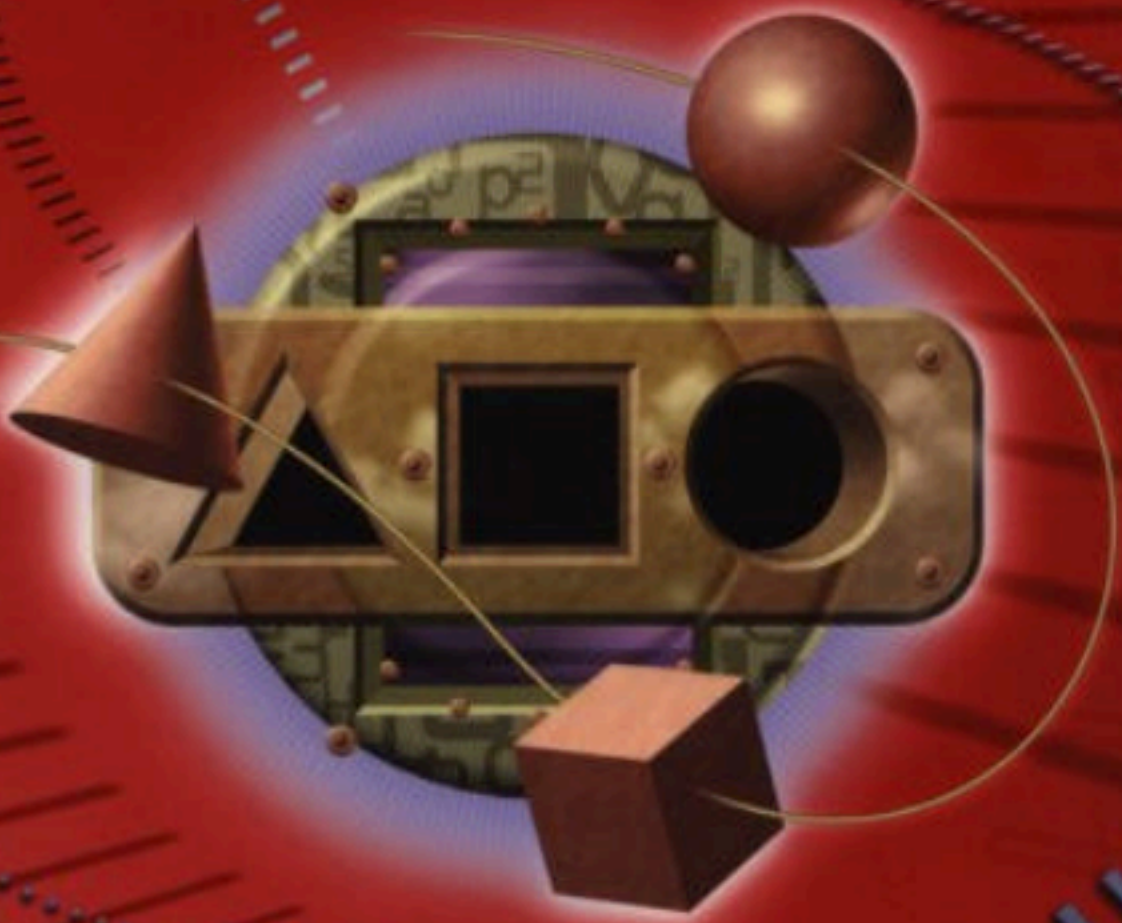
Inside the Java Virtual Machine
by
Bill Venners

Specially Made for [OR] by Outlawtorn

Application development

Inside the JAVA 2 Virtual Machine



Bill Venners

Inside the Java Virtual Machine

Acknowledgments

Introduction

Part One: Java's Architecture

Part Two: Java Internals

Preface

My primary goal in writing this book was to explain the Java Virtual Machine, and the software technologies upon which it is based, to Java programmers. Although the Java Virtual Machine incorporates technologies that had been tried and proven in other programming languages, prior to Java many of these technologies had not yet entered into common use. As a consequence, many programmers will be encountering these technologies for the first time as they begin to program in Java. Garbage collection, multi-threading, exception handling, dynamic extension--even the use of a virtual machine itself--may be new to many programmers. The aim of this book is to help programmers understand how all these things work, and in the process, to help them become more adept with the Java programming language.

Another goal I had in mind as I wrote this book was to experiment a bit with the changing nature of text. Web pages have three interesting characteristics that differentiate them from paper-based text: theyíre dynamic (they can evolve over time), theyíre interactive (especially if you embed Java applets in them), and theyíre interconnected (you can easily navigate from one to another). Besides the traditional text and figures, this book includes several Java applets (in a mini-website on the CD-ROM) that serve as "interactive illustrations" of the concepts presented in the text. In addition, I maintain a website on the internet that serves as a launching point for readers to find more (and more current) information on the topics covered in the book. This book is composed of all of these components: text, figures, interactive illustrations, and constantly evolving links to further reading.

Bill Venners

Sunnyvale, California

August, 1997

Introduction

This book describes the Java Virtual Machine, the abstract computer on which all Java programs run. Through a combination of tutorial explanations, working examples, reference material, and applets that interactively illustrate the concepts presented in the text, this book provides an in-depth, technical survey of Java as a technology.

The Java programming language seems poised to be the next popular language for mainstream commercial software development, the next step after C and C++. One of the fundamental reasons Java is a likely candidate for this role is that Javaís architecture helps programmers deal with emerging hardware realities. Java has features that the shifting hardware environment is demanding, features that are made possible by the Java Virtual Machine.

The evolution of programming languages has to a great extent been driven by changes in the hardware being programmed. As hardware has grown faster, cheaper, and more powerful, software has become larger and more complex. The migration from assembly languages to procedural languages, such as C, and to object oriented languages, such as C++, was largely driven by a need to manage ever greater complexity--complexity made possible by increasingly powerful hardware.

Today the progression towards cheaper, faster, and more powerful hardware continues, as does the need for managing increasing software complexity. Building on C and C++, Java helps programmers deal with complexity by rendering impossible certain kinds of bugs that frequently plague C and C++ programmers. Javaís inherent memory safety--garbage collection, lack of pointer arithmetic, run-time checks on the use of references--prevents most memory bugs from ever occurring. Javaís memory safety makes programmers more productive and helps them manage complexity.

In addition, besides the ongoing increase in the capabilities of hardware, there is another fundamental shift taking place in the hardware environment--the network. As networks interconnect more and more computers and devices, new demands are being made on software. With the rise of the network, platform independence and security have become much more important than they were in the past.

The Java Virtual Machine is responsible for the memory safety, platform neutrality, and security features of the Java programming language. Although virtual machines have been around for a long time, prior to Java they hadnít quite entered the mainstream. But given todayís emerging hardware realities, software developers needed a programming language with a virtual machine, and Sun hit the market window with Java.

Thus, the Java Virtual Machine embodies the right software "stuff" for the coming years of computing. This book will help you get to know this virtual machine, and armed with this knowledge, youíll know how best to put the Java Virtual Machine to use in your programs.

Who Should Read the Book

This book is aimed primarily at professional software developers and students who want to understand Java technology. I assume you are familiar, though not necessarily proficient, with the Java language. Reading this book should help you add a depth to your knowledge of Java programming. If you are one of the elite few who are actually writing Java compilers or creating implementations of the Java Virtual Machine, this book can serve as a companion to the Java Virtual Machine specification. Where the specification specifies, this book explains.

How to Use the Book

This book has five basic parts:

1. An introduction to Javaís architecture (Chapters 1 through 4)
2. An in-depth, technical tutorial of Java internals (Chapters 5 through 20)
3. A class file and instruction set reference (Chapter 6, Appendices A through C)
4. Interactive illustrations, example source code, and the JDK (On the CD-ROM)
5. The Resources Web Site (`http://www.artima.com/insidejvm`)

An Introduction to Javaís Architecture

Chapters 1 through 4 (Part I of this book) give an overview of Javaís architecture, including the motivations behind--and the implications of--Javaís architectural design. These chapters show how the Java Virtual Machine relates to the other components of Javaís architecture: the class file, API, and language. If you want a basic understanding of Java as a technology, consult these chapters. Here are some specific points of interest from this portion of the book:

- For an overview of Javaís architecture and a discussion of its inherent tradeoffs, see Chapter 1, "Introduction to Javaís Architecture."
- For a discussion of what "platform independence" really means, how Javaís architecture supports it, and seven steps to take to create a platform independent Java program, see Chapter 2, "Platform Independence."
- For a description of the security model built into Javaís core architecture, including a tutorial explaining how to write a security-minded class loader, see Chapter 3, "Security." • For a discussion of the new paradigm of network-mobile software, see Chapter 4, "Network Mobility."

A Tutorial of Java Internals

Chapters 5 through 20 (Part II of this book) give an in-depth technical description of the inner workings of the Java Virtual Machine. These chapters will help you understand how Java programs actually work. All the material in Part II is presented in a tutorial manner, with lots of examples. Here are some specific points of interest from this portion of the book:

- For a comprehensive overview of the inner workings of the Java Virtual Machine, see Chapter 5, "The Java Virtual Machine."
- If you are parsing, generating, or simply peering into Java class files, see Chapter 6, "The Java Class File," for a complete tutorial and reference on the class file format.
- For a discussion of the lifetime of a class inside the Java Virtual Machine, including the circumstances in which classes can be unloaded, see Chapter 7, "The Lifetime of a Class." • For a thorough explanation of Javaís linking model, including a tutorial and examples on writing your own class loaders, see Chapter 8, "The Linking Model."
- For a discussion of garbage collection and finalization, and suggestions on how to use finalizers, see Chapter 9, "Garbage Collection."
- For a tutorial on the Java Virtual Machineís instruction set, read Chapters 10 through 20. • For an explanation of monitors and how you can use them to write thread-safe Java code, see Chapter 20, "Thread Synchronization."

A Class File and Instruction Set Reference

In addition to being a tutorial on the Java class file, Chapter 6, "The Java Class File," serves as a complete reference of the class file format. Similarly, Chapters 10 through 20 form a tutorial of the Java Virtual Machineís instruction set, and Appendices A through C serve as a complete reference of the instruction set. If you need to look something up, check out these chapters and the appendices.

Interactive Illustrations, Example Source Code, and the JDK

For most of this bookís chapters, material associated with the chapter--such as example code or simulation applets--appears on the CD-ROM.

The `applets` directory of the CD-ROM contains a mini-website that includes 14 Java applets that illustrate the concepts presented in the text. These "interactive illustrations" form an integral part of this book. Ten of the applets simulate the Java Virtual Machine executing bytecodes. The other applets illustrate garbage collection, twoís-complement and IEEE 754 floating point numbers, and the loading of class files. The applets can be viewed on any platform by any Java-capable browser. The source code for the simulation applets is also included on the CD-ROM.

The copyright notice accompanying the HTML, .java, and .class files for the mini-website enables you to post the mini-website on any network, including the internet, providing you adhere to a few simple rules. For example, you must post the mini-website in its entirety, you canít make any changes to it, and you canít charge people to look at it. The full text of the copyright notice is given later in this introduction.

All the example source code shown in this book appears on the CD-ROM in both source and compiled (class files) form. If some example code in the text strikes you as interesting (or dubious), you can try it out for yourself.

Most of the example code is illustrative and not likely to be of much practical use besides helping you to understand Java. Nevertheless, you are free to cut and paste from the example code, use it in your own programs, and distribute it in binary (such as Java class file) form. The full text of the copyright notice for the example source code is shown later in this introduction.

Besides the interactive illustrations and example source code, the CD-ROM contains one last item: a full distribution of version 1.1.3 of Sunís JDK. This is contained in the CD-ROMís `jdk` directory.

The Resources Web Site

To help you find more information and keep abreast of changes, I maintain a "Resources Web Site" with links to further reading about the material presented in this book. There is at least one "resources page" for each chapter in the book. The main URL of the Resources Web Site is `http://www.artima.com/insidejvm`. The URL for each chapterís individual resources page is given at the end of each chapter in the "The Resources Page" section.

Chapter by Chapter Summary

Part One: Javaís Architecture

Chapter 1. Introduction to Javaís Architecture

This chapter gives an introduction to Java as a technology. It gives an overview of Javaís architecture,

discusses why Java is important, and looks at Javaís pros and cons.

Chapter 2. Platform independence

This chapter shows how Javaís architecture enables programs to run on any platform, discusses the factors that determine the true portability of Java programs, and looks at the relevant tradeoffs.

Chapter 3. Security

This chapter gives an overview of the security model built into Javaís core architecture.

Chapter 4. Network-mobility

This chapter examines the new paradigm of network-mobile software heralded by the arrival of Java, and shows how Javaís architecture makes it possible.

Part Two: Java Internals

Chapter 5. The Java Virtual Machine

This chapter gives a detailed overview of the Java Virtual Machineís internal architecture. Accompanying the chapter on the CD-ROM is an applet, named Eternal Math, that simulates the Java Virtual Machine executing a short sequence of bytecodes.

Chapter 6. The Java Class File

This chapter describes the contents of the class file, including the structure and format of the constant pool, and serves as both a tutorial and a complete reference of the Java class file format. Accompanying the chapter on the CD-ROM is an applet, named Getting Loaded, that simulates the Java Virtual Machine loading a Java class file.

Chapter 7. The Lifetime of a Class

This chapter follows the lifetime of a type (class or interface) from the typeís initial entrance into the virtual machine to its ultimate exit. It discusses the processes of loading, linking, and initialization; object instantiation, garbage collection, and finalization; and type finalization and unloading.

Chapter 8. The Linking Model

This chapter takes an in-depth look at Javaís linking model. It describes constant pool resolution and shows how to write class loaders to enable a Java application to dynamically extend itself at run-time.

Chapter 9. Garbage Collection

This chapter describes various garbage collection techniques and explains how garbage collection works in Java Virtual Machines. Accompanying this chapter on the CD-ROM is an applet, named Heap of Fish, that simulates a compacting, mark-and-sweep garbage-collected heap.

Chapter 10. Stack and Local Variable Operations

This chapter describes the Java Virtual Machine instructions that focus most exclusively on the operand stack--those that push constants onto the operand stack, perform generic stack operations, and transfer

values back and forth between the operand stack and local variables. Accompanying this chapter on the CD-ROM is an applet, named Fibonacci Forever, that simulates the Java Virtual Machine executing a method that generates the Fibonacci sequence.

Chapter 11. Type Conversion

This chapter describes the instructions that convert values from one primitive type to another. Accompanying the chapter on the CD-ROM is an applet, named Conversion Diversion, that simulates the Java Virtual Machine executing a method that performs type conversion.

Chapter 12. Integer Arithmetic

This chapter describes integer arithmetic in the Java Virtual Machine. It explains twoís complement arithmetic and describes the instructions that perform integer arithmetic. Accompanying this chapter on the CD-ROM are two applets that interactively illustrate the material presented in the chapter. One applet, named Inner Int, allows you to view and manipulate a twoís complement number. The other applet, named Prime Time, simulates the Java Virtual Machine executing a method that generates prime numbers.

Chapter 13. Logic

This chapter describes the instructions that perform bitwise logical operations inside the Java Virtual Machine. These instructions include opcodes to perform shifting and boolean operations on integers. Accompanying this chapter on the CD-ROM is an applet, named Logical Results, that simulates the Java Virtual Machine executing a method that includes uses several of the logic opcodes.

Chapter 14. Floating Point Arithmetic

This chapter describes the floating point numbers and the instructions that perform floating point arithmetic inside the Java Virtual Machine. Accompanying this chapter on the CD-ROM are two applets that interactively illustrate the material presented in the chapter. One applet, named Inner Float, allows you to view and manipulate the individual components that make up a floating point number. The other applet, named Circle of Squares, simulates the Java Virtual Machine executing a method that uses several of the floating point opcodes.

Chapter 15. Objects and Arrays

This chapter describes the Java Virtual Machine instructions that create and manipulate objects and arrays. Accompanying this chapter on the CD-ROM is an applet, named Three-Dimensional Array, that simulates the Java Virtual Machine executing a method that allocates and initializes a three-dimensional array.

Chapter 16. Control Flow

This chapter describes the instructions that cause the Java Virtual Machine to conditionally or unconditionally branch to a different location within the same method. Accompanying this chapter on the CD-ROM is an applet, named Saying Tomato, that simulates the Java Virtual Machine executing a method that includes bytecodes that perform table jumps (the compiled version of a Java `switch` statement).

Chapter 17. Exceptions

This chapter shows how exceptions are implemented in bytecodes. It describes the instruction for

throwing an exception explicitly, explains exception tables, and shows how catch clauses work. Accompanying this chapter on the CD-ROM is an applet, named Play Ball!, that simulates the Java Virtual Machine executing a method that throws and catches exceptions.

Chapter 18. Finally Clauses

This chapter shows how finally clauses are implemented in bytecodes. It describes the relevant instructions and gives examples of their use. The chapter also describes some surprising behaviors exhibited by finally clauses in Java source code and explains this behavior at the bytecode level. Accompanying this chapter on the CD-ROM is an applet, named Hop Around, that simulates the Java Virtual Machine executing a method that includes finally clauses.

Chapter 19. Method Invocation and Return

This chapter describes the four instructions that the Java Virtual Machine uses to invoke methods and the situations in which each instruction is used.

Chapter 20. Thread Synchronization

This chapter describes monitors--the mechanism that Java uses to support synchronization--and shows how they are used by the Java Virtual Machine. It shows how one aspect of monitors, the locking and unlocking of data, is supported in the instruction set.

The Appendices

Appendix A. Instruction Set by Opcode Mnemonic

This appendix lists the opcodes alphabetically by mnemonic. For each opcode, it gives the mnemnonic, opcode byte value, instruction format (the operands, if any), a snapshot image of the stack before and after the instruction is executed, and a description of the execution of the instruction. Appendix A serves as the primary instruction set reference of the book.

Appendix B. Opcode Mnemonic by Functional Group

This appendix organizes the instructions by functional group. The organization used in this appendix corresponds to the order the instructions are described in Chapters 10 through 20.

Appendix C. Opcode Mnemonic by Opcode

This appendix organizes the opcodes in numerical order. For each numerical value, this appendix gives the mnemonic.

.

Chapter One

Introduction to Javaís 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. "Javaís 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 Javaís 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. Javaís 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. Javaís 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.

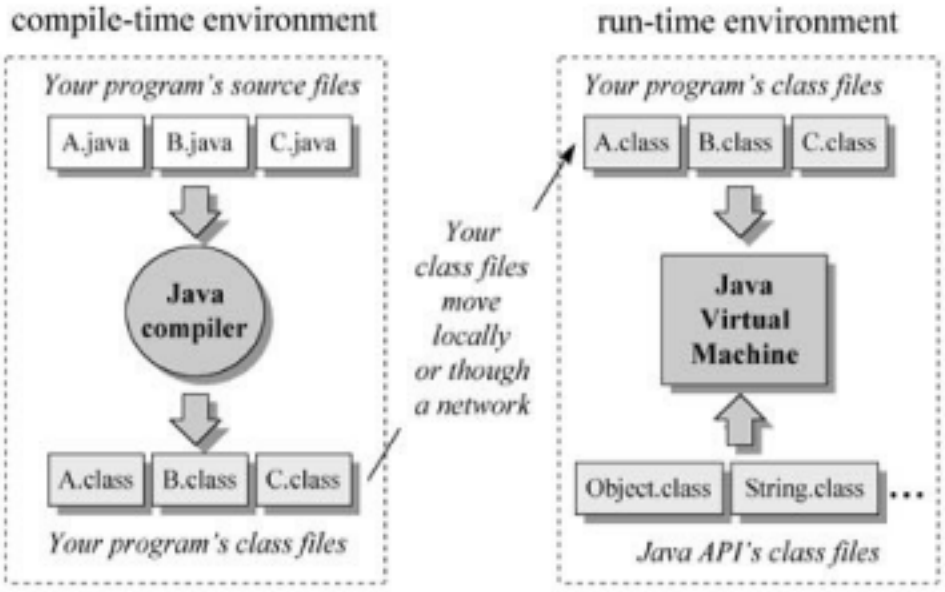


Figure 1-1. The Java programming environment.

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.

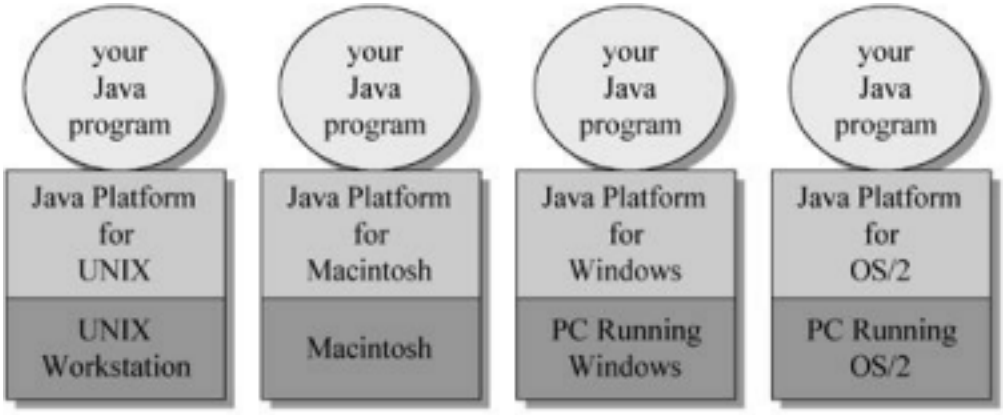


Figure 1-2. Java programs run on top of the Java Platform.

The Java Virtual Machine

At the heart of Javaís network-orientation is the Java Virtual Machine, which supports all three prongs of Javaís 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 Machineís specification enables it to be implemented on a wide variety of computers and devices.

