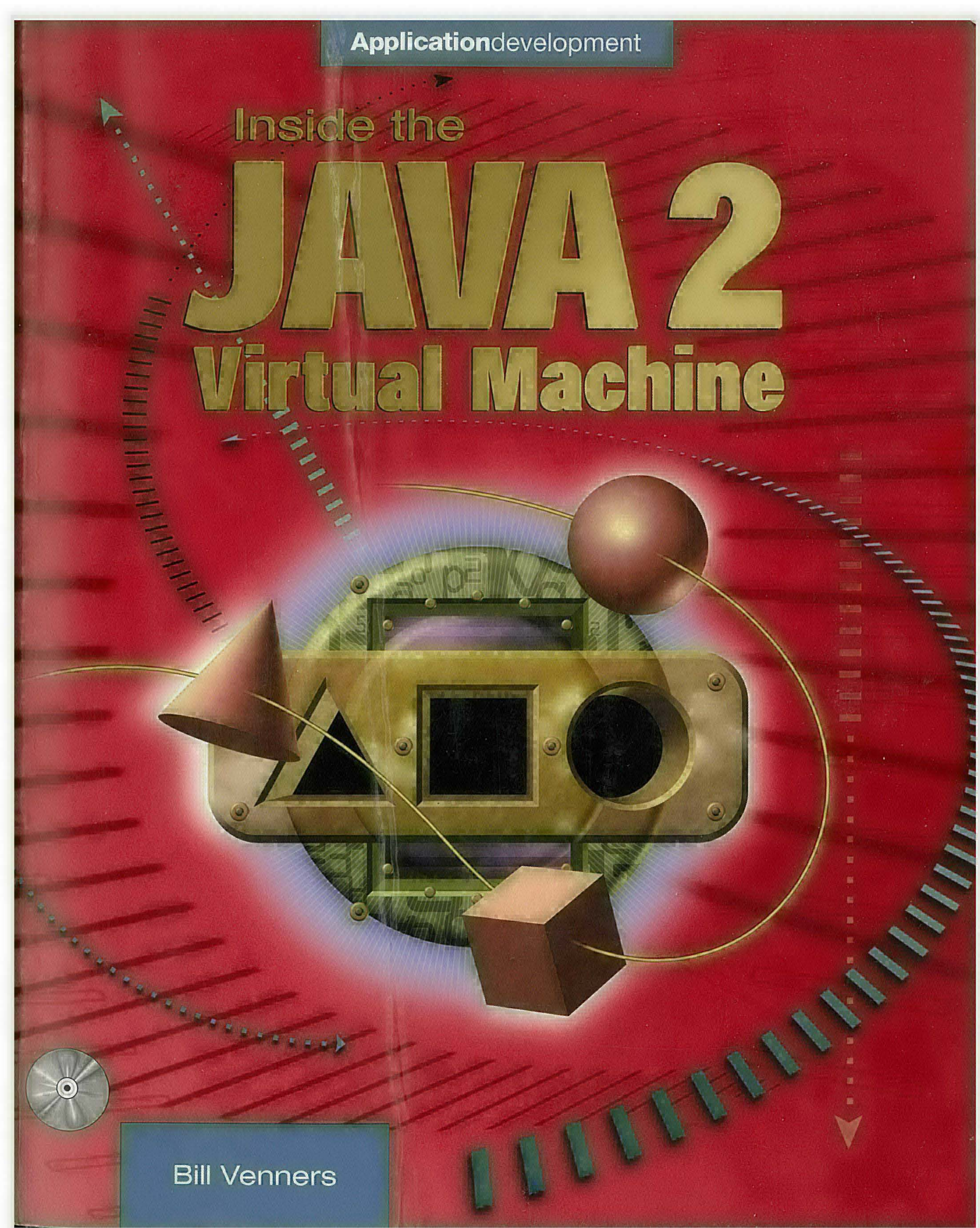
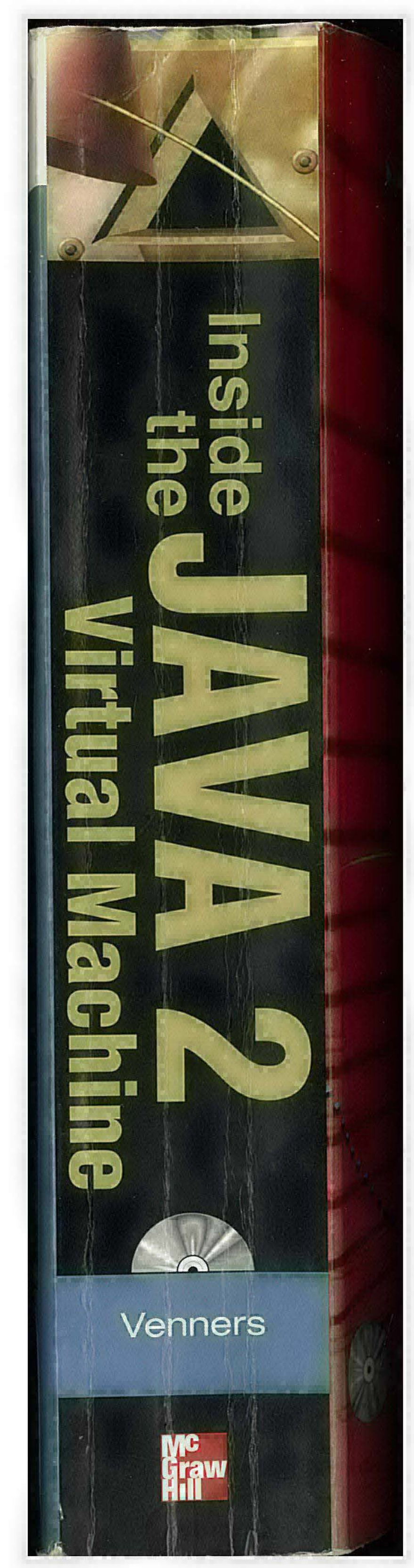


