requires more memory, is a *just-in-time compiler*. In this scheme, the bytecodes of a method are compiled to native machine code the first time the method is invoked. The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked. On a Java Virtual Machine built on top of a chip that executes Java bytecodes natively, the execution engine is actually embedded in the chip.

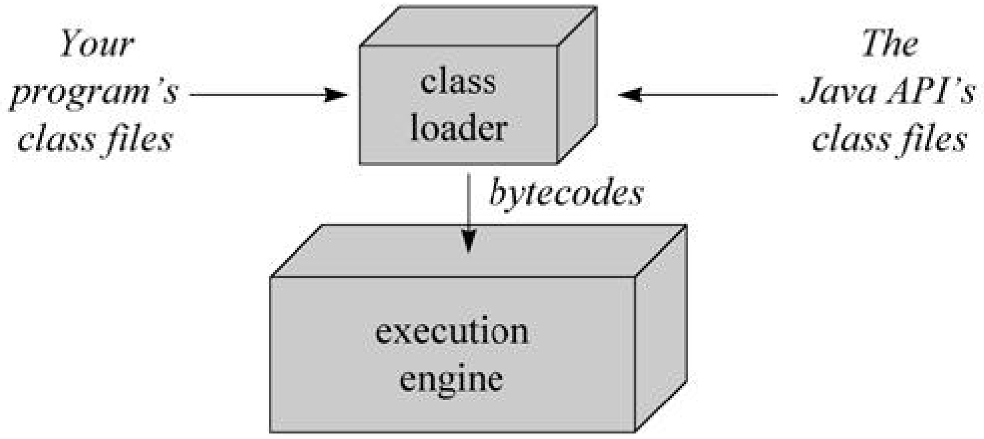

Figure 1-3. A basic block diagram of the Java Virtual Machine.

Sometimes the Java Virtual Machine is called the *Java interpreter*; however, given the various ways in which bytecodes can be executed, this term can be misleading. While "Java interpreter" is a reasonable name for a Java Virtual Machine that interprets bytecodes, virtual machines also use other techniques (such as just-in-time compiling) to execute bytecodes. Therefore, although all Java interpreters are Java Virtual Machines, not all Java Virtual Machines are Java interpreters.

When running on a Java Virtual Machine that is implemented in software on top of a host operating system, a Java program interacts with the host by invoking *native methods*. In Java, there are two kinds of methods: Java and native. A Java method is written in the Java language, compiled to bytecodes, and stored in class files. A native method is written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular processor. Native methods are stored in a dynamically linked library whose exact form is platform specific. While Java methods are platform independent, native methods are not. When a running Java program calls a native method, the virtual machine loads the dynamic library that contains the native method and invokes it. As you can see in Figure 1-4, native methods are the connection between a Java program and an underlying host operating system.

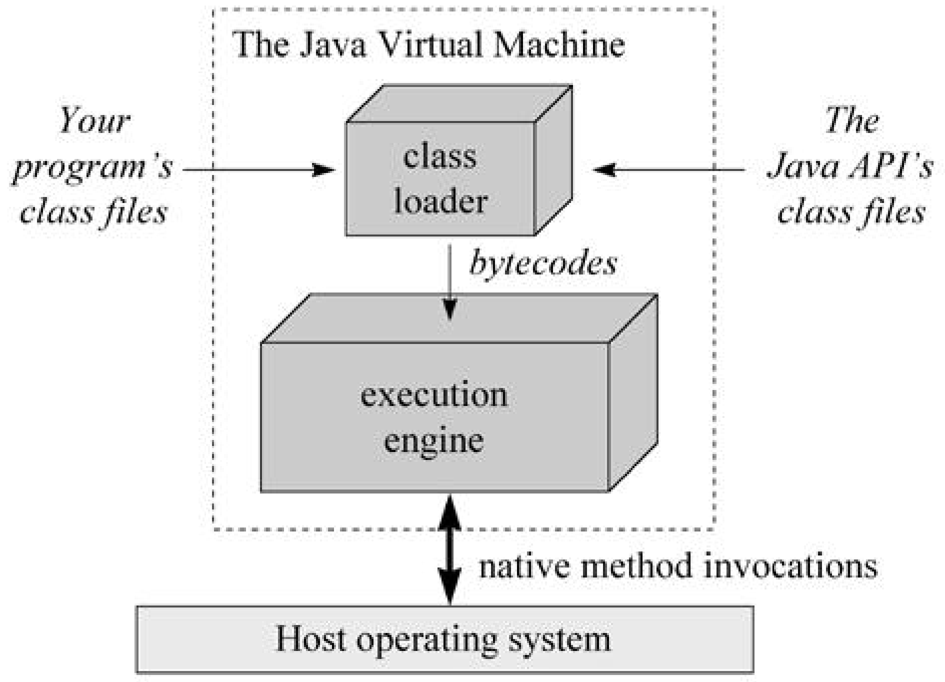

Figure 1-4. A Java Virtual Machine implemented in software on top of a host operating system.

You can use native methods to give your Java programs direct access to the resources of the underlying operating system. Their use, however, will render your program platform specific. This is because the dynamic libraries containing the native methods are platform specific. In addition, the use of native methods may render your program specific to a particular implementation of the Java Platform. One native method interface--the *Java Native Interface*, or *JNI*--enables native methods to work with any Java Platform implementation on a particular host computer. Vendors of the Java Platform, however, are not required to support JNI. They may provide their own proprietary native method interfaces in addition to (or in place of) JNI.

Java gives you a choice. If you want to access resources of a particular host that are unavailable through the Java API, you can write a platform-specific Java program that calls native methods. If you want to keep your program platform independent, however, you must call only Java methods and access the system resources of the underlying operating system through the Java API.