A Java Virtual Machineís 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.

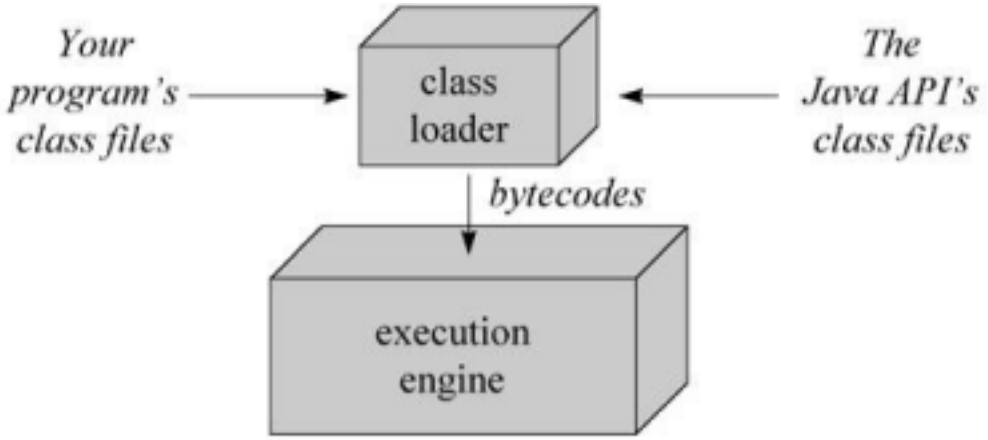


Figure 1-3. A basic block diagram of the Java Virtual Machine.

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.

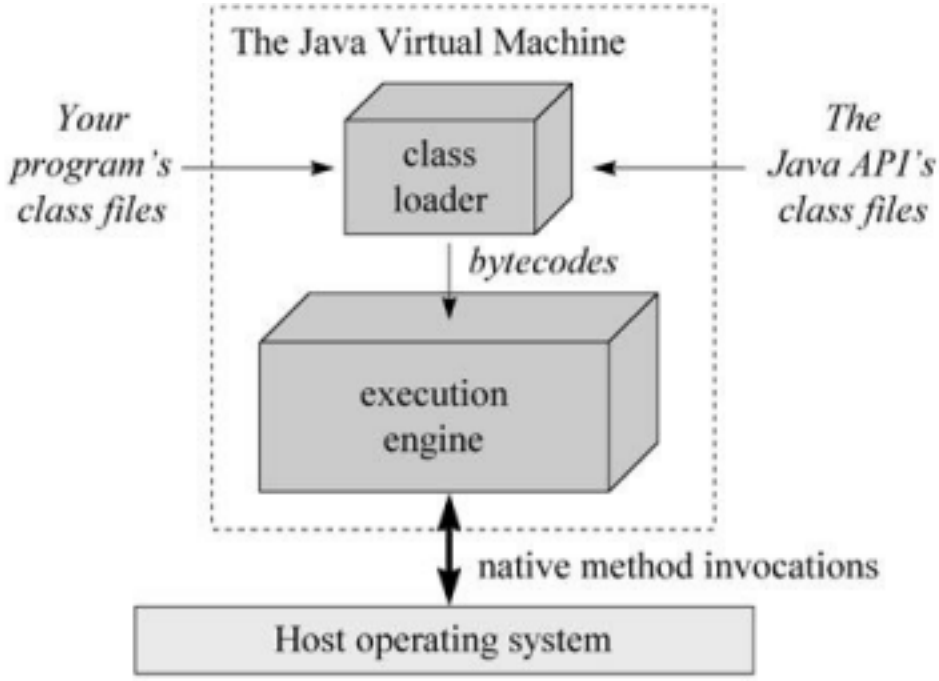


Figure 1-4. A Java Virtual Machine implemented in software on top of a host operating system.

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.

The Class Loader Architecture

One aspect of the Java Virtual Machine that plays an important role in both security and network mobility is the class loader architecture. In the block diagrams of Figures 1-3 and 1-4, a single mysterious cube identifies itself as "the class loader," but in reality there may be more than one class loader inside a Java Virtual Machine. Thus the class loader cube of the block diagram actually represents a subsystem that may involve many class loaders. The Java Virtual Machine has a flexible class loader architecture that allows a Java application to load classes in custom ways.

A Java application can use two types of class loaders: a "primordial" class loader and class loader objects. The primordial class loader (there is only one of them) is a part of the Java Virtual Machine implementation. For example, if a Java Virtual Machine is implemented as a C program on top of an existing operating system, then the primordial class loader will be part of that C program. The primordial class loader loads trusted classes, including the classes of the Java API, usually from the local disk.

At run-time, a Java application can install class loader objects that load classes in custom ways, such as

by downloading class files across a network. The Java Virtual Machine considers any class it loads through the primordial class loader to be trusted, regardless of whether or not the class is part of the Java API. Classes it loads through class loader objects, however, it views with suspicion--by default, it considers them to be untrusted. While the primordial class loader is an intrinsic part of the virtual machine implementation, class loader objects are not. Instead, class loader objects are written in Java, compiled to class files, loaded into the virtual machine, and instantiated just like any other object. They are really just another part of the executable code of a running Java application. You can see a graphical depiction of this architecture in Figure 1-5.

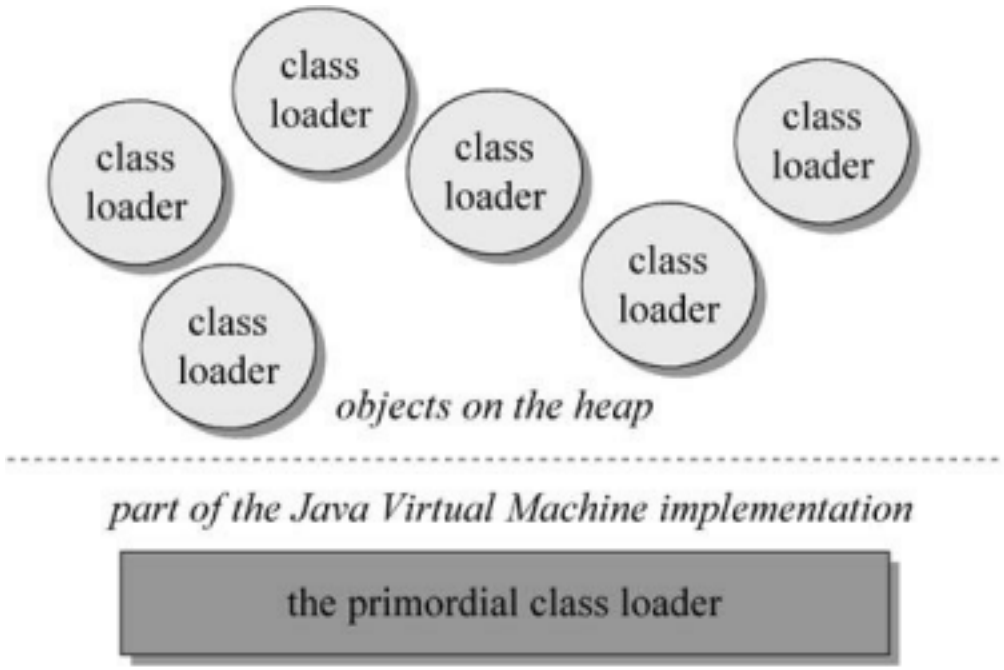


Figure 1-5. Java's class loader architecture.

Because of class loader objects, you donít have to know at compile-time all the classes that may ultimately take part in a running Java application. They enable you to dynamically extend a Java application at run-time. As it runs, your application can determine what extra classes it needs and load them through one or more class loader objects. Because you write the class loader in Java, you can load classes in any manner. You can download them across a network, get them out of some kind of database, or even calculate them on the fly.

For each class it loads, the Java Virtual Machine keeps track of which class loader--whether primordial or object--loaded the class. When a loaded class first refers to another class, the virtual machine requests the referenc*ed* class from the same class loader that originally loaded the referenc*ing* class. For example, if the virtual machine loads class `Volcano` through a particular class loader, it will attempt to load any classes `Volcano` refers to through the same class loader. If `Volcano` refers to a class named `Lava`, perhaps by invoking a method in class `Lava`, the virtual machine will request `Lava` from the class loader object that loaded `Volcano`. The `Lava` class returned by the class loader is dynamically linked with class `Volcano`.

Because the Java Virtual Machine takes this approach to loading classes, classes can by default only see other classes that were loaded by the same class loader. This is how Javaís architecture enables you to create multiple *name-spaces* inside a single Java application. Each class loader in your running Java program maintains its own name-space, which is populated by the names of all the classes it has loaded.

A Java application can instantiate multiple class loader objects either from the same class or from multiple classes. It can, therefore, create as many (and as many different kinds of) class loader objects as it needs. Classes loaded by different class loaders are in different name-spaces and cannot gain access to

each other unless the application explicitly allows it. When you write a Java application, you can segregate classes loaded from different sources into different name-spaces. In this way, you can use Javaís class loader architecture to control any interaction between code loaded from different sources. You can prevent hostile code from gaining access to and subverting friendly code.

One example of dynamic extension is the web browser, which uses class loader objects to download the class files for an applet across a network. A web browser fires off a Java application that installs a class loader object--usually called an *applet class loader*--that knows how to request class files from an HTTP server. Applets are an example of dynamic extension, because the Java application doesnít know when it starts which class files the browser will ask it to download across the network. The class files to download are determined at run-time, as the browser encounters pages that contain Java applets.

The Java application started by the web browser usually creates a different applet class loader object for each location on the network from which it retrieves class files. As a result, class files from different sources are loaded by different class loader objects. This places them into different name-spaces inside the host Java application. Because the class files for applets from different sources are placed in separate name-spaces, the code of a malicious applet is restricted from interfering directly with class files downloaded from any other source.

By allowing you to instantiate class loader objects that know how to download class files across a network, Javaís class loader architecture supports network-mobility. It supports security be allowing you to load class files from different sources through different class loader objects. This puts the class files from different sources into different name-spaces, which allows you to restrict or prevent access between code loaded from different sources.

The Java Class File

The Java class file helps make Java suitable for networks mainly in the areas of platform-independence and network-mobility. Its role in platform independence is serving as a binary form for Java programs that is expected by the Java Virtual Machine but independent of underlying host platforms. This approach breaks with the tradition followed by languages such as C or C++. Programs written in these languages are most often compiled and linked into a single binary executable file specific to a particular hardware platform and operating system. In general, a binary executable file for one platform wonít work on another. The Java class file, by contrast, is a binary file that can be run on any hardware platform and operating system that hosts the Java Virtual Machine.

When you compile and link a C++ program, the executable binary file you get is specific to a particular target hardware platform and operating system because it contains machine language specific to the target processor. A Java compiler, by contrast, translates the instructions of the Java source files into bytecodes, the "machine language" of the Java Virtual Machine.

In addition to processor-specific machine language, another platform-dependent attribute of a traditional binary executable file is the byte order of integers. In executable binary files for the Intel X86 family of processors, for example, the byte order is *little-endian*, or lower order byte first. In executable files for the PowerPC chip, however, the byte order is *big-endian*, or higher order byte first. In a Java class file, byte order is big-endian irrespective of what platform generated the file and independent of whatever platforms may eventually use it.

In addition to its support for platform independence, the Java class file plays a critical role in Javaís architectural support for network-mobility. First, class files were designed to be compact, so they can more quickly move across a network. Also, because Java programs are dynamically linked and dynamically extensible, class files can be downloaded as needed. This feature helps a Java application manage the time it takes to download class files across a network, so the end-userís wait time can be kept to a minimum.

The Java API

The Java API helps make Java suitable for networks through its support for platform independence and security. The Java API is set of runtime libraries that give you a standard way to access the system resources of a host computer. When you write a Java program, you assume the class files of the Java API will be available at any Java Virtual Machine that may ever have the privilege of running your program. This is a safe assumption because the Java Virtual Machine and the class files for the Java API are the required components of any implementation of the Java Platform. When you run a Java program, the virtual machine loads the Java API class files that are referred to by your programís class files. The combination of all loaded class files (from your program and from the Java API) and any loaded dynamic libraries (containing native methods) constitute the full program executed by the Java Virtual Machine.

The class files of the Java API are inherently specific to the host platform. The APIís functionality must be implemented expressly for a particular platform before that platform can host Java programs. In a system where bytecodes are executed directly in silicon (on a "Java chip") the API will likely be implemented as part of a Java-based operating system. On a system where the virtual machine is implemented in software on top of a non-Java operating system, the Java API will access the host resources through native methods. As you can see in Figure 1-6, the class files of the Java API invoke native methods so your Java program doesnít have to. In this manner, the Java APIís class files provide a Java program with a standard, platform-independent interface to the underlying host. To the Java program, the Java API looks the same and behaves predictably no matter what platform happens to be underneath. Precisely because the Java Virtual Machine and Java API are implemented specifically for each particular host platform, Java programs themselves can be platform independent.

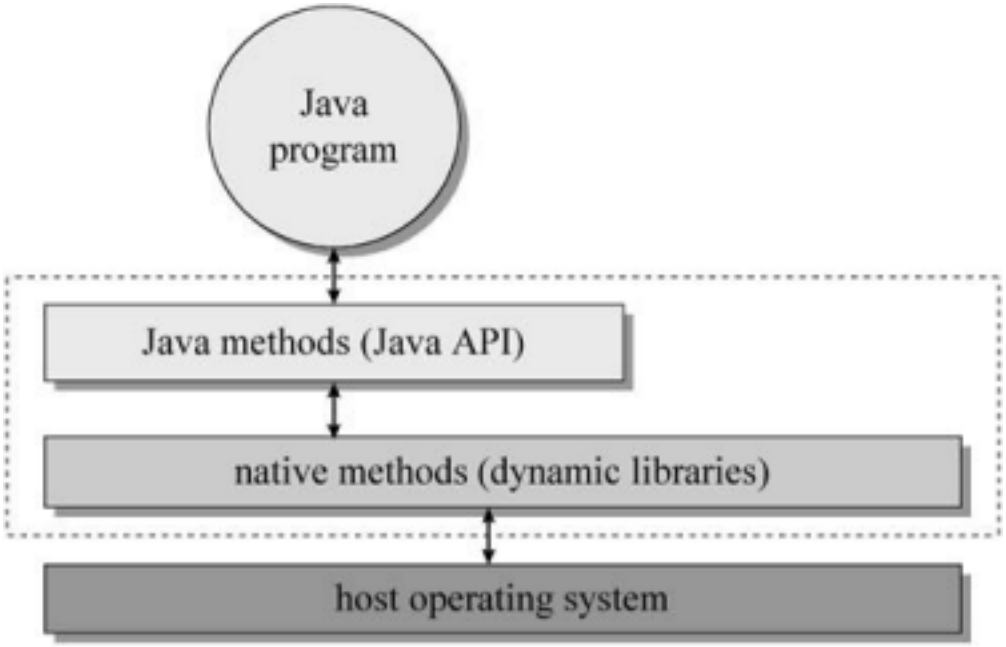


Figure 1-6. A platform-independent Java program.

The internal design of the Java API is also geared towards platform independence. For example, the graphical user interface library of the Java API, called the Abstract Windows Toolkit (or AWT), is designed to facilitate the creation of user interfaces that work on all platforms. Creating platform independent user interfaces is inherently difficult, given that the native look and feel of user interfaces vary greatly from one platform to another. The AWT libraryís architecture does not coerce implementations of the Java API to give Java programs a user interface that looks exactly the same everywhere. Instead, it encourages implementations to adopt the look and feel of the underlying platform. Also, because the size of fonts, buttons, and other user interface components will vary from

platform to platform, the AWT includes *layout managers* to position the elements of a window or dialog box at run-time. Rather than forcing you to indicate exact X and Y coordinates for the various elements that constitute, say, a dialog box, the layout manager positions them when your dialog box is displayed. With the aim of making the dialog look its best on each platform, the layout manager will very likely position the dialog box elements slightly differently on different platforms. In these ways and others, the internal architecture of the Java API is aimed at facilitating the platform independence of the Java programs that use it.

In addition to facilitating platform independence, the Java API contributes to Javaís security model. The methods of the Java API, before they perform any action that could potentially be harmful (such as writing to the local disk), check for permission from the *security manager*. The security manager is a special object that a Java application can instantiate that defines a custom security policy for the application. A security manager could, for example, forbid access to the local disk. If the application requested a local disk write by invoking a method from the Java API, that method would first check with the security manager. Upon learning from the security manager that disk access is forbidden, the Java API would refuse to perform the write. By enforcing the security policy established by the security manager, the Java API helps to establish a safe environment in which you can run potentially unsafe code.

The Java Programming Language

Although Java was designed for the network, its utility is not restricted to networks.

latform independence, network-mobility, and security are of prime importance in a networked computing environment, but you may not always find yourself facing network-oriented problems. As a result, you may not always want to write programs that are platform independent. You may not always want to deliver your programs across networks or limit their capabilities with security restrictions. There may be times when you use Java technology primarily because you want to get the advantages of the Java programming language.

As a whole, Java technology leans heavily in the direction of networks, but the Java programming language is quite general-purpose. The Java language allows you to write programs that take advantage of many software technologies:

- object-orientation
- multi-threading
- structured error-handling
- garbage collection
- dynamic linking
- dynamic extension

Instead of serving as a test bed for new and experimental software technologies, the Java language combines in a new way concepts and techniques that have already been tried and proven in other languages. These features make the Java programming language a powerful general-purpose tool that you can apply to a variety of situations, independent of whether or not they involve a network.

At the beginning of a new project, you may be faced with the question, "Should I use C++ (or some other language) for my next project, or should I use Java?" As an implementation language, Java has some advantages and some disadvantages over other languages. One of the most compelling reasons for using Java as a language is that it can enhance developer productivity. The main disadvantage is slower execution speed.

Java is, first and foremost, an object-oriented language. One promise of object-orientation is that it promotes the re-use of code, resulting in better productivity for developers. This may make Java more attractive than a procedural language such as C, but doesnít add much value to Java over C++, the

object-oriented language that Java most closely resembles. Yet compared to C++, Java has some significant differences that can improve a developerís productivity. This productivity boost comes mostly from Javaís restrictions on direct memory manipulation.

In Java, there is no way to directly access memory by arbitrarily casting pointers to a different type or by using pointer arithmetic, as there is in C++. Java requires that you strictly obey rules of type when working with objects. If you have a *reference* (similar to a pointer in C++) to an object of type `Mountain`, you can only manipulate it as a `Mountain`. You canít cast the reference to type `Lava` and manipulate the memory as if it were a `Lava`. Neither can you simply add an arbitrary offset to the reference, as pointer arithmetic allows you to do in C++. You can, in Java, cast a reference to a different type, but only if the object really is of the new type. For example, if the `Mountain` reference actually referred to an instance of class `Volcano` (a specialized type of `Mountain`), you could cast the `Mountain` reference to a `Volcano` reference. Because Java enforces strict type rules at run-time, you are not able to directly manipulate memory in ways that can accidentally corrupt it. As a result, you canít ever create certain kinds of bugs in Java programs that regularly harass C++ programmers and reduce their productivity.
Another way Java prevents you from inadvertently corrupting memory is through automatic garbage collection. Java has a `new` operator, just like C++, that you use to allocate memory on the heap for a new object. But unlike C++, Java has no corresponding `delete` operator, which C++ programmers use to free the memory for an object that is no longer needed by the program. In Java, you merely stop referencing an object, and at some later time, the garbage collector will reclaim the memory occupied by the object.

The garbage collector prevents Java programmers from needing to explicitly indicate which objects should be freed. As a C++ project grows in size and complexity, it often becomes increasingly difficult for programmers to determine when an object should be freed, or even whether an object has already been freed. This results in memory leaks, in which unused objects are never freed, and memory corruption, in which the same object is accidentally freed multiple times. Both kinds of memory troubles cause C++ programs to crash, but in ways that make it difficult to track down the exact source of the problem. You can be more productive in Java in part because you donít have to chase down memory corruption bugs. But perhaps more significantly, you can be more productive because when you no longer have to worry about explicitly freeing memory, program design becomes easier.

A third way Java protects the integrity of memory at run-time is array bounds checking. In C++, arrays are really shorthand for pointer arithmetic, which brings with it the potential for memory corruption. C++ allows you to declare an array of ten items, then write to the eleventh item, even though that tramples on memory. In Java, arrays are full-fledged objects, and array bounds are checked each time an array is used. If you create an array of ten items in Java and try to write to the eleventh, Java will throw an exception. Java wonít let you corrupt memory by writing beyond the end of an array.

One final example of how Java ensures program robustness is by checking object references, each time they are used, to make sure they are not `null`. In C++, using a null pointer usually results in a program crash. In Java, using a null reference results in an exception being thrown.

The productivity boost you can get just by using the Java language results in quicker development cycles and lower development costs. You can realize further cost savings if you take advantage of the potential platform independence of Java programs. Even if you are not concerned about a network, you may still want to deliver a program on multiple platforms. Java can make support for multiple platforms easier, and therefore, cheaper.

As you might expect, however, all this good news about productivity, quick development cycles, and lower development costs does not come without a catch. The designers of Java made tradeoffs. They designed an architecture that favors network-oriented features--such as platform-independence, program robustness, security, and network-mobility--over other concerns. The primary tradeoff, and thus the

primary hit you will take if you use Java, is execution speed.

Javaís extra run-time housekeeping--array bounds checking, type-safe reference casting, checking for null references, and garbage-collection--will cause your Java program to be slower than an equivalent C++ program. Yet often the tradeoff in speed is made up for in productivity increases enjoyed by the developer and robustness enjoyed by the end-user. And often, the Java program simply runs quickly *enough* to satisfy end-users.

Another speed hit, and one that can be far more substantial, arises from the interpreted nature of Java programs. Whereas C++ programs are usually compiled to native machine code, which is stored in a monolithic executable file, Java programs are usually compiled to Java bytecodes, which are stored in class files. When the Java program runs, a virtual machine loads the class files and executes the bytecodes they contain. When running on a virtual machine that interprets bytecodes, a Java program may be 10 to 30 times slower than an equivalent C++ program compiled to native machine code.

This performance degradation is primarily a tradeoff in exchange for platform independence. Instead of compiling a Java program to platform-specific native machine code, you compile it to platform independent Java bytecodes. Native machine code can run fast, but only on the native platform. Java bytecodes (when interpreted) run slowly, but can be executed on any platform that hosts the Java Virtual Machine.

Fortunately, other techniques can improve the performance of bytecode execution. For example, just-in time compiling can speed up program execution 7 to 10 times over interpreting. Rather than merely interpreting a methodís bytecodes, a virtual machine can compile the bytecodes to native machine code the first time the method is invoked. (The method is compiled "just-in-time" for its first use by the virtual machine.) The native machine code version of the method is then cached by the virtual machine, and re-used the next time the method is invoked by the program. Execution techniques such as just-in time compiling allows Java programs to be delivered as platform-independent class files, and still, in many cases, run quickly enough to satisfy end-users.

Raw execution speed is not always the most important factor determining an end-userís perception of a programís performance. In some situations, programs spend much of their time waiting for data to come across a network or waiting for the user to hit another key on the keyboard. In such cases, even executing the program via an interpreter may be adequate. For more demanding applications, a just-in time compiler may be sufficient to satisfy the end-userís need for speed.

The simulation applets incorporated into Part II of this book are an example of a type of program for which execution speed is not that critical. Most of time in these programs is spent waiting for the user to click a button. For many programs, however, execution speed is extremely important. For such programs, if you want to use the Java language, you may have to execute part or all of your program natively. One way to do that is to run the class files on a virtual machine built on top of a chip that executes bytecodes directly in silicon. If you (or your end-users) donít have such a chip handy, another possibility is to identify time-critical portions of your program and implement them as native methods. Using native methods yields a program that is delivered as a combination of platform independent class files and platform-specific dynamic libraries. The bytecodes from the class files are executed by interpreting or just-in-time compiling, but the time-critical code stored in the dynamic libraries is executed natively.

One final alternative is to compile the Java program to a platform-specific, monolithic native executable, as is usually done with C++ programs. Such a strategy bypasses class files entirely, and generates a platform-specific binary. A monolithic native executable can be faster than the same program just-in time compiled for several reasons. First, just-in-time compilers do not usually do as much optimization as native compilers because of the time trade-off. When compiling a Java program to a monolithic native executable, you have plenty of time to spend performing optimization. When just-in-time

compiling, however, time is more scarce. The whole point of just-in-time compiling is to speed up program execution on the fly, but at some stage the speedup gained by certain optimizations will not be worth the time spent doing the optimization. Another reason using just-in-time compiler is slower than a native executable is the just-in-time compiled program will likely occupy a larger memory footprint. The larger footprint could require more paging (or swapping) on a virtual memory system.

So when you compile your Java program to a monolithic native executable, you give up binary platform independence in return for speed. In cases where platform independence is not important to you, or speed is more important, compiling to a native executable can give you both fast execution and the productivity benefits of the Java language.

One way to get the best of both the platform independence and speed execution worlds is by install-time compiling. In this scheme, you deliver platform-independent class files, which are compiled at install time to a platform-specific, monolithic native executable. The binary form that you deliver (Java class files) is platform independent, but the binary form that the end-user executes (a monolithic native executable) is platform specific. Because the translation from class files to native executable is done during installation on the end-userís system, optimizations can be made for the userís particular system setup.

Java, therefore, gives you many options of program delivery and execution. Moreover, if you write your program in the Java language, you need not choose just one option. You can use several or all methods of program delivery and execution made possible by Java. You can deliver the same program to some users over a network, where they are executed via interpreting or just-in-time compiling. To other users you can deliver class files that are install-time compiled. To still other users you can deliver a monolithic native executable.

Although program speed is a concern when you use Java, there are ways you can address it. By appropriate use of the various techniques for developing, delivering, and executing Java programs, you can often satisfy end-userís expectations for speed. As long as you are able to address the speed issue successfully, you can use the Java language and realize its benefits: productivity for the developer and program robustness for the end-user.

Architectural Tradeoffs

Although Javaís network-oriented features are desirable, especially in a networked environment, they did not come for free. They required tradeoffs against other desirable features. Whenever a potential tradeoff between desirable characteristics arose, the designers of Java made the architectural choice that made better sense in a networked world. Hence, Java is not the right tool for every job. It is suitable for solving problems that involve networks and has utility in many problem that donít involve networks, but its architectural tradeoffs will disqualify it for certain types of jobs.

As mentioned before, one of the prime costs of Javaís network-oriented features is the potential reduction in program execution speed compared to other technologies such as C++. Java programs can run slower than an equivalent C++ program for many reasons:

- Interpreting bytecodes is 10 to 30 times slower than native execution.
- Just-in-time compiling bytecodes can be 7 to 10 times faster than interpreting, but still not quite as fast as native execution.
- Java programs are dynamically linked.
- The Java Virtual Machine may have to wait for class files to download across a network.
- Array bounds are checked on each array access.
- All objects are created on the heap (no objects are created on the stack).
- All uses of object references are checked at run-time for `null`.

- All reference casts are checked at run-time for type safety.
- The garbage collector is likely less efficient (though often more effective) at managing the heap than you could be if you managed it directly as in C++.
- Primitive types in Java are the same on every platform, rather than adjusting to the most efficient size on each platform as in C++.
- Strings in Java are always UNICODE. When you really need to manipulate just an ASCII string, a Java program will be slightly less efficient than an equivalent C++ program.

Although many of Javaís speed hits are manageable through techniques such as just-in-time compiling, some--such as those that result from run-time checking--canít be eliminated even by compilation to native executable. Still, you get something, such as platform independence or program robustness, for all of the speed hits associated with Java programs. In many cases the end-user will not be able to perceive any speed deficit. In many other cases, the benefits of platform independence and improved program robustness will be worth the speed degradation. Sometimes, however, Java may be disqualified as a tool to help you solve a problem because that problem requires the utmost in speed and Java canít deliver it.

Another tradeoff is loss of control of memory management. Garbage collection can help make programs more robust and easier to design, but adds a level of uncertainty to the runtime performance of the program. You canít always be sure when a garbage collector will decide it is time to collect garbage, nor how long it will take. This loss of control of memory management makes Java a questionable candidate for software problems that require a real-time response to events. While it is possible to create a garbage collector that attempts to meet real-time requirements, for many real-time problems, robustness and platform independence are simply not important enough to justify using Java.

Still another tradeoff arises from Javaís goal of platform independence. One difficulty inherent in any API that attempts to provide cross-platform functionality is the lowest-common-denominator problem. Although there is much overlap between operating systems, each operating system usually has a handful of traits all its own. An API that aims to give programs access to the system services of any operating system has to decide which capabilities to support. If a feature exists on only one operating system, the designers of the API may decide not to include support for that feature. If a feature exists on most operating systems, but not all, the designers may decide to support it anyway. This will require an implementation of something similar in the API on operating systems that lack the feature. Both of these lowest-common-denominator kinds of choices may to some degree offend developers and end-users on the affected operating systems.

Whatís worse, not only does the lowest-common-denominator problem afflict the designers of a platform independent API, it also affects the designer of a program that uses that API. Take user interface as an example. The AWT attempts to give your program a user interface that adopts the native look on each platform. You might find it difficult, however, to design a user interface in which the components interact in a way that *feels* native on every platform, even though the individual components may have the native look. So on top of the lowest-common-denominator choices that were made when the AWT was designed, you may find yourself faced with your own lowest-common-denominator choices when you use the AWT.

One last tradeoff stems from the dynamically linked nature of Java programs combined with the close relationship between Java class files and the Java programming language. Because Java programs are dynamically linked, the references from one class file to another are symbolic. In a statically-linked executable, references between classes are direct pointers or offsets. Inside a Java class file, by contrast, a reference to another class spells out the name of the other class in a text string. If the reference is to a field, the fieldís name and *descriptor* (the fieldís type) are also specified. If the reference is to a method, the methodís name and descriptor (the methodís return type, number and types of its arguments) are specified. Moreover, not only do Java class files contain symbolic references to the fields and methods

of other classes, they also contain symbolic references to their own fields and methods. Java class files also may contain optional debugging information that includes the names and types of local variables. A class fileís symbolic information, and the close relationship between the bytecode instruction set and the Java language, make it quite easy to decompile Java class files back into Java source. This in turn makes it quite easy for your competitors to borrow heavily from your hard work.

While it has always been possible for competitors to decompile a statically-linked binary executable and glean insights into your program, by comparison decompilation is far easier with an intermediate (not yet linked) binary form such as Java class files. Decompilation of statically-linked binary executables is more difficult not only because the symbolic information (the original class, field, method, and local variable names) is missing, but also because statically-linked binaries are usually heavily optimized. The more optimized a statically-linked binary is, the less it corresponds to the original source code. Still, if you have an algorithm buried in your binary executable, and it is worth the trouble to your competitors, they can peer into your binary executable and retrieve that algorithm.

Fortunately, there is a way to combat the easy borrowing of your intellectual property: you can obfuscate your class files. Obfuscation alters your class files by changing the names of classes, fields, methods, and local variables, but without altering the operation of the program. Your program can still be decompiled, but will no longer have the (hopefully) meaningful names you originally gave to all of your classes, fields, methods, and local variables. For large programs, obfuscation can make the code that comes out of the decompiler so cryptic as to require nearly the same effort to steal your work as would be required by a statically-linked executable.

Future Trends

As Java matures, some of the tradeoffs described in this chapter may change. One area in which you can expect improvement over time is in the execution speed of Java programs. Sun, for example, is currently working on a technology they call "hot-spot compiling," which is a hybrid of interpreting and just-in time compiling. They claim this technique will yield Java programs that run as fast as natively compiled C++. Although this seems like a rash claim, when you look at the approach, it makes sense that speeds very close to natively compiled C++ could be achievable.

As a programmer, you may sometimes be faced with the task of speeding up a program by looking through your code for ways to optimize. Often, programmers waste time optimizing code that is rarely executed when the program runs. The proper approach is usually to profile the program to discover exactly where the program spends most of its time. Programs often spend 80 or 90 percent of their time in 10 to 20 percent of the code. To be most effective, you should focus your optimization efforts on just the 10 to 20 percent of the code that really matters to execution speed.

In a sense, a Java Virtual Machine that does just-in-time compiling is like a programmer who spends time to optimize all the code in a program. 80 to 90 percent of the time such a virtual machine spends just-in-time compiling is probably spent on code that only runs 10 to 20 percent of the time. Because all the code is just-in-time compiled, the memory footprint of the program grows much larger than that of an interpreted program, where all the code remains in bytecode form. Also, because so much time is spent just-in-time compiling everything, the virtual machine doesnít have enough time left over to do a thorough job of optimization.

A Java Virtual Machine that does hot-spot compiling, by contrast, is like a programmer who profiles the code and only optimizes the codeís time-critical portions. In this approach, the virtual machine begins by interpreting the program. As it interprets bytecodes, it analyzes the execution of the program to determine the programís "hot spot" --that part of the code where the program is spending most of its time. When it identifies the hot spot, it just-in-time compiles only that part of the code that makes up the hot spot. As the program continues to run, the virtual machine continues to analyze it. If the hot spot moves, the virtual machine can just-in-time compile and optimize new areas as they move into the hot

spot. Also, it can revert back to using bytecodes for areas that move out of the hot spot back, to keep the memory footprint at a minimum.

Because only a small part of the program is just-in-time compiled, the memory footprint of the program remains small and the virtual machine has more time to do optimizations. On systems with virtual memory, a smaller memory footprint means less paging. On systems that lack virtual memory--such as many embedded devices--a smaller memory footprint may mean the difference between a program fitting or not fitting in memory at all. More time for optimizations yields hot-spot code that could potentially be optimized as much as natively compiled C++.

In the hot-spot compiling approach, the Java Virtual Machine loads platform-independent class files, just-in-time compiles and heavily optimizes only the most time-critical code, and interprets the rest of the code. Such a program could spend 80 to 90 percent of its time executing native code that is optimized as heavily as natively compiled C++. At the same time, it could keep a memory footprint that is not much larger than a Java program that is 100 percent interpreted. It makes sense that a Java program running on such a virtual machine could achieve speeds very close to the speed of natively compiled C++.

If emerging technologies, such as hot-spot compiling, fulfill their promise, the speed tradeoff of Java programs could eventually become much less significant. It remains to be seen, however, what execution speeds such technologies will actually be able to achieve. For links to the latest information about emerging virtual machine technologies, visit the resources page for this chapter.

Another area in which much work is being done is user interface. One of the tradeoffs listed in this chapter for writing platform-independent programs is the lowest-common-denominator problem. A major area in which this problem reveals itself is user interfaces. In an effort to provide a user-interface library that could map to native components on most platforms, Sun filled the AWT library in Java 1.0 and 1.1 with a lowest-common-denominator subset of components. The 1.0 AWT library included a button class, for example, because every platform had a native button. The library did not include more advanced components such as tab controls or spin controls, however, in part because of schedule constraints, but also because these kinds of controls werenít native to enough platforms.

The Java programmer was faced with an AWT library that directly supported the creation of rather simple user interfaces. With work, however, the programmer could build a fancier user interface on top of the AWT primitives. Many third party vendors built more advanced user interface libraries on top of AWT to help ease the programmerís burden. Microsoftís AFC (Application Foundation Classes) and Netscapeís IFC (Internet Foundation Classes) are two good examples. These libraries add support for more advanced user-interface components and functionality to those directly supported by the AWT. They are, however, built on top of AWT, so programs that use them are still platform independent.

Sun has announced JFC (Java Foundation Classes), which is their approach to solving the lowest common-denominator problem with the 1.0 and 1.1 AWT libraries. Rather than attempting to map more components to native counterparts, Sunís strategy is to provide what they call "lightweight components." A lightweight component doesnít directly map to a native component. Instead, it is built out of the AWT primitives. So for example, instead of providing a tab control that maps to a native tab control on each platform that supports one, JFC would provide a "Java Platform tab control". When such a control is used on Windows95, which supports tab controls natively, the control would not necessarily have the native Windows look and feel. It would have the Java Platform look and feel.

As Java user interface libraries evolve, they will reduce the pain of writing platform-independent user interfaces. Sunís lightweight component approach could enable the Java Platform to become more of a driving force in the evolution of user-interface. Rather than just trying to catch up with the user interfaces available on native platforms, Sun can develop the user interface of the Java Platform. With lightweight components, they need not be restrained by lowest-common-denominator choices between

native user interfaces.

Itís not clear to what extent users will accept a Java Platform look and feel over a native one, but user interface does seem to be evolving towards more heterogeneity. Back in the eighties, the Apple Macintosh established a principle that stated all Macintosh applications should adhere to certain user interface guidelines. The theory was that software would be easier to use if all a userís applications were homogeneous: if they all used familiar metaphors and exhibited the same look and feel. Today, however, when a Macintosh user browses the World Wide Web, they donít expect every web page to look like a Macintosh page. When they go to the IBM site, they expect it to look like IBM. When they go to the Disney site, they expect it to look like Disney. This is similar to the real world in that when you go to New York, you expect it to look and feel like New York. When you go to Paris, you expect it to look and feel like Paris. You donít expect all cities to have the same look and feel. As users are exposed to the web, they are becoming accustomed to working with more heterogeneous user interfaces than they might have encountered on an isolated personal computer.

As Java user-interface libraries evolve, the lowest-common-denominator problem inherent in platform independent user interfaces may gradually become less painful. It remains to be seen, however, the extent to which users will accept interfaces that do not look and feel 100% native. For links to the latest information about the evolution of user interface technologies for Java, visit the resources page for this chapter.

The Resources Page

For links to more information about the material presented in this chapter, visit the resources page at http://www.artima.com/insidejvm/intro.html

Chapter Two

Platform Independence

The last chapter showed how Javaís architecture makes it a useful tool for developing software solutions in a networked environment. The next three chapters take a closer look at how Javaís architecture accomplishes its suitability for networks. This chapter examines platform independence in detail. It shows how Javaís architecture enables programs to run on any platform, discusses the factors that determine the true portability of Java programs, and looks at the relevant tradeoffs.

Why Platform Independence?

One of the key reasons Java technology is useful in a networked environment is that Java makes it possible to create binary executables that will run unchanged on multiple platforms. This is important in a networked environment because networks usually interconnect many different kinds of computers and devices. An internal network at a medium-sized company might connect Macintoshes in the art department, UNIX workstations in engineering, and PCs running Windows everywhere else. Also, various kinds of embedded devices, such as printers, scanners, and fax machines, would typically be connected to the same network. Although this arrangement enables various kinds of computers and devices within the company to share data, it requires a great deal of administration. Such a network presents a system administrator with the task of keeping different platform-specific editions of programs up to date on many different kinds of computers. Programs that can run without change on any networked computer, regardless of the computerís type, make the system administratorís job simpler, especially if those programs can actually be delivered across the network.

On the developerís side, Java can reduce the cost and time required to develop and deploy applications on multiple platforms. Even though historically, many (or most) applications have been supported on only one platform, often the reason was that the cost involved in supporting multiple platforms wasnít worth the added return. Java can help make multi-platform support affordable for more types of programs.

For software developers, Javaís platform independence can be both an advantage and a disadvantage. If you are developing and selling a software product, Javaís support for platform independence can help you to compete in more markets. Instead of developing a product that runs only on Windows, for example, you can write one that runs on Windows, Macintosh, UNIX, and OS/2. With Java, you can have more potential customers. The trouble is, so can everyone else. Imagine, for example, that you have focused your efforts on writing great software for OS/2. Java makes it easier for others to write software that competes in your chosen market niche. With Java, therefore, you may not only end up with more potential customers, but also with more potential competitors.

Javaís Architectural Support for Platform Independence

Javaís architecture facilitates the creation of platform-independent software, but also allows you to create software that is platform-specific. When you write a Java program, platform independence is an *option*.

Support for platform independence, like support for security and network-mobility, is spread throughout Javaís architecture. All the components of the architecture--the language, the class file, the API, and the virtual machine--play a role in enabling platform independence.

The Java Platform

Javaís architecture supports the platform independence of Java programs in several ways, but primarily through the Java Platform itself. The Java Platform acts as a buffer between a running Java program and the underlying hardware and operating system. Java programs are compiled to run on a Java Virtual Machine, with the assumption that the class files of the Java API will be available at run-time. The virtual machine runs the program; the API gives the program access the underlying computerís resources. No matter where a Java program goes, it need only interact with the Java Platform. It neednít worry about the underlying hardware and operating system. As a result, it can run on any computer that hosts a Java Platform.

The Java Language

The Java programming language reflects Javaís platform independence in one principal way: the ranges and behavior of its primitive types are defined by the language. In languages such as C or C++, the range of the primitive type `int` is determined by its size, and its size is determined by the target platform. The size of an `int` in C or C++ is generally chosen by the compiler to match the word size of the platform for which the program is compiled. This means that a C++ program might have different behavior when compiled for different platforms merely because the ranges of the primitive types are not consistent across the platforms. For example, no matter what underlying platform might be hosting the program, an `int` in Java behaves as a signed 32-bit twoís complement number. A `float` adheres to the 32-bit IEEE 754 floating point standard. This consistency is also reflected in the internals of the Java Virtual Machine, which has primitive data types that match those of the language, and in the class file, where the same primitive data types appear. By guaranteeing that primitive types behave the same on all platforms, the Java language itself promotes the platform independence of Java programs.

The Java Class File

As mentioned in the previous chapter, the class file defines a binary format that is specific to the Java Virtual Machine. Java class files can be generated on any platform. They can be loaded and run by a Java Virtual Machine that sits on top of any platform. Their format, including the big-endian order of multi-byte values, is strictly defined and independent of any platform that hosts a Java Virtual Machine.

Scaleability

One aspect of Javaís support for platform independence is its scaleability. The Java Platform can be implemented on a wide range of hosts with varying levels of resources, from embedded devices to mainframe computers.
Even though Java first came to prominence by riding on top of a wave that was crashing through the desktop computer industry, the World Wide Web, Java was initially envisioned as a technology for embedded devices, not desktop computers. Part of the early reasoning behind Java was that although Microsoft and Intel had a dominant clutch on the desktop market, no such dominance existed in the embedded systems market. Microprocessors had been appearing in device after device for years--audio video equipment, cell phones, printers, fax machines, copiers--and the coming trend was that, increasingly, embedded microprocessors would be connected to networks. An original design goal of Java, therefore, was to provide a way for software to be delivered across networks to any kind of embedded device--independent of its microprocessor and operating system.

To accomplish this goal, the Java runtime system (the Java Platform) had to be compact enough to be implemented in software using the resources available to a typical embedded system. Embedded microprocessors often have special constraints, such as small memory footprint, no hard disk, a non graphical display, or no display.

Given the special requirements of embedded systems, several incarnations of the Java Platform exist just for embedded systems:

- the Java Embedded Platform
- the Java Personal Platform
- the Java Card Platform

These Java Platforms are composed of a Java Virtual Machine and a smaller shell of runtime libraries than are available in the Java Core Platform. The difference between the Core and the Embedded Platform, therefore, is that the Embedded Platform guarantees the availability of fewer Java API runtime libraries. The Personal Platform guarantees fewer APIs than the Embedded Platform, and the Card Platform fewer than the Personal.

In addition to guaranteeing the smallest set of APIs, the Card Platform, which is targeted at SmartCards, uses only a subset of the full Java Virtual Machine instruction set. Only a subset of the features of the Java language are supported by this smaller instruction set. As a result, only Java programs that restrict themselves to features available on the Card Platform can run on a SmartCard.

Because the Java Platform is compact, it can be implemented on a wide variety of embedded systems. The compactness of the Java Platform, however, does not restrict implementation at the opposite end of the spectrum. The Java Platform also scales up to personal computers, workstations, and mainframes.

Factors that Influence Platform Independence

When you write a Java program, its degree of platform independence depends on several factors. As a developer, some of these factors are beyond your control, but most are within your control. Primarily, the degree of platform independence of any Java program you write depends on how you write it.

Java Platform Deployment

The most basic factor determining the a Java programís platform independence is the extent to which the Java Platform has been deployed on multiple platforms. Java programs will only run on computers and devices that host a Java Platform. Thus, before one of your Java programs will run on a particular computer owned by, say, your friend Alicia, two things must happen. First, the Java Platform must be ported to Aliciaís particular type of hardware and operating system. Once the port has been done by some Java Platform vendor, that port must in some way get installed on Aliciaís computer. So a critical factor determining the true extent of platform independence of Java programs--and one that is beyond the control of the average developer--is the availability of Java Platform implementations and their distribution.

Fortunately for the Java developer, the deployment of the Java Platform has proceeded with great momentum, starting with Web-browsers, then moving on to desktop, workstation, and network operating systems. With the advent of chips optimized to execute Java bytecodes efficiently, the Java Platform will to some extent work its way into many different kinds of embedded devices. It is increasingly likely, therefore, that your friend Alicia will have a Java Platform implementation on her computer.

The Java Platform Version and Edition

The deployment of the Java Platform is a bit more complicated, however, because not all standard runtime libraries are guaranteed to be available at every Java Platform. The basic set of libraries guaranteed to be available at a Java Platform is called the *Java Core API*. A Java Virtual Machine accompanied by the class files that constitute the Core API is called the *Java Core Platform*. This edition of the Java Platform has the minimum set of Java API libraries that you can assume will be available at network computers, desktop computers, and workstations. As mentioned earlier, three other editions of the Java Platform--the Embedded, Personal, and Card Platforms--provide subsets of the Core API for embedded systems. The standard runtime libraries not guaranteed to be available at the Core Platform are collectively called the *Java Standard Extension API*. These libraries include such services as telephony, commerce, and media such as audio, video, or 3D. If your program uses libraries from the Standard Extension API, it will run anywhere those standard extension API libraries are available, but not on a computer that implements only the basic Java Core Platform.

Another complicating factor is that in a sense the Java Platform is a moving target--it evolves over time. Although the Java Virtual Machine is likely to evolve very gradually, the Java API will probably change more frequently. Over time, features will be added to and removed from both the Core and Standard Extension APIs, and parts of the Standard Extension API may migrate into the Core API. The changes made to the Java Platform should for the most part be upwards compatible, meaning they wonít break existing Java programs, but some changes may not be. As obsolete features are removed in a new version of the Java Platform, existing Java programs that depend upon those features wonít run on the new version. Also, changes may not be downwards compatible, meaning programs that are compiled for a new version of the Java Platform wonít necessarily work on an old version. The dynamic nature of the Java Platform complicates things somewhat for the developer wishing to write a Java program that will run on any computer.

In theory, your program should run on all computers that host a Java Core Platform so long as you depend only upon the runtime libraries in the Core API. In practice, however, new versions of the Core API will take time to percolate everywhere. When your program depends on newly added features of the latest version of the Java Core API, there may be some hosts that canít run it because they have an older version. This is not a new problem to software developers--programs written for Windows 95, for example, donít work on the previous version of the operating system, Windows 3.1--but because Java enables the network delivery of software, it becomes a more acute problem. The promise of Java is not

only that it is easy to port programs from one platform to another, but that one version of a binary executable Java program placed on a server can be delivered across a network and run on all computers.

As a developer, you canít control the release cycles or deployment schedules of the Java Platform, but you can choose the Java Platform edition and version that your programs depend upon. In practice, therefore, you will have to decide when a new version of the Java Platform has been distributed to a great enough extent to justify writing programs for that version.

Native Methods

Besides the Java Platform version and edition your program depends on, the other major factor determining the extent of platform independence of your Java program is whether or not you call native methods. The most important rule to follow when you are writing a platform independent Java program is: donít directly or indirectly invoke any native methods that arenít part of the Java API. As you can see in Figure 2-1, calling native methods outside the Java API renders your program platform-specific.

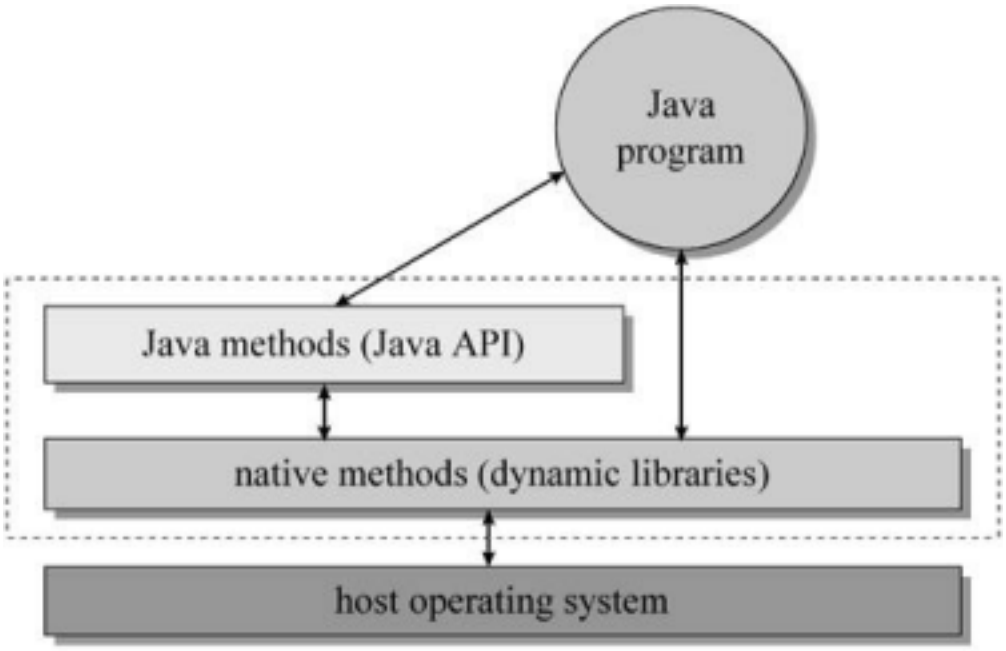


Figure 2-1. A platform-specific Java program.

Calling native methods directly is appropriate in situations where you donít desire platform independence. In general, native methods are useful in three cases:

- for accessing features of an underlying host platform that are not accessible through the Java API
- for accessing a legacy system or using an already existing library that isnít written in Java
- for speeding up the performance of a program by implementing time-critical code as native methods

If you need to use native methods and also need your program to run on several platforms, youíll have to port the native methods to all the required platforms. This porting must be done the old fashioned way, and once youíve done it, youíll have to figure out how to deliver the platform-specific native method libraries to the appropriate hosts. Because Javaís architecture was designed to simplify multi-platform support, your initial goal in writing a platform-independent Java program should be to avoid native methods altogether and interact with the host only through the Java API.

Non-Standard Runtime Libraries

Native methods arenít inherently incompatible with platform independence. Whatís important is whether or not the methods you invoke are implemented "everywhere." Implementations of the Java API on

operating systems such as Windows or Solaris use native methods to access the host. When you call a method in the Java API, you are certain it will be available everywhere. It doesnít matter if in some places the method is implemented as a native method.

Java Platform implementations can come from a variety of vendors, and although every vendor must supply the standard runtime libraries of the Java API, individual vendors may also supply extra libraries. If you are interested in platform independence, you must remain aware of whether any non-standard runtime libraries you use call native methods. Non-standard libraries that donít call native methods donít degrade your programís platform independence. Using non-standard libraries that do call native methods, however, yields the same result as calling native methods directly--it renders your program platform-specific.

For example, Microsoft offers several sets of non-standard runtime libraries with their Java Platform implementation. One set is the Application Foundation Classes (or AFC). The AFC library is delivered along with the Java Platform implementation in Microsoftís Internet Explorer (Version 4.0 and beyond) web browser. The AFC library extends the capabilities provided by the standard runtime libraries of the Java API, but doesnít call native methods outside the Java API. The AFC library accesses the host only through the Java API. As a result, programs that use AFC should run on all implementations of the Java Platform.

One concern surrounding the use of a non-standard class library (such as AFC) that doesnít call native methods, is you must deliver it to Java Platforms that donít support it directly. For instance, imagine you write a Java program that uses AFC and make it available for download across a network. If you want the program to run on all platforms, youíll have to make the AFC classes available for download too. Compared to other browsers, the program might get started sooner at Microsoftís browser because it already has the AFC classes close at hand. Other browsers that donít store the AFC library locally would have to download the needed AFC classes across the network. Other than this potential difference in download time, using a non-standard library that interacts with the host only through the Java API wonít reduce the platform independence of a Java program. To keep your program platform independent, however, you must deliver the library along with your program.

One other Microsoft extension to the standard libraries of the Java Platform gives you a way to generate Java class files that grant access to COM (Component Object Model) objects. If you want to interact with a particular COM object, you use Microsoftís tool to generate a Java class file that gives your program a Java interface to the COM object. In effect, the generated class files form an extra runtime library that is available only at Microsoftís Java Platform implementation. Although this extra library offers more capabilities to Java programs, it reduces the platform independence of any Java program that takes advantage of it. Why? Because the extra libraries call native methods that are, initially at least, only available on Windows 95 and Windows NT. (In the future, Microsoft plans to port the Java/COM interface to non-Microsoft platforms.) Because the extra libraries provided by Microsoft use native methods that are specific to Microsoft operating systems, Java programs that use the extra libraries will only work on Microsoft operating systems.

Another potential ramification of using a vendorís non-standard runtime library that calls native methods directly is that your program will only work on that vendorís Java Platform implementation. For example, the class files described above that give your Java program access to COM objects use a special native method interface (the Java/COM interface) of Microsoftís Java Virtual Machine. Currently, this special native method interface is available only on Microsoftís virtual machine implementation on Windows 95 and Windows NT. Microsoft may port the Java/COM interface to other platforms, to increase the platform independence of programs that take advantage of it. But even if Microsoft is able to port it to every major platform, Java programs that use it will most likely only work properly when running on a Microsoft implementation of the Java Virtual Machine.

Because Java Platforms can come from different vendors, there can be different Java Platform

implementations from different vendors for the same hardware and operating system. To run a Java program on Windows 95, for example, you could use a Java Platform from Sun, Microsoft, Borland, Symantec, or Asymetrix. The level of platform independence that a Java program has depends not only on how many different host computers it can run on, but also on how many different Java Platform implementations it can run on each host.

Virtual Machine Dependencies

Two other rules to follow when writing a platform independent Java program involve portions of the Java Virtual Machine that can be implemented differently by different vendors. The rules are:

1. donít depend upon timely finalization for program correctness, and
2. donít depend upon thread prioritization for program correctness.

These two rules address the variations allowed in the Java Virtual Machine specification for garbage collection and threads.

All Java Virtual Machine must have a garbage-collected heap, but different implementations can use different garbage collection techniques. This flexibility in the Java Virtual Machine specification means that the objects of a particular Java program can be garbage collected at completely different times on different virtual machines. This in turn means that finalizers, which are run by the garbage collector before an object is freed, can run at different times on different virtual machines. If you use a finalizer to free finite memory resources, such as file handles, your program may run on some virtual machine implementations but not others. On some implementations, your program could run out of the finite resource before the garbage collector gets around to invoking the finalizers that free the resource.

Another variation allowed in different implementations of the Java Virtual Machine involves thread prioritization. The Java Virtual Machine specification guarantees that all runnable threads at the highest priority in your program will get some CPU time. The specification also guarantees that lower priority threads will run when higher priority threads are blocked. The specification does not, however, prohibit lower priority threads from running when higher priority threads arenít blocked. On some virtual machine implementations, therefore, lower priority threads may get some CPU time even when the higher priority threads arenít blocked. If your program depends for correctness on this behavior, however, it may work on some virtual machine implementations but not others. To keep your multi threaded Java program platform independent, you must rely on synchronization--not prioritization--to coordinate inter-activity between threads.

User Interface Dependencies

Another major variation between different Java Platform implementations is user interface. User interface is one of the more difficult issues in writing platform independent Java programs. The AWT user interface library gives you a set of basic user-interface components that map to native components on each platform. Libraries such as Microsoftís AFC, Netscapeís IFC, and Sunís JFC, give you advanced components that donít map directly to native components. From this raw material, you must build an interface that users on many different platforms will feel comfortable with. This is not always an easy task.

Users on different platforms are accustomed to different ways of interacting with their computer. They metaphors are different. The components are different. The interaction between the components is different. Although the AWT library makes it fairly easy to create a user interface that runs on multiple platforms, it doesnít necessarily make it easy to devise an interface that keeps users happy on multiple platforms.

Bugs in Java Platform Implementations

One final source of variation among different implementations of the Java Platform is bugs. Although Sun has developed a comprehensive suite of tests that Java Platform implementations must pass, it is still possible that some implementations will be distributed with bugs in them. The only way you can defend yourself against this possibility is through testing. If there is a bug, you can determine through testing whether the bug affects your program, and if so, attempt to find a work-around.

Testing

Given the allowable differences between Java Platform implementations, the platform dependent ways you can potentially write a Java program, and the simple possibility of bugs in any particular Java Platform implementation, you should test your Java programs on all platforms you plan to claim it runs on. Java programs are not platform independent to a great enough extent that you only need test them on one platform. You still need to test a Java program on multiple platforms, and you should probably test it on the various Java Platform implementations that are likely to be found on each host computer you claim your program runs on. In practice, therefore, testing your Java program on the various host computers and Java Platform implementations that you plan to claim your program works on is a key factor in making your program platform independent.

Seven Steps to Platform Independence

Javaís architecture allows you to choose between platform independence and other concerns. You make your choice by the way in which you write your program. If your goal is to take advantage of platform specific features not available through the Java API, to interact with a legacy system, to use an existing library written not written in Java, or to maximize the execution speed of your program, you can use native methods to help you achieve that goal. In such cases, your programs will have reduced platform independence, and that will usually be acceptable. If, on the other hand, your goal is platform independence, then you should follow certain rules when writing your program. The following seven steps outline one path you can take to maximize your programís portability:

1. Choose a set of host computers that you will claim your program runs on (your "target hosts").

2. Choose a version of the Java Platform that you feel is well enough distributed among your target hosts. Write your program to run on this version of the Java Platform.

3. For each target host, choose a set of Java Platform implementations that you will claim your program runs on (your "target runtimes").
4. Write your program so that it accesses the host computer only through the standard runtime libraries of the Java API. (Donít invoke native methods, or use vendor-specific libraries that invoke native methods.)

5. Write your program so that it doesnít depend for correctness on timely finalization by the garbage collector or on thread prioritization.

6. Strive to design a user interface that works well on all of your target hosts.

7. Test your program on all of your target runtimes and all of your target hosts.

If you follow the seven steps outlined above, your Java program will definitely run on all your target hosts. If your target hosts cover most major Java Platform vendors on most major host computers, there is a very good chance you program will run everywhere else as well.

If you wish, you can have your program certified as "100% Pure Java." There are several reasons that you may wish to do this if you are writing a program that you intend to be platform independent. For example, if your program is certified 100% Pure, you can brand it with the 100% Pure Java coffee cup icon. You can also potentially participate in co-marketing programs with Sun. You may, however, wish to go through the certification process simply as an added check of the platform independence of your program. In this case, you have the option of just running 100% Pure verification tools you can download for free. These tools will report problems with your programís "purity" without requiring you to go through the full certification process.

The 100% Pure certification is not quite a full measure of platform independence. Part of platform independence is that userís expectations are fulfilled on multiple platforms. The 100% Pure testing process does not attempt to measure user fulfillment. It only checks to make certain your program depends only on the Java Core Platform. You could write a Java program that passes the 100% Pure tests, but still doesnít work well on all platforms from the perspective of users. Nonetheless, running your code through the 100% Pure testing process can be a worthwhile step on the road to creating a platform independent Java program.

The Politics of Platform Independence

As illustrated in Figure 2-2, Java Platform vendors are allowed to extend the standard components of the Java Platform in non-standard and platform-specific ways, but they must always support the standard components. In the future, Sun Microsystems intends to prevent the standard components of the Java Platform from splitting into several competing, slightly incompatible systems, as happened, for instance, with UNIX. The license that all Java Platform vendors must sign requires compatibility at the level of the Java Virtual Machine and the Java API, but permits differentiation in the areas of performance and extensions. There is some flexibility, as mentioned above, in the way vendors are allowed to implement threads, garbage-collection, and user interface look and feel. If things go as Sun plans, the core components of the Java Platform will remain a standard to which all vendors faithfully adhere, and the ubiquitous nature of the standard Java Platform will enable you to write programs that really are platform independent.

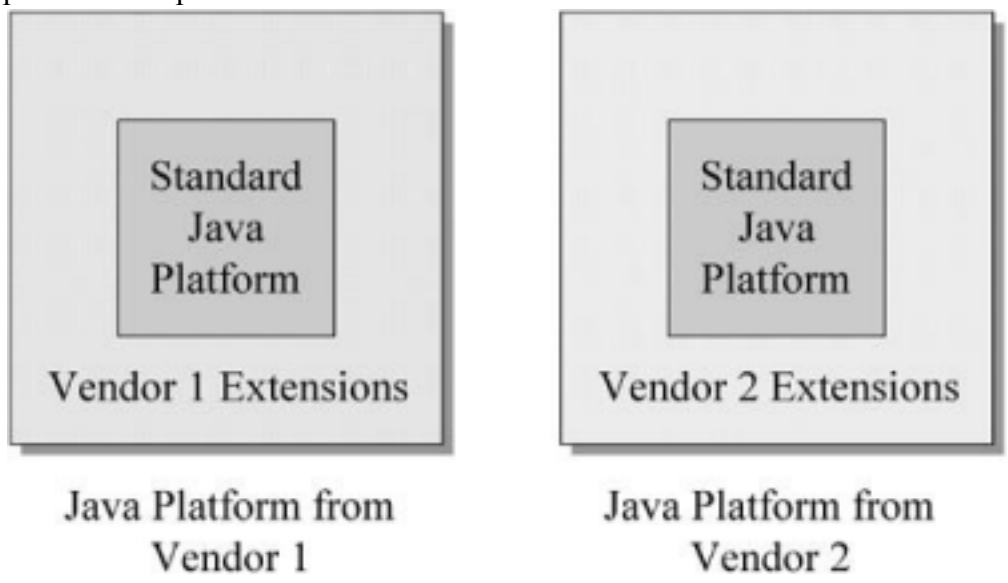


Figure 2-2. Java Platform implementations from different vendors.

You can rely on the standard components of the Java Platform because every Java Platform vendor must support them. If you write a program that only depends on these components, it should "run anywhere,"

but may suffer to some extent from the lowest-common-denominator problem. Yet because vendors are allowed to extend the Java Platform, they can give you a way to write platform-specific programs that take full advantage of the features of the underlying host operating system. The presence of both required standard components and permitted vendor extensions at any Java Platform implementation gives developers a choice. This arrangement allows developers to balance platform independence with other concerns.

There is currently a marketing battle raging for the hearts and minds of software developers over how they will write Java programs--in particular, whether or not they will choose to write platform independent or platform-specific programs. The choice that Java graciously gives to developers also potentially threatens some vested interests in the software industry.

Javaís support for platform independence threatens to weaken the "lock" enjoyed by operating system vendors. If all of your software runs on only one operating system, then your next computer will also very likely run that same operating system. You are "locked in" to one operating system vendor because your investment in software depends on an API proprietary to that vendor. You are also likely locked into one hardware architecture because the binary form of your programs requires a particular kind of microprocessor. If instead, much of your software is written to the Java API and stored as bytecodes in class files, it becomes easier for you to migrate to a different operating system vendor the next time you buy a computer. Because the Java Platform can be implemented in software on top of existing operating systems, you can switch operating systems and take all of your old platform-independent Java-based software with you.

Microsoft dominates the desktop operating system market largely because most available software runs only on Microsoft operating systems. It is in Microsoftís strategic interest to continue this status quo, so they are encouraging developers to use Java as a language to write programs that run only on Microsoft platforms. It is in just about every other operating system vendorís strategic interest to weaken Microsoftís lock on the operating system market, so the other players are encouraging developers to write Java programs that are platform independent. Sun, Netscape, IBM, and many others have banded together to promote Sunís "100% Pure Java initiative," through which they hope to educate and persuade developers to go the platform independence route.

Microsoftís approach to Java is to make Windows the best platform on which to develop and run Java programs. They want developers to use Microsoftís tools and libraries whether the developer chooses platform independence or not. Their AFC library, for example, enables Java developers to build advanced platform independent user interfaces. Microsoft also provides the Java/COM native method interface, which allows developers to use Java to write full-fledged, platform-specific Windows programs. Still, in the "spin" Microsoft gives to Java in promotional material to developers, they strongly favor the platform-specific Windows path. They extol the virtues of using Java to write programs that take full advantage of the Windows platform.

Sun and the other operating system vendors behind the 100% Pure Java initiative are attempting to counter Microsoftís spin with some of their own. The promotional material from these companies focuses on the benefits of writing platform-independent Java programs.

On one level, it is a battle between two icons. If you write your Java program Microsoftís way, you get to brand your product with a Windows 95/NT icon that displays the famous four-paneled Windows logo. If you go the 100% Pure Java route, you get to brand your product with a 100% Pure Java icon that displays the famous steaming coffee cup logo.

As a developer, the politics and propaganda swirling around the software industry need not be a major factor when you decide how to write a particular Java program. For some programs you write, platform independence may be the right approach; for others, a platform-specific program may make more sense. In each individual case, you can make a decision based on what you feel your customers want and how

you want to position yourself in the marketplace with respect to your competitors.

The Resources Page

For links to more information about Java and platform independence, visit the resources page for this chapter: http://www.artima.com/insidejvm/platindep.html

Chapter Three

Security

Aside from platform independence, discussed in the previous chapter, the other major technical challenge a network-oriented software technology must deal with is security. Networks, because they allow computers to share data and distribute processing, can potentially serve as a way to break into a computer system, enabling someone to steal information, destroy information, or steal computing resources. As a consequence, connecting a computer to a network raises many security issues.

To address the security concerns raised by networks, Javaís architecture comes with an extensive built in security model. This chapter gives an overview of the security model built into Javaís core architecture.

Why Security?

Javaís security model is one of the key architectural features that makes it an appropriate technology for networked environments. Security is important because networks represent a potential avenue of attack to any computer hooked to them. This concern becomes especially strong in an environment in which software is downloaded across the network and executed locally, as is done, for example, with Java applets. Because the class files for an applet are automatically downloaded when a user goes to the containing web page in a browser, it is likely that a user will encounter applets from untrusted sources. Without any security, this would be a convenient way to spread viruses. Thus, Javaís security mechanisms help make Java suitable for networks because they establish a needed trust in the safety of network-mobile code.

Javaís security model is focused on protecting end-users from hostile programs downloaded across a network from untrusted sources. To accomplish this goal, Java provides a customizable "sandbox" in which Java programs run. A Java program must play only inside its sandbox. It can do anything within

the boundaries of its sandbox, but canít take any action outside those boundaries. The sandbox for untrusted Java applets, for example, prohibits many activities, including:

- reading or writing to the local disk,
- making a network connection to any but the host from which the applet came,
- creating a new process, and
- loading a new dynamic library and directly calling a native method.

By making it impossible for downloaded code to perform certain actions, Javaís security model protects the end-user from the threat of hostile code.

The Sandbox

Traditionally, you had to trust software before you ran it. You achieved security by being careful only to use software from trusted sources, and by regularly scanning for viruses just to make sure. Once some software got access to your system, it had full reign. If it was malicious, it could do a great deal of damage because there were no restrictions placed on it by the runtime environment of your computer. So in the traditional security scheme, you tried to prevent malicious code from ever gaining access to your computer in the first place.

The sandbox security model makes it easier to work with software that comes from sources you donít fully trust. Instead of approaching security by requiring you to prevent any code you donít trust from ever making its way onto your computer, the sandbox model allows you to welcome code from any source. But as code from an untrusted source runs, the sandbox restricts code from untrusted sources from taking any actions that could possibly harm your system. You donít need to figure out what code you can and canít trust. You donít need to scan for viruses. The sandbox itself prevents any viruses or other malicious code you may invite into your computer from doing any damage.

If you have a properly skeptical mind, youíll need to be convinced a sandbox has no leaks before you trust it to protect you. To make sure the sandbox must has no leaks, Javaís security model involves every aspect of its architecture. If there were areas in Javaís architecture where security was weak, a malicious programmer (a "cracker") could potentially exploit those areas to "go around" the sandbox. To understand the sandbox, therefore, you must look at several different parts of Javaís architecture, and understand how they work together.

The fundamental components responsible for Javaís sandbox are:

- the class loader architecture
- the class file verifier
- safety features built into the Java Virtual Machine (and the language)
- the security manager and the Java API

One of the greatest strengths of Javaís security model is that two of these components, the class loader and the security manager, are customizable. By customizing these components, you can create a customized security policy for a Java application. As a developer, you may never need to create your own customized sandbox. You can often make use of sandboxes created by others. When you write and run a Java applet, for instance, you make use of a sandbox created by the developers of the web browser that hosts your applet.

The Class Loader Architecture

In Javaís sandbox, the class loader architecture is the first line of defense. It is the class loader, after all, that brings code into the Java Virtual Machine--code that could be hostile. The class loader architecture contributes to Javaís sandbox in two ways:

1. it guards the borders of the trusted class libraries, and

2. it prevents malicious code from interfering with benevolent code.

The class loader architecture guards the borders of the trusted class libraries by preventing an untrusted classes from pretending to be trusted. If a malicious class could successfully trick the Java Virtual Machine into believing it was a trusted class from the Java API, that malicious class could potentially break through the sandbox barrier. By preventing untrusted classes from impersonating trusted classes, the class loader architecture blocks one potential approach to compromising the security of the Java runtime.

The class loader architecture prevents malicious code from interfering with benevolent code by providing protected name-spaces for classes loaded by different class loaders. A *name-space* is a set of unique names for loaded classes that is maintained by the Java Virtual Machine. Once a Java Virtual Machine has loaded a class named `Volcano` into a particular name-space, for example, it is impossible to load a different class named `Volcano` into that same name-space. You can load multiple `Volcano` classes into a Java Virtual Machine, however, because you can create multiple name-spaces inside a Java application by creating multiple class loaders. If you create three separate name-spaces (one for each of three class loaders) in a running Java application, then, by loading one `Volcano` class into each name-space, your program could load three different `Volcano` classes into your application.

Name-spaces contribute to security because you can place a shield between classes loaded into different name-spaces. Inside the Java Virtual Machine, classes in the same name-space can interact with one another directly. Classes in different name-spaces, however, canít even detect each otherís presence unless you explicitly provide a mechanism that allows them to interact. If a malicious class, once loaded, had guaranteed access to every other class currently loaded by the virtual machine, that class could potentially learn things it shouldnít know or interfere with the proper execution of your program.

Often, a class loader object relies on other class loaders--at the very least, upon the primordial class loader--to help it fulfill some of the class load requests that come its way. For example, imagine you write a Java application that installs a class loader whose particular manner of loading class files is by downloading them across a network. Assume that during the course of running the Java application, a request is made of your class loader to load a class named `Volcano`. One way you could write the class loader is to have it first asks the primordial class loader to find and load the class from its trusted repository. In this case, since `Volcano` is not a part of the Java API, assume the primordial class loader canít find a class named `Volcano`. When the primordial class loader responds that it canít load the class, your class loader could then attempt to load the `Volcano` class in its custom manner, by downloading it across the network. Assuming your class loader was able to download class `Volcano`, that `Volcano` class could then play a role in the applicationís future course of execution.

To continue with the same example, assume that at some time later a method of class `Volcano` is invoked for the first time, and that method references class `String` from the Java API. Because it is the first time the reference was used by the running program, the virtual machine asks your class loader (the one that loaded `Volcano`) to load `String`. As before, your class loader first passes the request to the primordial class loader, but in this case, the primordial class loader is able to return a `String` class back to your class loader. (The primordial class loader most likely didnít have to actually load `String` at this point because, given that `String` is such a fundamental class in Java programs, it was almost certainly used before and therefore already loaded. Most likely, the primordial class loader just returned the `String` class that it had previously loaded from the trusted repository.) Since the primordial class loader was able to find the class, your class loader doesnít attempt to download it across the network; it merely passes to the virtual machine the `String` class returned by the primordial class loader. From that point forward, the virtual machine uses that `String` class whenever class `Volcano` references a class named `String`.

When you write a class loader, you create a new environment in which the loaded code runs. If you want the environment to be free of security holes, you must follow certain rules when you write your class loader. In general, you will want to write class loaders such that they protect the borders of trusted class libraries, such as those of the Java API.

Java allows classes in the same package to grant each other special access privileges that arenít granted to classes outside the package. So, if your class loader receives a request to load a class that by its name brazenly declares itself to be part of the Java API (for example, a class named `java.lang.Virus`), it could gain special access to the trusted classes of `java.lang` and could possibly use that special access for devious purposes. Consequently, you would normally write a class loader so that it simply refuses to load any class that claims to be part of the Java API (or any other trusted runtime library), but that doesnít exist in the local trusted repository. In other words, after your class loader passes a request to the primordial class loader, and the primordial class loader indicates it canít load the class, your class loader should check to make sure the class doesnít declare itself to be a member of a trusted package. If it does, your class loader, instead of trying to download the class across the network, should throw a security exception.

In addition, you may have installed some packages in the trusted repository that contain classes you want your application to be able to load through the primordial class loader, but that you donít want to be accessible to classes loaded through your class loader. For example, assume you have created a package named `absolutepower` and installed it on the local repository accessible by the primordial class loader. Assume also that you donít want classes loaded by your class loader to be able to load any class from the `absolutepower` package. In this case, you would write your class loader such that the very first thing it does is make sure the requested class doesnít declare itself as a member of the `absolutepower` package. If such a class is requested, your class loader, rather than passing the class name to the primordial class loader, should throw a security exception.

The only way a class loader can know whether or not a class is from a restricted package, such as `java.lang`, or a forbidden package, such as `absolutepower`, is by the classís name. Thus a class loader must be given a list of the names of restricted and forbidden packages. Because the name of class `java.lang.Virus` indicates it is from the `java.lang` package, and `java.lang` is on the list of restricted packages, your class loader should throw a security exception if the primordial class loader canít load it. Likewise, because the name of class `absolutepower.FancyClassLoader` indicates it is part of the `absolutepower` package, and the `absolutepower` package is on the list of forbidden packages, your class loader should throw a security exception absolutely.

A common way, therefore, to write a security-minded class loader is using the following four steps:

1. If packages exist that this class loader is not allowed to load from, the class loader checks whether the requested class is in one of those forbidden packages. If so, it throws a security exception. Else, it continues on to step two.

2. The class loader passes the request to the primordial class loader. If the primordial class loader successfully returns the class, the class loader returns that same class. Else, it continues on to step three.

3. If trusted packages exist that this class loader is not allowed to add classes to, the class loader checks whether the requested class is in one of those restricted packages. If so, it throws a security exception. Else, it continues on to step four.

4. Finally, the class loader attempts to load the class in the custom way, such as by downloading it across a network. If successful, it returns the class. Else, it throws a "no class definition found" error.

By performing steps one and three as outlined above, the class loader guards the borders of the trusted packages. With step one, it prevents a class from a forbidden package to be loaded at all. With step three, it doesnít allow an untrusted class to insert itself into a trusted package.

The Class File Verifier

Working in conjunction with the class loader, the class file verifier ensures that loaded class files have a proper internal structure. If the class file verifier discovers a problem with a class file, it throws an exception. Although compliant Java compilers should not generate malformed class files, a Java Virtual Machine canít tell how a particular class file was created. Because a class file is just a sequence of binary data, a virtual machine canít know whether a particular class file was generated by a well meaning Java compiler or by shady crackers bent on compromising the integrity of the virtual machine. As a consequence, all Java Virtual Machine implementations have a class file verifier that can be invoked on untrusted classes, to make sure the classes are safe to use.

One of the security goals that the class file verifier helps achieve is program robustness. If a buggy compiler or savvy cracker generated a class file that contained a method whose bytecodes included an instruction to jump beyond the end of the method, that method could, if it were invoked, cause the virtual machine to crash. Thus, for the sake of robustness, it is important that the virtual machine verify the integrity of the bytecodes it imports. Although Java Virtual Machine designers are allowed to decide when their virtual machines will perform these checks, many implementations will do most checking just after a class is loaded. Such a virtual machine, rather than checking every time it encounters a jump instruction as it executes bytecodes, analyzes bytecodes (and verifies their integrity) once, before they are ever executed. As part of its verification of bytecodes, the Java Virtual Machine makes sure all jump instructions cause a jump to another valid instruction in the bytecode stream of the method. In most cases, checking all bytecodes once, before they are executed, is a more efficient way to guarantee robustness than checking every bytecode instruction every time it is executed.

A class file verifier that performs its checking as early as possible most likely operates in two distinct phases. During phase one, which takes place just after a class is loaded, the class file verifier checks the internal structure of the class file, including verifying the integrity of the bytecodes it contains. During phase two, which takes place as bytecodes are executed, the class file verifier confirms the existence of symbolically referenced classes, fields, and methods.

Phase One: Internal Checks

During phase one, the class file verifier checks everything thatís possible to check in a class file by looking at only the class file itself. In addition to verifying the integrity of the bytecodes during phase one, the verifier performs many checks for proper class file format and internal consistency. For example, every class file must start with the same four bytes, the magic number: `0xCAFEBABE`. The purpose of magic numbers is to make it easy for file parsers to recognize a certain type of file. Thus, the first thing a class file verifier likely checks is that the imported file does indeed begin with `0xCAFEBABE`.

The class file verifier also checks to make sure the class file is neither truncated nor enhanced with extra trailing bytes. Although different class files can be different lengths, each individual component contained inside a class file indicates its length as well as its type. The verifier can use the component types and lengths to determine the correct total length for each individual class file. In this way, it can verify that the imported file has a length consistent with its internal contents.

The verifier also looks at individual components, to make sure they are well-formed instances of their type of component. For example, a method descriptor (its return type and the number and types of its

parameters) is stored in the class file as a string that must adhere to a certain context-free grammar. One check the verifier performs on individual components is to make sure each method descriptor is a well formed string of the appropriate grammar.

In addition, the class file verifier checks that the class itself adheres to certain constraints placed upon it by the specification of the Java programming language. For example, the verifier enforces the rule that all classes, except class `Object`, must have a superclass. Thus, the class file verifier checks at run-time some of the Java language rules that should have been enforced at compile-time. Because the verifier has no way of knowing if the class file was generated by a benevolent, bug-free compiler, it checks each class file to make sure the rules are followed.

Once the class file verifier has successfully completed the checks for proper format and internal consistency, it turns its attention to the bytecodes. During this part of phase one, which is commonly called the "bytecode verifier," the Java Virtual Machine performs a data-flow analysis on the streams of bytecodes that represent the methods of the class. To understand the bytecode verifier, you need to understand a bit about bytecodes and frames.

The bytecode streams that represent Java methods are a series of one-byte instructions, called *opcodes*, each of which may be followed by one or more *operands*. The operands supply extra data needed by the Java Virtual Machine to execute the opcode instruction. The activity of executing bytecodes, one opcode after another, constitutes a thread of execution inside the Java Virtual Machine. Each thread is awarded its own *Java Stack*, which is made up of discrete *frames*. Each method invocation gets its own frame, a section of memory where it stores, among other things, local variables and intermediate results of computation. The part of the frame in which a method stores intermediate results is called the methodís *operand stack*. An opcode and its (optional) operands may refer to the data stored on the operand stack or in the local variables of the methodís frame. Thus, the virtual machine may use data on the operand stack, in the local variables, or both, in addition to any data stored as operands following an opcode when it executes the opcode.

The bytecode verifier does a great deal of checking. It checks to make sure that no matter what path of execution is taken to get to a certain opcode in the bytecode stream, the operand stack always contains the same number and types of items. It checks to make sure no local variable is accessed before it is known to contain a proper value. It checks that fields of the class are always assigned values of the proper type, and that methods of the class are always invoked with the correct number and types of arguments. The bytecode verifier also checks to make sure that each opcode is valid, that each opcode has valid operands, and that for each opcode, values of the proper type are in the local variables and on the operand stack. These are just a few of the many checks performed by the bytecode verifier, which is able, through all its checking, to verify that a stream of bytecodes is safe for the Java Virtual Machine to execute.

Phase one of the class file verifier makes sure the imported class file is properly formed, internally consistent, adheres to the constraints of the Java programming language, and contains bytecodes that will be safe for the Java Virtual Machine to execute. If the class file verifier finds that any of these are not true, it throws an error, and the class file is never used by the program.

Phase Two: Verification of Symbolic References

Although phase one happens immediately after the Java Virtual Machine loads a class file, phase two is delayed until the bytecodes contained in the class file are actually executed. During phase two, the Java Virtual Machine follows the references from the class file being verified to the referenced class files, to make sure the references are correct. Because phase two has to look at other classes external to the class file being checked, phase two may require that new classes be loaded. Most Java Virtual Machine implementations will likely delay loading classes until they are actually used by the program. If an implementation does load classes earlier, perhaps in an attempt to speed up the loading process, then it

must still give the impression that it is loading classes as late as possible. If, for example, a Java Virtual Machine discovers during early loading that it canít find a certain referenced class, it doesnít throw a "class definition not found" error until (and unless) the referenced class is used for the first time by the running program. Therefore, phase two, the checking of symbolic references, is usually delayed until each symbolic reference is actually used for the first time during bytecode execution.

Phase two of class file verification is really just part of the process of dynamic linking. When a class file is loaded, it contains symbolic references to other classes and their fields and methods. A symbolic reference is a character string that gives the name and possibly other information about the referenced item--enough information to uniquely identify a class, field, or method. Thus, symbolic references to other classes give the full name of the class; symbolic references to the fields of other classes give the class name, field name, and field descriptor; symbolic references to the methods of other classes give the class name, method name, and method descriptor.

Dynamic linking is the process of *resolving* symbolic references into direct references. As the Java Virtual Machine executes bytecodes and encounters an opcode that, for the first time, uses a symbolic reference to another class, the virtual machine must resolve the symbolic reference. The virtual machine performs two basic tasks during resolution:

1. find the class being referenced (loading it if necessary)

2. replace the symbolic reference with a direct reference, such as a pointer or offset, to the class, field, or method

The virtual machine remembers the direct reference so that if it encounters the same reference again later, it can immediately use the direct reference without needing to spend time resolving the symbolic reference again.

When the Java Virtual Machine resolves a symbolic reference, phase two of the class file verifier makes sure the reference is valid. If the reference is not valid--for instance, if the class cannot be loaded or if the class exists but doesnít contain the referenced field or method--the class file verifier throws an error.

As an example, consider again the `Volcano` class. If a method of class `Volcano` invokes a method in a class named `Lava`, the name and descriptor of the method in `Lava` are included as part of the binary data in the class file for `Volcano`. So, during the course of execution when the `Volcano`ís method first invokes the `Lava`ís method, the Java Virtual Machine makes sure a method exists in class `Lava` that has a name and descriptor that matches those expected by class `Volcano`. If the symbolic reference (class name, method name and descriptor) is correct, the virtual machine replaces it with a direct reference, such as a pointer, which it will use from then on. But if the symbolic reference from class `Volcano` doesnít match any method in class `Lava`, phase two verification fails, and the Java Virtual Machine throws a "no such method" error.

Binary Compatibility

The reason phase two of the class file verifier must look at classes that refer to one

nother to make sure they are compatible is because Java programs are dynamically linked. Java compilers will often recompile classes that depend on a class you have changed, and in so doing, detect any incompatibility at compile-time. But there may be times when your compiler doesnít recompile a dependent class. For example, if you are developing a large system, you will likely partition the various parts of the system into packages. If you compile each package separately, then a change to one class in a package would cause a recompilation of affected classes within that same package, but not necessarily in any other package. Moreover, if you are using someone elseís packages, especially if your program downloads class files from someone elseís package across a network as it runs, it may be impossible for

you to check for compatibility at compile-time. Thatís why phase two of the class file verifier must check for compatibility at run-time.

As an example of incompatible changes, imagine you compiled class `Volcano` (from the above example) with a Java compiler. Because a method in `Volcano` invokes a method in another class named `Lava`, the Java compiler would look for a class file or a source file for class `Lava` to make sure there was a method in `Lava` with the appropriate name, return type, and number and types of arguments. If the compiler couldnít find any `Lava` class, or if it encountered a `Lava` class that didnít contain the desired method, the compiler would generate an error and would not create a class file for `Volcano`. Otherwise, the Java compiler would produce a class file for `Volcano` that is compatible with the class file for `Lava`. In this case, the Java compiler refused to generate a class file for `Volcano` that wasnít already compatible with class `Lava`.

The converse, however, is not necessarily true. The Java compiler could conceivably generate a class file for `Lava` that isnít compatible with `Volcano`. If the `Lava` class doesnít refer to `Volcano`, you could potentially change the name of the method `Volcano` invokes from the `Lava` class, and then recompile only the `Lava` class. If you tried to run your program using the new version of `Lava`, but still using the old version of `Volcano` that wasnít recompiled since you made your change to `Lava`, the Java Virtual Machine would, as a result of phase two class file verification, throw a "no such method" error when `Volcano` attempted to invoke the now non-existent method in `Lava`.

In this case, the change to class `Lava` broke *binary compatibility* with the pre-existing class file for `Volcano`. In practice, this situation may arise when you update a library you have been using, and your existing code isnít compatible with the new version of the library. To make it easier to alter the code for libraries, the Java programming language was designed to allow you to make many kinds of changes to a class that donít require recompilation of classes that depend upon it. The changes you are allowed to make, which are listed in the Java Language Specification, are called the rules of binary compatibility. These rules clearly define what can be changed, added, or deleted in a class without breaking binary compatibility with pre-existing class files that depend on the changed class. For example, it is always a binary compatible change to add a new method to a class, but never to delete a method that other classes may be using. So in the case of `Lava`, you violated the rules of binary compatibility when you changed the name of the method used by `Volcano`, because you in effect deleted the old method and added a new. If you had, instead, added the new method and then rewritten the old method so it calls the new, that change would have been binary compatible with any pre-existing class file that already used `Lava`, including `Volcano`.

Safety Features Built Into the Java Virtual Machine

Once the Java Virtual Machine has loaded a class and performed phase one of class file verification, the bytecodes are ready to be executed. Besides the verification of symbolic references (phase two of class file verification), the Java Virtual Machine has several other built-in security mechanisms operating as bytecodes are executed. These are the same mechanisms listed in Chapter 1 as features of the Java programming language that make Java programs robust. They are, not surprisingly, also features of the Java Virtual Machine:

- type-safe reference casting
- structured memory access (no pointer arithmetic)
- automatic garbage collection (canít explicitly free allocated memory)
- array bounds checking
- checking references for `null`

By granting a Java program only safe, structured ways to access memory, the Java Virtual Machine makes Java programs more robust, but it also makes their execution more secure. Why? There are two reasons. First, a program that corrupts memory, crashes, and possibly causes other programs to crash

represents one kind of security breach. If you are running a mission critical server process, it is critical that the process doesnít crash. This level of robustness is also important in embedded systems, such as a cell phone, which people donít usually expect to have to reboot. The second reason unrestrained memory access would be a security risk is because a wily cracker could potentially use it to subvert the security system. If, for example, a cracker could learn where in memory a class loader is stored, it could assign a pointer to that memory and manipulate the class loaderís data. By enforcing structured access to memory, the Java Virtual Machine yields programs that are robust, but also frustrates crackers who dream of harnessing the internal memory of the Java Virtual Machine for their own devious plots.

Another safety feature built into the Java Virtual Machine--one that serves as a backup to structured memory access--is the unspecified manner in which the runtime data areas are laid out inside the Java Virtual Machine. The *runtime data areas* are the memory areas in which the Java Virtual Machine stores the data it needs to execute a Java application: Java stacks (one for each thread), a *method area*, where bytecodes are stored, and a *garbage-collected heap*, where the objects created by the running program are stored. If you peer into a class file, you wonít find any memory addresses. When the Java Virtual Machine loads a class file, it decides where in its internal memory to put the bytecodes and other data it parses from the class file. When the Java Virtual Machine starts a thread, it decides where to put the Java stack it creates for the thread. When it creates a new object, it decides where in memory to put the object. Thus, a cracker cannot predict by looking at a class file where in memory the data representing that class, or objects instantiated from that class, will be kept. Whatís worse (for the cracker) is the cracker canít tell anything about memory layout by reading the Java Virtual Machine specification either. The manner in which a Java Virtual Machine lays out its internal data is not part of the specification. The designers of each Java Virtual Machine implementation decide which data structures their implementation will use to represent the runtime data areas, and where in memory their implementation will place them. As a result, even if a cracker were somehow able to break through the Java Virtual Machineís memory access restrictions, they would next be faced with the difficult task of finding something to subvert by looking around.

The prohibition on unstructured memory access is not something the Java Virtual Machine must actively enforce on a running program; rather, it is intrinsic to the bytecode instruction set itself. Just as there is no way to express an unstructured memory access in the Java programming language, there is also no way to express it in bytecodes--even if you write the bytecodes by hand. Thus, the prohibition on unstructured memory access is a solid barrier against the malicious manipulation of memory.

There is, however, a way to penetrate the security barriers erected by the Java Virtual Machine. Although the bytecode instruction set doesnít give you an unsafe, unstructured way to access memory, there is a way you can go around bytecodes: native methods. Basically, when you call a native method, Javaís security sandbox becomes dust in the wind. First of all, the robustness guarantees donít hold for native methods. Although you canít corrupt memory from a Java method, you can from a native method. But most importantly, native methods donít go through the Java API (they are how you go around the Java API) so the security manager isnít checked before a native method attempts to do something that could be potentially damaging. (This is, of course, often how the Java API itself gets anything done. But the native methods used by the Java API are "trusted.") Thus, once a thread gets into a native method, no matter what security policy was established inside the Java Virtual Machine, it doesnít apply anymore to that thread, so long as that thread continues to execute the native method. This is why the security manager includes a method that establishes whether or not a program can load dynamic libraries, which are necessary for invoking native methods. Applets, for example, arenít allowed to load a new dynamic library, therefore they canít install their own new native methods. They can, however, call methods in the Java API, methods which may be native, but which are always trusted. When a thread invokes a native method, that thread leaps outside the sandbox. The security model for native methods is, therefore, the same security model described earlier as the traditional approach to computer security: you have to trust a native method before you call it.

One final mechanism that is built into the Java Virtual Machine that contributes to security is structured

error handling with exceptions. Because of its support for exceptions, the Java Virtual Machine has something structured to do when a security violation occurs. Instead of crashing, the Java Virtual Machine can throw an exception or an error, which may result in the death of the offending thread, but shouldnít crash the system. Throwing an error (as opposed to throwing an exception) almost always results in the death of the thread in which the error was thrown. This is usually a major inconvenience to a running Java program, but wonít necessarily result in termination of the entire program. If the program has other threads doing useful things, those threads may be able to carry on without their recently departed colleague. Throwing an exception, on the other hand, may result in the death of the thread, but is often just used as a way to transfer control from the point in the program where the exception condition arose to the point in the program where the exception condition is handled.

The Security Manager and the Java API

By using class loaders, you can prevent code loaded by different class loaders from interfering with one another inside the Java Virtual Machine, but to protect assets external to the Java Virtual Machine, you must use a security manager. The security manager defines the outer boundaries of the sandbox. Because it is customizable, the security manager allows you to establish a custom security policy for an application. The Java API enforces the custom security policy by asking the security manager for permission before it takes any action that is potentially unsafe. For each potentially unsafe action, there is a method in the security manager that defines whether that action is allowed by the sandbox. Each methodís name starts with "check," so for example, `checkRead()` defines whether or not a thread is allowed to read to a specified file, and `checkWrite()` defines whether or not a thread is allowed to write to a specified file. The implementation of these methods is what defines the custom security policy of the application.

Most of the activities that are regulated by a "check" method are listed below. The classes of the Java API check with the security manager before they:

- accept a socket connection from a specified host and port number
- modify a thread (change its priority, stop it, etcÖ)
- open a socket connection to a specified host and port number
- create a new class loader
- delete a specified file
- create a new process
- cause the application to exit
- load a dynamic library that contains native methods
- wait for a connection on a specified local port number
- load a class from a specified package (used by class loaders)
- add a new class to a specified package (used by class loaders)
- access or modify system properties
- access a specified system property
- read from a specified file
- write to a specified file

Because the Java API always checks with the security manager before it performs any of the activities listed above, the Java API will not perform any action forbidden under the security policy established by the security manager.

Two actions not present in the above list that could potentially be unsafe are allocation of memory and invocation of threads. Currently, a hostile applet that can possibly crash the browser by:

- allocating memory until it runs out
- firing off threads until everything slows to a crawl

These kinds of attacks are called *denial of service*, because the deny the end-users from using their own computers. The security manager does not allow you to enforce any kind of limit on allocated memory or thread creation. (There are no `checkAllocateMemory()` or `checkCreateThread()` methods in the security manager class.) The difficulty in attempting to thwart this kind of hostile code is that it is hard to tell the difference, for example, between a hostile applet allocating a lot of memory and an image processing applet attempting to do useful work. Other kinds of hostile applets that are currently possible are:

- applets that send unauthorized e-mail from the end-userís computer
- applets that make annoying noises even after you leave the web page
- applets that display offensive images or animations

So a security manager isnít enough to prevent every possible action that could possibly offend or inconvenience an end-user. Other than the attacks listed here, however, the security manager attempts to provide a check method that allows you to control access to any potentially unsafe action.

When a Java application starts, it has no security manager, but the application can install one at its option. If it does not install a security manager, there are no restrictions placed on any activities requested of the Java API--the Java API will do whatever it is asked. (This is why Java applications, by default, do not have any security restrictions such as those that limit the activities of untrusted applets.) If the application does install a security manager, then that security manager will be in charge for the entire remainder of the lifetime of that application. It canít be replaced, extended, or changed. From that point on, the Java API will only fulfill those requests that are sanctioned by the security manager.

In general, a "check" method of the security manager throws a security exception if the checked upon activity is forbidden, and simply returns if the activity is permitted. Therefore, the procedure a Java API method generally follows when it is about to perform a potentially unsafe activity involves two steps. First, the Java API code checks whether a security manager has been installed. If not, it skips step two and goes ahead with the potentially unsafe action. Otherwise, as step two, it calls the appropriate "check" method in the security manager. If the action is forbidden, the "check" method will throw a security exception, which will cause the Java API method to immediately abort. The potentially unsafe action will never be taken. If, on the other hand, the action is permitted, the "check" method will simply return. In this case, the Java API method carries on and performs the potentially unsafe action.

Although you can only install one security manager, you can write the security manager so that it establishes multiple security policies. In addition to the "check" methods, the security manager also has methods that allow you to determine if a request is being made either directly or indirectly from a class loaded by a class loader object, and if so, which class loader object. This enables you to implement a security policy that varies depending upon which class loader loaded the classes making the request. You can also vary the security policy based on information about the class files loaded by the class loader, such whether or not the class files were down loaded across a network or imported from the local disk. So even though an application can only have one security manager, that security manager can establish a flexible security policy that varies depending upon the trustworthiness of the code requesting the potentially unsafe action.

Authentication

The support for authentication introduced in Java 1.1 in the `java.security` package expands your ability to establish multiple security policies by enabling you to implement a sandbox that varies depending upon who actually created the code. Authentication allows you to verify that a set of class files was blessed as trustworthy by some vendor, and that the class files were not altered en route to your virtual machine. Thus, to the extent you trust the vendor, you can ease the restrictions placed on the code by the sandbox. You can establish different security policies for code that comes from different vendors.

For links to more information about authentication and `java.security`, visit the resources page for this chapter.

Security Beyond the Architecture

Security is a tradeoff between cost and risk: the lower the security risk, the higher the cost of security. The costs associated with any computer or network security strategy must be weighed against the costs that would be associated with the theft or destruction of the information or computing resources being protected. The nature of a computer or network security strategy should be shaped by the value of the assets being protected.

To be effective, a computer or network security strategy must be comprehensive. It cannot consist exclusively of a sandbox for running downloaded Java code. For instance, it may not matter much that the Java applets you download from the internet and run on your computer canít read the word processing file of your top-secret business plan if you:

- routinely download untrusted native executables from the internet and run them
- throw away extra printed copies of your business plan without shredding them
- leave your doors unlocked when youíre gone
- hire someone to help you who is actually a spy for your arch-rival

In the context of a comprehensive security strategy, however, Javaís security model can play a useful role.
The nice thing about Javaís security model is that once you set it up, it does most of the work for you. You donít have to worry about whether a particular program is trusted or not--the Java runtime will determine that for you; and if it is untrusted, the Java runtime will protect your assets by encasing the untrusted code in a sandbox.

End-users of Java software cannot rely only on the security mechanisms built into Javaís architecture. They must have a comprehensive security policy appropriate to their actual security requirements.

Similarly, the security strategy of Java technology itself does not rely exclusively on the architectural security mechanisms described in this chapter. For example, one aspect of Javaís security strategy is that anyone can sign a license agreement and get a copy of the source code of Sunís Java Platform implementation. Instead of keeping the internal implementation of Javaís security architecture a secret "black box," it is open to anyone who wishes to look at it. This encourages security experts seeking a good technical challenge to try and find security holes in the implementation. When security holes are discovered, they can be patched. Thus, the openness of Javaís internal implementation is part of Javaís overall security strategy.

Besides openness, there are several other aspects to Javaís overall security strategy that donít directly involve its architecture. You can find out more information about these on the resources page for this chapter.

The Resources Page

For more information about Java and security, see the resource page for this chapter:
http://www.artima.com/insidejvm/security.html


McGraw-Hill
A Division of The McGraw-Hill Companies


Chapter Four

Network mobility

The previous two chapters discussed how Javaís architecture deals with the two major challenges presented to software developers by a networked computing environment. Platform independence deals with the challenge that many different kinds of computers and devices are usually connected to the same network. The sandbox security model deals with the challenge that networks represent a convenient way to transmit viruses and other forms of malicious code. This chapter describes not how Javaís architecture deals with a challenge, but how it seizes an opportunity made possible by the network.

One of the fundamental reasons Java is a useful tool for networked software environments is that Javaís architecture enables the network mobility of software. In fact, it was primarily this aspect of Java technology that was considered by many in the software industry to represent a paradigm shift. This chapter examines the nature of this new paradigm of network-mobile software, and how Javaís architecture makes it possible.

Why Network Mobility?

Prior to the advent of the personal computer, the dominant computing model was the large mainframe computer serving multiple users. By time-sharing, a mainframe computer divided its attention among several users, who logged onto the mainframe at dumb terminals. Software applications were stored on disks attached to the mainframe computer, allowing multiple users to share the same applications while they shared the same CPU. A drawback of this model was that if one user ran a CPU-intensive job, all other users would experience degraded performance.

The appearance of the microprocessor led to the proliferation of the personal computer. This change in the hardware status quo changed the software paradigm as well. Rather than sharing software applications stored at a mainframe computer, individual users had individual copies of software applications stored at each personal computer. Because each user ran software on a dedicated CPU, this new model of computing addressed the difficulties of dividing CPU-time among many users attempting to share one mainframe CPU.

Initially, personal computers operated as unconnected islands of computing. The dominant software model was of isolated executables running on isolated personal computers. But soon, personal

computers began to be connected to networks. Because a personal computer gave its user undivided attention, it addressed the CPU-time sharing difficulties of mainframes. But unless personal computers were connected to a network, they couldnít replicate the mainframeís ability to let multiple users view and manipulate a central repository of data.
As personal computers connected to networks became the norm, another software model began to increase in importance: client/server. The client/server model divided work between two processes running on two different computers: a client process ran on the end-userís personal computer, and a server process ran on some other computer hooked to the same network. The client and server processes communicated with one another by sending data back and forth across the network. The server process often simply accepted data query commands from clients across the network, retrieved the requested data from a central database, and sent the retrieved data back across the network to the client. Upon receiving the data, the client processed, displayed, and allowed the user to manipulate the data. This model allowed users of personal computers to view and manipulate data stored at a central repository, while not forcing them to share a central CPU for all of the processing of that data. Users did share the CPU running the server process, but to the extent that data processing was performed by the clients, the burden on the central CPU hosting the server process was lessened.

The client/server architecture was soon extended to include more than two processes. The original client/server model began to be called 2-tier client/server, to indicate two processes: one client and one server. More elaborate architectures were called 3-tier, to indicate three processes, 4-tier, to indicate four processes, or N-tier, to indicate people were getting tired of counting processes. Eventually, as more processes became involved, the distinction between client and server blurred, and people just started using the term *distributed processing* to encompass all of these schemes.

The distributed processing model leveraged the network and the proliferation of processors by dividing processing work loads among many processors while allowing those processors to share data. Although this model had many advantages over the mainframe model, there was one notable disadvantage: distributed processing systems were more difficult to administer than mainframe systems. On mainframe systems, software applications were stored on a disk attached to the mainframe. Even though an application could serve many users, it only needed to be installed and maintained in one place. When an application was upgraded, all users got the new version the next time they logged on and started the application. By contrast, the software executables for different components of a distributed processing system were usually stored on many different disks. In a client/server architecture, for example, each computer that hosted a client process usually had its own copy of the client software stored on its local disk. As a result, a system administrator had to install and maintain the various components of a distributed software system in many different places. When a software component was upgraded, the system administrator had to physically upgrade each copy of the component on each computer that hosted it. As a result, system administration was more difficult for the distributed processing model than for the mainframe model.

The arrival of Java, with an architecture that enabled the network-mobility of software, heralded yet another model for computing. Building on the prevailing distributed processing model, the new model added the automatic delivery of software across networks to computers that ran the software. This addressed the difficulties involved in system administration of distributed processing systems. For example, in a client/server system, client software could be stored at one central computer attached to the network. Whenever an end-user needed to use the client software, the binary executable would be sent from the central computer across the network to the end-userís computer, where the software would run.

So network-mobility of software represented another step in the evolution of the computing model. In particular, it addressed the difficulty of administering a distributed processing system. It simplified the job of distributing any software that was to be used on more than one CPU. It allowed data to be delivered together with the software that knows how to manipulate or display the data. Because code was sent along with data, end-users would always have the most up-to-date version of the code. Thus,

because of network-mobility, software can be administered from a central computer, reminiscent of the mainframe model, but processing can still be distributed among many CPUs.

A New Software Paradigm

The shift away from the mainframe model towards the distributed processing model was a consequence of the personal computer revolution, which was made possible by the rapidly increasing capabilities and decreasing costs of processors. Similarly, lurking underneath the latest software paradigm shift towards distributed processing with network-mobile code is another hardware trend--the increasing capabilities and decreasing costs of network bandwidth. As bandwidth, the amount of information that can be carried by a network, increases, it becomes practical to send new kinds of information across a network; and with each new kind of information a network carries, the network takes on a new character. Thus, as bandwidth grows, simple text sent across a network can become enhanced with graphics, and the network begins to take on an appearance reminiscent of newspapers or magazines. Once bandwidth expands enough to support live streams of audio data, the network begins to act like a radio, a CD player, or a telephone. With still more bandwidth, video becomes possible, resulting in a network that competes with TV and VCRs for the attention of couch potatoes. But there is still one other kind of bandwidth-hungry content that becomes increasingly practical as bandwidth improves: computer software. Because networks by definition interconnect processors, one processor can, given enough bandwidth, send code across a network for another processor to execute. Once networks begin to move software as well as data, the network begins to look like a computer in its own right.

As software begins to travel across networks, not only does the network begin to take on a new character, but so does the software itself. Network-mobile code makes it easier to ensure that an end user has the necessary software to view or manipulate some data sent across the network, because the software can be sent along with the data. In the old model, software executables from a local disk were invoked to view data that came across the network, thus the software application was usually a distinct entity, easily discernible from the data. In the new model, because software and data are both sent across the network, the distinction between software and data is not as stark--software and data blur together to become "content."

As the nature of software evolves, the end-userís relationship to software evolves as well. Prior to network-mobility, an end-user had to think in terms of software applications and version numbers. Software was generally distributed on media such as tapes, floppy disks, or CD-ROMs. To use an application, an end-user had to get the installation media, physically insert them into a drive or reader attached to the computer, and run an installation program that copied files from the installation media to the computerís hard disk. Moreover, the end-user often did this process multiple times for each application, because software applications were routinely replaced by new versions that fixed old bugs and added new features (and usually added new bugs too). When a new version was released, end-users had to decide whether or not to upgrade. If an end-user decided to upgrade, the installation process had to repeated. Thus, end-users had to think in terms of software applications and version numbers, and take deliberate action to keep their software applications up-to-date.

In the new model, end-users think less in terms of software applications with discrete versions, and more in terms of self-evolving "content services." Whereas installing a traditional software application or an upgrade was a deliberate act on the part of the end-user, network-mobility of software enables installation and upgrading that is more automatic. Network-delivered software need not have discrete version numbers that are known to the end-user. The end-user need not decide whether to upgrade, and need not take any special action to upgrade. Network-delivered software can just evolve of its own accord. Instead of buying discrete versions of a software application, end-users can subscribe to a content service--software that is delivered across a network along with relevant data--and watch as both the software and data evolve automatically over time.

Once you move away from delivering software in discrete versions towards delivering software as self

evolving streams of interactive content, your end-user loses some control. In the old model, if a new version appeared that had serious bugs, an end-user could simply opt not to upgrade. But in the new model, an end-user canít necessarily wait until the bugs are worked out of a new version before upgrading to the new version, because the end-user may have no control over the upgrading process.

For certain kinds of products, especially those that are large and full-featured, end-users may prefer to retain control over whether and when to upgrade. Consequently, in some situations software vendors may publish discrete versions of a content service over the network. At the very least, a vendor can publish two branches of a service: a beta branch and a released branch. End-users that want to stay on the bleeding edge can subscribe to the beta service, and the rest can subscribe to the released service that, although it may not have all the newest features, is likely more robust.

Yet for many content services, especially simple ones, most end-users wonít want to have to worry about versions, because worrying about versions makes software harder to use. The end-user has to have knowledge about the differences between versions, make decisions about when and if to upgrade, and take deliberate action to cause an upgrade. Content services that are not chopped up into discrete versions are easier to use, because they evolve automatically. Such a content service, because the end user doesnít have to maintain it but can just simply use it, takes on the feel of a "software appliance."

Many self-evolving content services will share two fundamental characteristics with common household appliances: a focused functionality and a simple user-interface. Consider the toaster. A toasterís functionality is focused exclusively on the job of preparing toast, and it has a simple user-interface. When you walk up to a toaster, you donít expect to have to read a manual. You expect to put the bread in at the top and push down a handle until it clicks. You expect to be able to peer in and see orange wires glowing, and after a moment, to hear that satisfying pop and see your bread transformed into toast. If the result is too light or too dark, you expect to be able to slide a knob to indicate to the toaster that the next time, you want your toast a bit darker or lighter. Thatís the extent of the functionality and user interface of a toaster. Likewise, the functionality of many content services will be as focused and the user-interface will be as simple. If you want to order a movie through the network, for example, you donít want to worry whether you have the correct version of movie-ordering software. You donít want to have to install it. You just want to switch on the movie-ordering content service, and through a simple user-interface, order your movie. Then you can sit back and enjoy your network-delivered movie as you eat your toast.

A good example of a content service is a World Wide Web page. If you look at an HTML file, it looks like a source file for some kind of program. But if you see the browser as the program, the HTML file looks more like data. Thus, the distinction between code and data is blurred. Also, people who browse the World Wide Web expect web pages to evolve over time, without any deliberate action on their part. They donít expect to see discrete version numbers for web pages. They donít expect to have to do anything to upgrade to the latest version of a page besides simply revisiting the page in their browser.

In the coming years, many of todayís media may to some extent be assimilated by the network and transformed into content services. (As with the Borg from Star Trek, resistance is futile.) Broadcast radio, broadcast and cable TV, telephones, answering machines, faxes, video rental stores, newspapers, magazines, books, computer software--all of these will be affected by the proliferation of networks. But just as TV didnít supplant radio entirely, content services will not entirely supplant existing media. Instead, content services will likely take over some aspects of existing media, leaving the existing media to adjust accordingly, and create some new forms that didnít previously exist.
In the computer software domain, the content service model will not completely replace the old models either. Instead, it will likely take over certain aspects of the old models that fit better in the new model, add new forms that didnít exist before, and leave the old models to adjust their focus slightly in light of the newcomer.

This book is an example of how the network can affect existing media. The book was not entirely

replaced by a content service counterpart, but instead of including resource pointers (sources where you can find further information on topics presented in the book) as part of the book, they were placed on a web page. Because resource pointers change so often, it made sense to let the network assimilate that part of the book. Thus, the resource pointers portion of the book has become a content service.

The crux of the new software paradigm, therefore, is that software begins to act more like appliances. End-users no longer have to worry about installation, version numbers, or upgrading. As code is sent along with data across the network, software delivery and updating become automatic. In this way, simply by making code mobile, Java unleashes a whole new way to think about software development, delivery, and use.

Javaís Architectural Support for Network-Mobility

Javaís architectural support for network-mobility begins with its support for platform independence and security. Although they are not strictly required for network-mobility, platform independence and security help make network-mobility practical. Platform independence makes it easier to deliver a program across the network because you donít have to maintain a separate version of the program for different platforms, and you donít have to figure out how to get the right version to each computer. One version of a program can serve all computers. Javaís security features help promote network-mobility because they give end-users confidence to download class files from untrusted sources. In practice, therefore, Javaís architectural support for platform independence and security facilitate the network mobility of its class files.

Beyond platform independence and security, Javaís architectural support for network-mobility is focused on managing the time it takes to move software across a network. If you store a program on a server and download it across a network when you need it, it will likely take longer for your program to start than if you had started the same program from a local disk. Thus, one of the primary issues of network-mobile software is the time it takes to send a program across a network. Javaís architecture addresses this issue by rejecting the traditional monolithic binary executable in favor of small binary pieces: Java class files. Class files can travel across networks independently, and because Java programs are dynamically linked and dynamically extensible, an end-user neednít wait until all of a programís class files are downloaded before the program starts. The program starts when the first class file arrives. Class files themselves are designed to be compact, so that they fly more quickly across networks. Therefore, the main way Javaís architecture facilitates network-mobility directly is by breaking up the monolithic binary executable into compact class files, which can be loaded as needed.

The execution of a Java application begins at a `main()` method of some class, and other classes are loaded and dynamically linked as they are needed by the application. If a class is never actually used during one session, that class wonít ever be loaded during that session. For example, if you are using a word processor that has a spelling checker, but during one session you never invoke the spelling checker, the class files for the spelling checker will not be loaded during that session.

In addition to dynamic linking, Javaís architecture also enables dynamic extension. Dynamic extension is another way the loading of class files (and the downloading of them across a network) can be delayed in a Java application. Using class loader objects, a Java program can load extra classes at run-time, which then become a part of the running program. Therefore, dynamic linking and dynamic extension give a Java programmer some flexibility in designing when class files for a program are loaded, and as a result, how much time an end-user must spend waiting for class files to come across the network.

Besides dynamic linking and dynamic extension, another way Javaís architecture directly supports network mobility is through the class file format itself. To reduce the time it takes to send them across networks, class files are designed to be compact. In particular, the bytecode streams they contain are designed to be compact. They are called "bytecodes" because each instruction occupies only one byte. With only two exceptions, all opcodes and their ensuing operands are byte aligned to make the bytecode

streams smaller. The two exceptions are opcodes that may have one to three bytes of padding after the opcode and before the start of the operands, so that the operands are aligned on word boundaries. Other than the two opcodes that may have a small amount of padding, all data in a class file is byte aligned.

One of the implications of the compactness goal for class files is that Java compilers are not likely to do any local optimization. Because of binary compatibility rules, Java compilers canít perform global optimizations such as inlining the invocation of another classís method. (Inlining means replacing the method invocation with the code performed by the method, which saves the time it takes to invoke and return from the method as the code executes.) Binary compatibility requires that a methodís implementation can be changed without breaking compatibility with pre-existing class files that depend on the method. Inlining could be performed in some circumstances on methods within a single class, but in general that kind of optimization is not done by Java compilers, partly because it goes against the grain of class file compactness. Optimizations are often a tradeoff between execution speed and code size. Therefore, Java compilers generally leave optimization up to the Java Virtual Machine, which can optimize code as it loads classes for interpreting or just-in-time compiling.

Beyond the architectural features of dynamic linking, dynamic extension and class file compactness, there are some strategies that, although they are really not necessarily part of the architecture, help manage the time it takes to move class files across a network. Because HTTP protocols require that each class file of Java applet be requested individually, it turns out that often a large percentage of applet download time is due not to the actual transmission of class files across the network, but to the network handshaking of each class file request. The overhead for a file request is multiplied by the number of class files being requested. To address this problem, Java 1.1 included support for JAR (Java ARchive) files. JAR files enable many class files to be sent in one network transaction, which greatly reduces the overhead time required to move class files across a network compared with sending one class file at a time. Moreover, the data inside a JAR file can be compressed, which results in an even shorter download time. So sometimes it pays to send software across a network in one big chunk. If a set of class files is definitely needed by a program before that program can start, those class files can be more speedily transmitted if they are sent together in a JAR file.

One other strategy to minimize an end-userís wait time is to not download class files on-demand. Through various techniques, such as the subscription model used by Marimba Castanet, class files can be downloaded before they are needed, resulting in a program that starts up faster. You can obtain more information about Castanetís model from the resource page for this chapter.

Therefore, other than platform independence and security, which help make network-mobility practical, the main focus of Javaís architectural support for network-mobility is managing the time it takes to send class files across a network. Dynamic linking and dynamic extension allow Java programs to be designed in small functional units that are downloaded as needed by the end-user. Class file compactness helps reduce the time it takes to move a Java program across the network. The JAR file enables compression and the sending of multiple class files across the network in a single network file-transfer transaction.

The Applet: An Example of Network-Mobile Java

Java is a network-oriented technology that first appeared at a time when the network was looking increasingly like the next revolution in computing. The reason Java was adopted so rapidly and so widely, however, was not simply because it was a timely technology, but because it had timely marketing. Java was not the only network-oriented technology being developed in the early to mid 1990s. And although it was a good technology, it wasnít the necessarily the best technology--but it probably had the best marketing. Java was the one technology to hit a slim market window in early 1995, resulting in such a strong response that many companies developing similar technologies canceled their projects, including Microsoft, which canceled a project code-named Blackbird. Companies that carried on with their technologies, such as AT&T did with a network-oriented technology named

Inferno, saw Java steal much of their potential thunder.

There were several important factors in how Java was initially unleashed on the world that contributed to its successful marketing. First, it had a cool name--one that could be appreciated by programmers and non-programmers alike. Second, it was, for all practical purposes, free--always a strong selling point among prospective buyers. But the most critical factor contributing to the successful marketing of Java, however, was that Sunís engineers hooked Java technology to the World Wide Web at the precise moment Netscape was looking to transform their web browser from a graphical hypertext viewer to a full-fledged computing platform. As the World Wide Web swept through the software industry (and the global consciousness) like an ever-increasing tidal wave, Java rode with it. Therefore, in a sense Java became a success because Java "surfed the web." It caught the wave at just the right time and kept riding it as one by one, its potential competitors dropped uneventfully into the cold, dark sea. The way the engineers at Sun hooked Java technology to the World Wide Web--and therefore, the key way Java was successfully marketed--was by creating a special flavor of Java program that ran inside a web browser: the Java applet.

The Java applet showed off all of Javaís network-oriented features: platform independence, network mobility, and security. Platform independence was one of the main tenets of the World Wide Web, and Java applets fit right in. Java applets can run on any platform so long as there is a Java-capable browser for that platform. Java applets also demonstrated Javaís security capabilities, because they run inside a strict sandbox. But most significantly, Java applets demonstrated the promise of network-mobility. As shown in Figure 4-1, Java applets can be maintained on one server, from which they can travel across a network to many different kinds of computers. To update an applet, you only need to update the server. Users will automatically get the updated version the next time they use the applet. Thus, maintenance is localized, but processing is distributed.

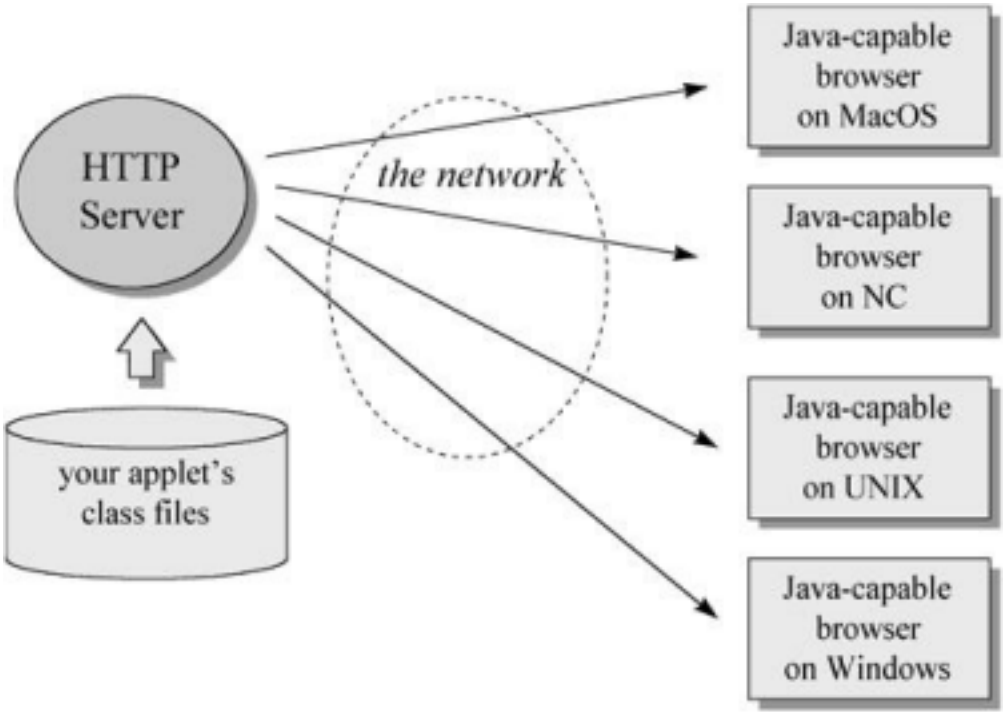


Figure 4-1. The Java Applet and the network.

Java-capable browsers fire off a Java application that hosts the applets the browser displays. To display a web page, a web browser requests an HTML file from an HTTP server. If the HTML file includes an applet, the browser will see an HTML tag such as this:

```
begin

<applet CODE="HeapOfFish.class"
```

```
CODEBASE="gcsupport/classes"

WIDTH=525

HEIGHT=360</applet>

end
```

This "applet" tag provides enough information to enable the browser to display the applet. The `CODE` attribute indicates the name of the appletís starting class file, in this case: `HeapOfFish.class`. The `CODEBASE` attribute gives the location of the appletís class files relative to the base URL of the web page. The `WIDTH` and `HEIGHT` attributes indicate the size in pixels of the appletís panel, the visible portion of the applet that is displayed as part of the web page.

When a browser encounters a web page that includes an applet tag, it passes information from the tag to the running Java application. The Java application either creates a new class loader object object, or re uses an existing one, to download the starting class file for the applet. It then initializes the applet, by invoking the `init()` method of the appletís starting class. The other class files for the applet are downloaded on an as needed basis, by the normal process of dynamic linking. For example, when a new class is first used by the appletís starting class, the symbolic reference to the new class must be resolved. During resolution, if the class has not already been loaded, the Java Virtual Machine will ask the same class loader object that loaded the appletís starting class to load the new class. If the class loader object is unable to load the class from the local trusted repository through the primordial class loader, the class loader object will attempt to download the class file across the network from the same location it retrieved the appletís starting class. Once initialization of the applet is complete, the applet appears as part of the web page inside the browser.

Network-mobile Java can take other forms besides Java applets, but the framework that supports other network-mobile forms will likely look very similar to the framework for applets. Like applets, other forms of network-mobile Java code will in general be implemented as special flavors of Java program running in the context of a host Java application. Network-mobile class files will generally be loaded by class loader objects for two reasons. The first reason is simply that a class loader object can download class files across a network in custom ways that a primordial class loader canít. But the second reason involves security. Because network-mobile class files are not always known to be trustworthy, the separate name-spaces provided by class loader objects are needed to protect a malicious applet from interfering with applets loaded from other sources. Also because network-mobile class files canít always be trusted, there will generally be a security manager establishing a sandbox for the network-mobile code.

The Resources Page

For links to information about other examples of network-mobile Java, such as Marimba Castanet Channels, Jeeves Servlets, and Aglets (Java-based autonomous software agents), see the resource page for this chapter: http://www.artima.com/insidejvm/mobility.html

McGraw-Hill
A Division of The McGraw-Hill Companies

Chapter Five

The Java Virtual Machine

The previous four chapters of this book gave a broad overview of Javaís architecture. They showed how the Java Virtual Machine fits into the overall architecture relative to other components such as the language and API. The remainder of this book will focus more narrowly on the Java Virtual Machine. This chapter gives an overview of the Java Virtual Machineís internal architecture.

The Java Virtual Machine is called "virtual" because it is an abstract computer defined by a specification. To run a Java program, you need a concrete implementation of the abstract specification. This chapter describes primarily the abstract specification of the Java Virtual Machine. To illustrate the abstract definition of certain features, however, this chapter also discusses various ways in which those features could be implemented.

What is a Java Virtual Machine?

To understand the Java Virtual Machine you must first be aware that you may be talking about any of three different things when you say "Java Virtual Machine." You may be speaking of:

- the abstract specification,
- a concrete implementation, or
- a runtime instance.

The abstract specification is a concept, described in detail in the book: *The Java Virtual Machine Specification*, by Tim Lindholm and Frank Yellin. Concrete implementations, which exist on many platforms and come from many vendors, are either all software or a combination of hardware and software. A runtime instance hosts a single running Java application.

Each Java application runs inside a runtime instance of some concrete implementation of the abstract specification of the Java Virtual Machine. In this book, the term "Java Virtual Machine" is used in all three of these senses. Where the intended sense is not clear from the context, one of the terms "specification," "implementation," or "instance" is added to the term "Java Virtual Machine".

The Lifetime of a Java Virtual Machine

A runtime instance of the Java Virtual Machine has a clear mission in life: to run one Java application.

When a Java application starts, a runtime instance is born. When the application completes, the instance dies. If you start three Java applications at the same time, on the same computer, using the same concrete implementation, youíll get three Java Virtual Machine instances. Each Java application runs inside its own Java Virtual Machine.

A Java Virtual Machine instance starts running its solitary application by invoking the `main()` method of some initial class. The `main()` method must be public, static, return `void`, and accept one parameter: a `String` array. Any class with such a `main()` method can be used as the starting point for a Java application.

For example, consider an application that prints out its command line

arguments: `begin`

```
// On CD-ROM in file jvm/ex1/Echo.java
class Echo {



 public static void main(String[] args) {

 int len = args.length;

 for (int i = 0; i < len; ++i) {

 System.out.print(args[i] + " ");

 }

 System.out.println();

 }

}
```

`end`

You must in some implementation-dependent way give a Java Virtual Machine the name of the initial class that has the `main()` method that will start the entire application. One real world example of a Java Virtual Machine implementation is the `java` program from Sunís JDK. If you wanted to run the `Echo` application using Sunís `java` on Window95, for example, you would type in a command such as:

`insert`

`java Echo Greetings, Planet.`

`end`

The first word in the command, "`java`," indicates that the Java Virtual Machine from Sunís JDK should be run by the operating system. The second word, "`Echo`," is the name of the initial class. `Echo` must have a public static method named `main()` that returns `void` and takes a `String` array as its only parameter. The subsequent words, "`Greetings, Planet.`," are the command line arguments for the application. These are passed to the `main()` method in the `String` array in the order in which they appear on the command line. So, for the above example, the contents of the `String` array passed to main in `Echo` are:

`arg[0]` is "`Greetings,`"

`arg[1]` is `"Planet."`

The `main()` method of an applicationís initial class serves as the starting point for that applicationís initial thread. The initial thread can in turn fire off other threads.

Inside the Java Virtual Machine, threads come in two flavors: *daemon* and *non-daemon*. A daemon thread is ordinarily a thread used by the virtual machine itself, such as a thread that performs garbage collection. The application, however, can mark any threads it creates as daemon threads. The initial thread of an application--the one that begins at `main()`--is a non-daemon thread.

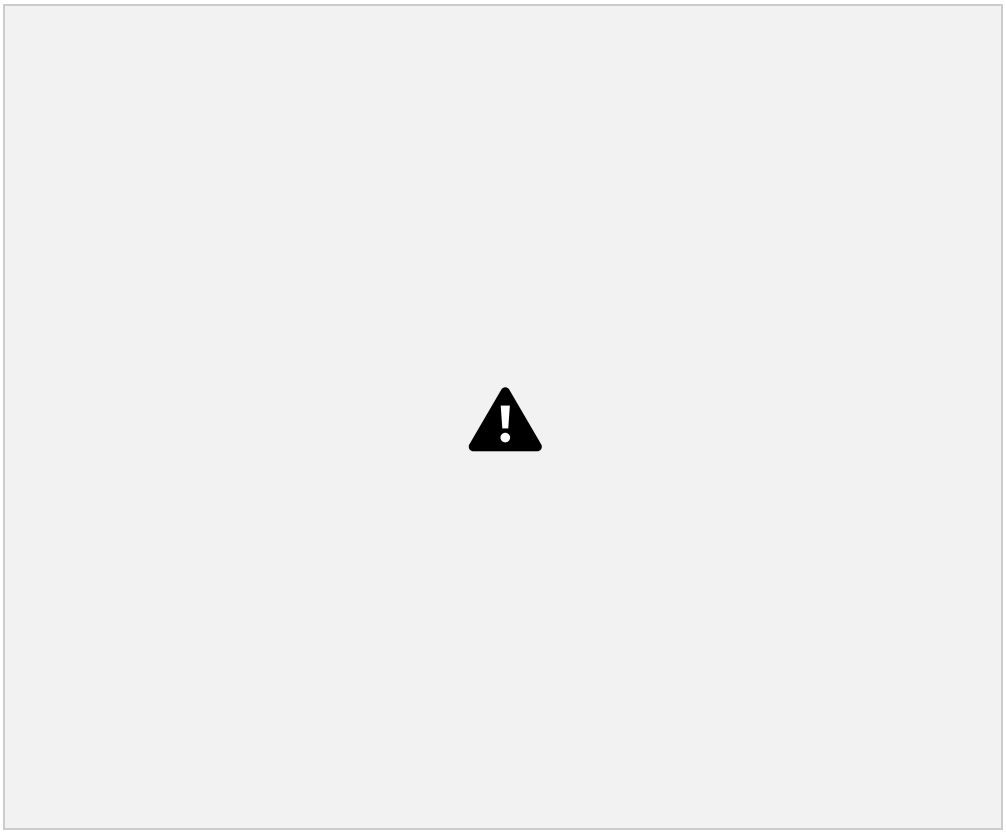
A Java application continues to execute (the virtual machine instance continues to live) as long as any non-daemon threads are still running. When all non-daemon threads of a Java application terminate, the virtual machine instance will exit. If permitted by the security manager, the application can also cause its own demise by invoking the `exit()` method of class `Runtime` or `System`.

In the `Echo` application above, the `main()` method doesnít invoke any other threads. After it prints out the command line arguments, `main()` returns. This terminates the applicationís only non-daemon thread, which causes the virtual machine instance to exit.

The Architecture of the Java Virtual Machine

In the Java Virtual Machine specification, the behavior of a virtual machine instance is described in terms of subsystems, memory areas, data types, and instructions. These components describe an abstract inner architecture for the abstract Java Virtual Machine. The purpose of these components is not so much to dictate an inner architecture for implementations. It is more to provide a way to strictly define the external behavior of implementations. The specification defines the required behavior of any Java Virtual Machine implementation in terms of these abstract components and their interactions.

Figure 5-1 shows a block diagram of the Java Virtual Machine that includes the major subsystems and memory areas described in the specification. As mentioned in previous chapters, each Java Virtual Machine has a *class loader subsystem*: a mechanism for loading types (classes and interfaces) given fully qualified names. Each Java Virtual Machine also has an *execution engine*: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

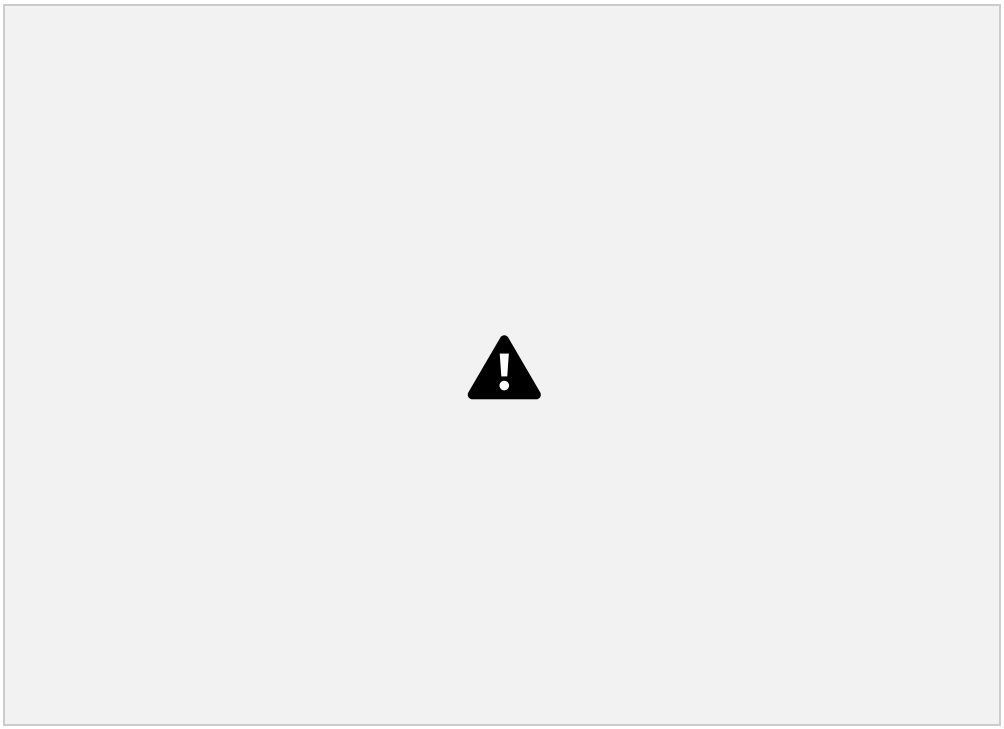


When a Java Virtual Machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The Java Virtual Machine organizes the memory it needs to execute a program into several *runtime data areas*.

Although the same runtime data areas exist in some form in every Java Virtual Machine implementation, their specification is quite abstract. Many decisions about the structural details of the runtime data areas are left to the designers of individual implementations.

Different implementations of the virtual machine can have very different memory constraints. Some implementations may have a lot of memory in which to work, others may have very little. Some implementations may be able to take advantage of virtual memory, others may not. The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java Virtual Machine on a wide variety of computers and devices.

Some runtime data areas are shared among all of an applicationís threads and others are unique to individual threads. Each instance of the Java Virtual Machine has one *method area* and one *heap*. These areas are shared by all threads running inside the virtual machine. When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap. See Figure 5-2 for a graphical depiction of these memory areas.



As each new thread comes into existence, it gets its own *pc register* (program counter) and *Java stack*. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A threadís Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in *native method stacks*, as well as possibly in registers or other implementation-dependent memory areas.

The Java stack is composed of *stack frames* (or *frames*). A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java Virtual Machine pushes a new frame onto that threadís Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

The Java Virtual Machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values. This approach was taken by Javaís designers to keep the Java Virtual Machine's instruction set compact and to facilitate implementation on architectures with few or irregular general purpose registers.

See Figure 5-3 for a graphical depiction of the memory areas the Java Virtual Machine creates for each thread. These areas are private to the owning thread. No thread can access the pc register or Java stack of another thread.

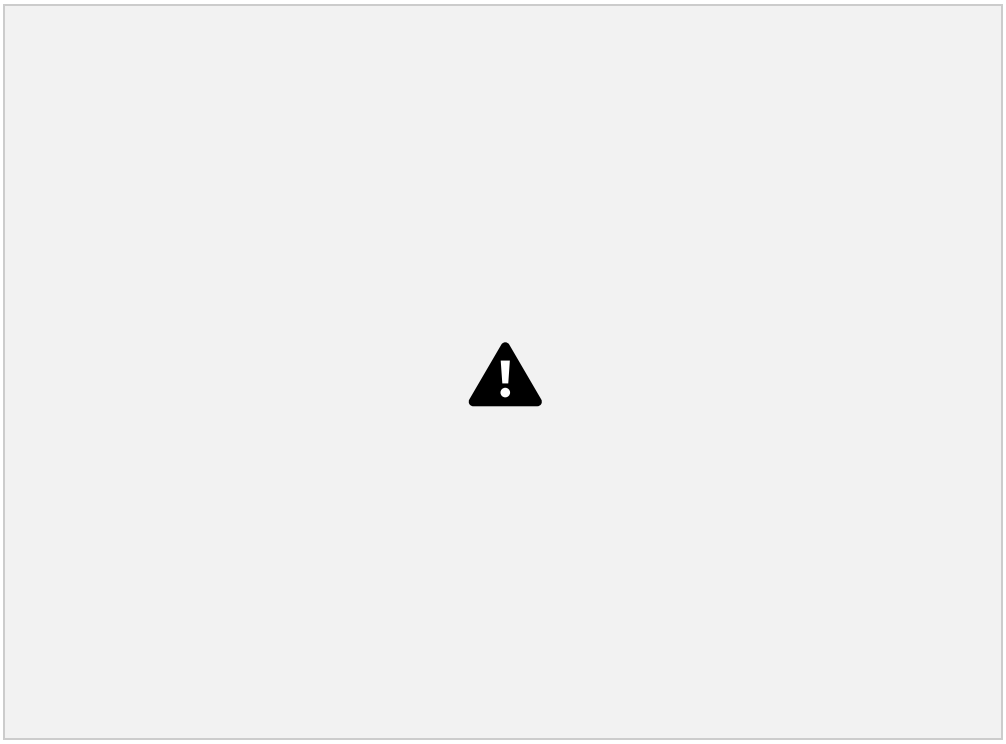
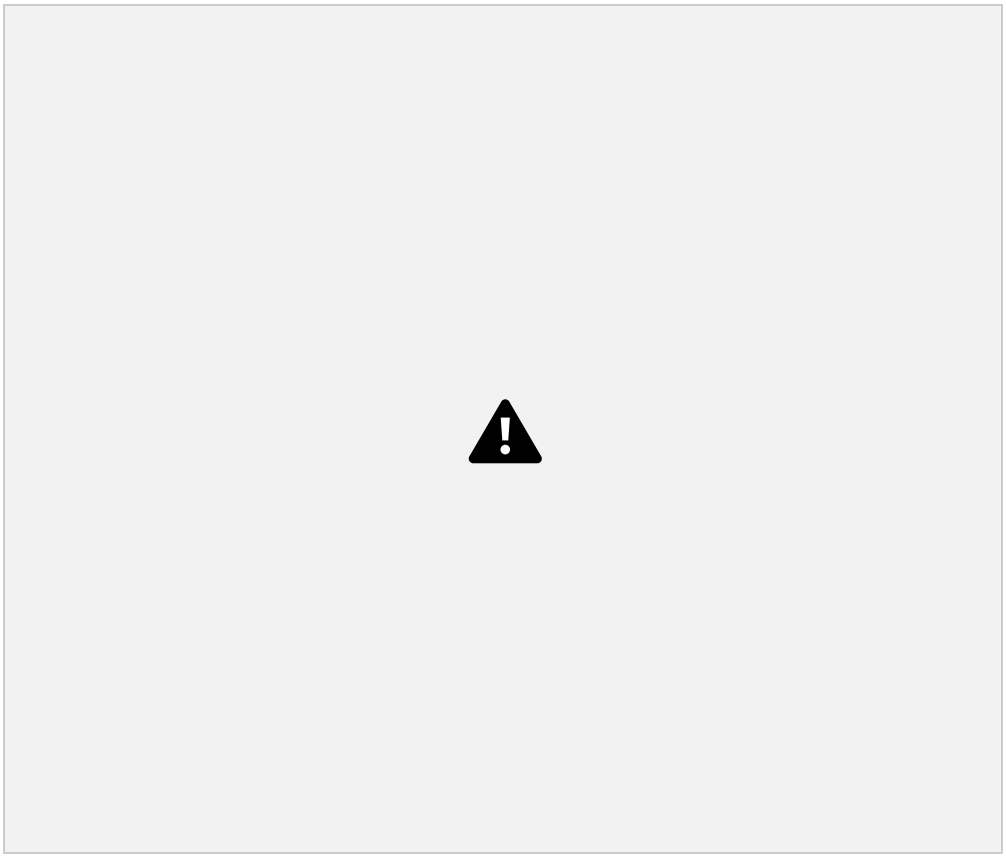


Figure 5-3 shows a snapshot of a virtual machine instance in which three threads are executing. At the instant of the snapshot, threads one and two are executing Java methods. Thread three is executing a native method.

In Figure 5-3, as in all graphical depictions of the Java stack in this book, the stacks are shown growing downwards. The "top" of each stack is shown at the bottom of the figure. Stack frames for currently executing methods are shown in a lighter shade. For threads that are currently executing a Java method, the pc register indicates the next instruction to execute. In Figure 5-3, such pc registers (the ones for threads one and two) are shown in a lighter shade. Because thread three is currently executing a native method, the contents of its pc register--the one shown in dark gray--is undefined.

Data Types

The Java Virtual Machine computes by performing operations on certain types of data. Both the data types and operations are strictly defined by the Java Virtual Machine specification. The data types can be divided into a set of *primitive types* and a *reference type*. Variables of the primitive types hold *primitive values*, and variables of the reference type hold *reference values*. Reference values refer to objects, but are not objects themselves. Primitive values, by contrast, do not refer to anything. They are the actual data themselves. You can see a graphical depiction of the Java Virtual Machineís families of data types in Figure 5-4.



All the primitive types of the Java programming language, except `boolean`, are primitive types of the Java Virtual Machine. When a compiler translates Java source code into bytecodes, it uses `int`s or `byte`s to represent `boolean`s. In the Java Virtual Machine, `false` is represented by integer zero and `true` by any non-zero integer. Operations involving `boolean` values use `int`s. Arrays of `boolean` are accessed as arrays of `byte`, though they may be represented on the heap as arrays of `byte` or as bit fields.

The primitive types of the Java programming language other than `boolean` form the *numeric types* of the Java Virtual Machine. The numeric types are divided between the *integral types*: `byte`, `short`, `int`, `long`, and `char`, and the *floating-point types*: `float` and `double`. As with the Java programming language, the primitive types of the Java Virtual Machine have the same range everywhere. A `long` in the Java Virtual Machine always acts like a 64-bit signed twos complement number, independent of the underlying host platform.

The Java Virtual Machine works with one other primitive type that is unavailable to the Java programmer: the `returnValue` type. This primitive type is used to implement `finally` clauses of Java programs. The use of the `returnValue` type is described in detail in Chapter 18, "Finally Clauses."

The reference type of the Java Virtual Machine is cleverly named `reference`. Values of type `reference` come in three flavors: the *class type*, the *interface type*, and the *array type*. All three types have values that are references to dynamically created objects. The class typeís values are references to class instances. The array typeís values are references to arrays, which are full-fledged objects in the Java Virtual Machine. The interface typeís values are references to class instances that implement an interface. One other reference value is the `null` value, which indicates the `reference` variable
doesnít refer to any object.

The Java Virtual Machine specification defines the range of values for each of the data types, but does not define their sizes. The number of bits used to store each data type value is a decision of the designers of individual implementations. The ranges of the Java Virtual Machines data typeís are shown in Table

5-1. More information on the floating point ranges is given in Chapter 14, "Floating Point Arithmetic."

Table 5-1. Ranges of the Java Virtual Machineís data types

Type	Range
`byte`	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
`short`	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
`int`	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
`long`	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
`char`	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
`float`	32-bit IEEE 754 single-precision float
`double`	64-bit IEEE 754 double-precision float
`returnValue`	address of an opcode within the same method
`reference`	reference to an object on the heap, or `null`

Word Size

The basic unit of size for data values in the Java Virtual Machine is the *word*--a fixed size chosen by the designer of each Java Virtual Machine implementation. The word size must be large enough to hold a value of type `byte`, `short`, `int`, `char`, `float`, `returnValue`, or `reference`. Two words must be large enough to hold a value of type `long` or `double`. An implementation designer must therefore choose a word size that is at least 32 bits, but otherwise can pick whatever word size will yield the most efficient implementation. The word size is often chosen to be the size of a native pointer on the host platform.

The specification of many of the Java Virtual Machineís runtime data areas are based upon this abstract concept of a word. For example, two sections of a Java stack frame--the local variables and operand stack--are defined in terms of words. These areas can contain values of any of the virtual machineís data types. When placed into the local variables or operand stack, a value occupies either one or two words.

As they run, Java programs cannot determine the word size of their host virtual machine implementation. The word size does not affect the behavior of a program. It is only an internal attribute of a virtual machine implementation.

The Class Loader Subsystem

The part of a Java Virtual Machine implementation that takes care of finding and loading types is the *class loader subsystem*. Chapter 1, "Introduction to Javaís Architecture," gives an overview of this subsystem. Chapter 3, "Security," shows how the subsystem fits into Javaís security model. This chapter describes the class loader subsystem in more detail and show how it relates to the other components of the virtual machineís internal architecture.

As mentioned in Chapter 1, the Java Virtual Machine contains two kinds of class loaders: a *primordial class loader* and *class loader objects*. The primordial class loader is a part of the virtual machine implementation, and class loader objects are part of the running Java application. Classes loaded by

different class loaders are placed into separate *name spaces* inside the Java Virtual Machine.

The class loader subsystem involves many other parts of the Java Virtual Machine and several classes from the `java.lang` library. For example, class loader objects are regular Java objects whose class descends from `java.lang.ClassLoader`. The methods of class `ClassLoader` allow Java applications to access the virtual machineís class loading machinery. Also, for every type a Java Virtual Machine loads, it creates an instance of class `java.lang.Class` to represent that type. Like all objects, class loader objects and instances of class `Class` reside on the heap. Data for loaded types resides in the method area.

Loading, Linking and Initialization

The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. These activities are performed in a strict order:

1. Loading: finding and importing the binary data for a type
2. Linking: performing verification, preparation, and (optionally) resolution
    a. Verification: ensuring the correctness of the imported type
    b. Preparation: allocating memory for class variables and initializing the memory to default values
    c. Resolution: transforming symbolic references from the type into direct references.
- Initialization: invoking Java code that initializes class variables to their proper starting values.

The details of these processes are given Chapter 7, "The Lifetime of a Class."

The Primordial Class Loader

Java Virtual Machine implementations must be able to recognize and load classes and interfaces stored in binary files that conform to the Java class file format. An implementation is free to recognize other binary forms besides class files, but it must recognize class files. One example of an alternative binary format recognized by a particular Java Virtual Machine implementation is the CAB file. This file format, which is an archive of class files and other data files, is defined by Microsoft and recognized by their implementation of the Java Virtual Machine.

Every Java Virtual Machine implementation has a primordial class loader, which knows how to load trusted classes, including the classes of the Java API. The Java Virtual Machine specification doesnít define how the primordial loader should locate classes. That is another decision the specification leaves to implementation designers.

Given a fully qualified type name, the primordial class loader must *in some way* attempt to locate a file with the typeís simple name plus "`.class`". One common approach is demonstrated by the Java Virtual Machine implementation in Sunís 1.1 JDK on Windows95. This implementation searches a user defined directory path stored in an environment variable named `CLASSPATH`. The primordial loader looks in each directory, in the order the directories appear in the `CLASSPATH`, until it finds a file with the appropriate name: the typeís simple name plus "`.class`". Unless the type is part of the unnamed package, the primordial loader expects the file to be in a subdirectory of one the directories in the `CLASSPATH`. The path name of the subdirectory is built from the package name of the type. For

example, if the primordial class loader is searching for class `java.lang.Object`, it will look for `Object.class` in the `java\lang` subdirectory of each `CLASSPATH` directory.

Class Loader Objects

Although class loader objects themselves are part of the Java application, three of the methods in class `ClassLoader` are gateways into the Java Virtual Machine:

```
begin

// Three of the methods declared in class java.lang.ClassLoader:

protected final Class defineClass(byte data[], int offset,

 int length);

protected final Class findSystemClass(String name);

protected final void resolveClass(Class c);

end
```

Any Java Virtual Machine implementation must take care to connect these methods of class `ClassLoader` to the internal class loader subsystem.

The `defineClass()` method accepts a `byte` array, `data[]`, as input. Starting at position `offset` in the array and continuing for `length` bytes, class `ClassLoader` expects binary data conforming to the Java class file format--binary data that represents a new type for the running application. Every Java
Virtual Machine implementation must make sure the `defineClass()` method of class `ClassLoader` can cause a new type to be imported into the method area.

The `findSystemClass()` method accepts a `String` representing a fully qualified name of a type. When a class loader object invokes this method, it is requesting that the virtual machine attempt to load the named type via its primordial class loader. If the primordial class loader has already loaded or successfully loads the type, it returns a reference to the `Class` object representing the type. If it canít locate the binary data for the type, it throws `ClassNotFoundException`. Every Java Virtual Machine implementation must make sure the `findSystemClass()` method can invoke the primordial class loader in this way.
The `resolveClass()` method accepts a reference to a `Class` instance. This method causes the type represented by the `Class` instance to be linked and initialized (if it hasnít already been linked and initialized). The `defineClass()` method, described above, only takes care of loading. (See the above section, "Loading, Linking, and Initialization" for definitions of these terms.) When `defineClass()` returns a `Class` instance, the binary file for the type has definitely been located and imported into the method area, but not necessarily linked and initialized. Java Virtual Machine implementations make sure the `resolveClass()` method of class `ClassLoader` can cause the class loader subsystem to perform linking and initialization.

The details of how a Java Virtual Machine performs class loading, linking, and initialization, with class loader objects is given in Chapter 8, "The Linking Model."

Name Spaces

As mentioned in Chapter 3, "Security," each class loader maintains its own name space populated by the types it has loaded. Because each class loader has its own name space, a single Java application can load multiple types with the same fully qualified name. A typeís fully qualified name, therefore, is not always

enough to uniquely identify it inside a Java Virtual Machine instance. If multiple types of that same name have been loaded into different name spaces, the identity of the class loader that loaded the type (the identity of the name space it is in) will also be needed to uniquely identify that type.

Name spaces arise inside a Java Virtual Machine instance as a result of the process of resolution. As part of the data for each loaded type, the Java Virtual Machine keeps track of the class loader that imported the type. When the virtual machine needs to resolve a symbolic reference from one class to another, it requests the referenced class from the same class loader that loaded the referencing class. This process is described in detail in Chapter 8, "The Linking Model."

The Method Area

Inside a Java Virtual Machine instance, information about loaded types is stored in a logical area of memory called the method area. When the Java Virtual Machine loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area. Memory for class (static) variables declared in the class is also taken from the method area.

The manner in which a Java Virtual Machine implementation represents type information internally is a decision of the implementation designer. For example, multi-byte quantities in class files are stored in big-endian (most significant byte first) order. When the data is imported into the method area, however, a virtual machine can store the data in any manner. If an implementation sits on top of a little-endian processor, the designers may decide to store multi-byte values in the method area in little-endian order.

The virtual machine will search through and use the type information stored in the method area as it executes the application it is hosting. Designers must attempt to devise data structures that will facilitate speedy execution of the Java application, but must also think of compactness. If designing an implementation that will operate under low memory constraints, designers may decide to trade off some execution speed in favor of compactness. If designing an implementation that will run on a virtual memory system, on the other hand, designers may decide to store redundant information in the method area to facilitate execution speed. (If the underlying host doesnít offer virtual memory, but does offer a hard disk, designers could create their own virtual memory system as part of their implementation.) Designers can choose whatever data structures and organization they feel optimize their implementations performance, in the context of its requirements.

All threads share the same method area, so access to the method areaís data structures must be designed to be thread-safe. If two threads are attempting to find a class named `Lava`, for example, and `Lava` has not yet been loaded, only one thread should be allowed to load it while the other one waits.

The size of the method area need not be fixed. As the Java application runs, the virtual machine can expand and contract the method area to fit the applicationís needs. Also, the memory of the method area need not be contiguous. It could be allocated on a heap--even on the virtual machineís own heap. Implementations may allow users or programmers to specify an initial size for the method area, as well as a maximum or minimum size.

The method area can also be garbage collected. Because Java programs can be dynamically extended via class loader objects, classes can become "unreferenced" by the application. If a class becomes unreferenced, a Java Virtual Machine can unload the class (garbage collect it) to keep the memory occupied by the method area at a minimum. The unloading of classes--including the conditions under which a class can become "unreferenced"--is described in Chapter 7, "The Lifetime of a Class."

Type Information

For each type it loads, a Java Virtual Machine must store the following kinds of information in the method area:

- The fully qualified name of the type
- The fully qualified name of the typeís direct superclass (unless the type is an interface or class `java.lang.Object`, neither of which have a superclass)
- Whether or not the type is a class or an interface
- The typeís modifiers (some subset of` public`, `abstract`, `final`)
- An ordered list of the fully qualified names of any direct superinterfaces

Inside the Java class file and Java Virtual Machine, type names are always stored as *fully qualified names*. In Java source code, a fully qualified name is the name of a typeís package, plus a dot, plus the typeís *simple name*. For example, the fully qualified name of class `Object` in package `java.lang` is `java.lang.Object`. In class files, the dots are replaced by slashes, as in `java/lang/Object`. In the method area, fully qualified names can be represented in whatever form and data structures a designer chooses.

In addition to the basic type information listed above, the virtual machine must also store for each loaded type:

- The constant pool for the type
- Field information
- Method information
- All class (static) variables declared in the type, except constants
- A reference to class `ClassLoader`
- A reference to class `Class`

This data is described in the following sections.