GOOGLE EXHIBIT 1032


Inside the
JAVA 2
Virtual Machine
Venners
Mc
Graw
Hill

INSIDE THE JAVA VIRTUAL MACHINE

SECOND EDITION

Inside the Java Virtual Machine

Bill Venners

Second Edition

McGraw-Hill
New York San Francisco Washington, D.C.
Auckland Bogotá Caracas Lisbon London Madrid
Mexico City Milan Montreal New Delhi San Juan
Singapore Sydney Tokyo Toronto

McGraw-Hill

Copyright © 1999 by The McGraw-Hill Companies, Inc. All Rights Reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Portions of this book were derived from articles written by Bill Venners and first published in the column "Under the Hood" of *JavaWorld*, a division of Web Publishing, Inc., June, 1996 through October, 1997.

1 2 3 4 5 6 7 8 9 0 AGM/AGM 9 0 4 3 2 1 0 9

P/N 135094-2
Part of ISBN 0-07-135093-4

The sponsoring editor for this book was Simon Yates and the production supervisor was Clare Stanley. It was set in Century Schoolbook by Douglas & Gayle, Limited.

Printed and bound by Quebecor/Martinsburg.

Throughout this book, trademarked names are used. Rather than put a trademark symbol after every occurrence of a trademarked name, we used the names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

Information contained in this work has been obtained by The McGraw-Hill Companies, Inc. ("McGraw-Hill") from sources believed to be reliable. However, neither McGraw-Hill nor its authors guarantees the accuracy or completeness of any information published herein and neither McGraw-Hill nor its authors shall be responsible for any errors, omissions, or damages arising out of use of this information. This work is published with the understanding that McGraw-Hill and its authors are supplying information but are not attempting to render engineering or other professional services. If such services are required, the assistance of an appropriate professional should be sought.

This book is printed on recycled, acid-free paper containing a minimum of 50 percent recycled de-inked fiber.

To my parents

CONTENTS

Preface xv
Introduction xvii
Acknowledgments xxxii

Chapter 1 Introduction to Java's Architecture 1

Why Java? 2
The Challenges and Opportunities of Networks 2
The Architecture 4
  The Java Virtual Machine 5
  The Class Loader Architecture 8
  The Java Class File 11
  The Java API 12
  The Java Programming Language 14
Architectural Tradeoffs 16
Conclusion 21
The Resources Page 21

Chapter 2 Platform Independence 23

Why Platform Independence? 24
Java's Architectural Support for Platform Independence 25
  The Java Platform 25
  The Java Language 26
  The Java Class File 26
  Scaleability 26
Factors that Influence Platform Independence 29
  Java Platform Deployment 29
  The Java Platform Version and Edition 30
  Native Methods 31
  Non-Standard Run-Time Libraries 32
  Virtual Machine Dependencies 33
  User Interface Dependencies 34
  Bugs in Java Platform Implementations 34
  Testing 34
Seven Steps to Platform Independence 35
The Politics of Platform Independence 36
Platform Independence and Network-Mobile Objects 39
The Resources Page 40

Chapter 3 Security 41

Why Security? 42
The Basic Sandbox 43
The Class Loader Architecture 45
The Class File Verifier 52
Pass One: Structural Checks on the Class File 53
Pass Two: Semantic Checks on the Type Data 54
Pass Three: Bytecode Verification 54
Pass Four: Verification of Symbolic References 56
Binary Compatibility 58
Safety Features Built Into the Java Virtual Machine 59
The Security Manager and the Java API 62
Code Signing and Authentication 68
A Code-Signing Example 75
Policy 79
Policy File 82
Protection Domains 84
The Access Controller 86
The implies( ) Method 87
Stack Inspection Examples 90
A Stack Inspection That Says "Yes" 93
A Stack Inspection That Says "No" 97
The doPrivileged( ) Method 100
A Futile Use of doPrivileged( ) 105
Missing Pieces and Future Directions 109
Security Beyond the Architecture 110
The Resources Page 111

Chapter 4 Network Mobility 113

Why Network Mobility? 114
A New Software Paradigm 116
Java's Architectural Support for Network Mobility 120
The Applet: An Example of Network-Mobile Code 122
The Jini Service Object: An Example of Network-Mobile Objects 125
What is Jini? 126
How Jini Works 126
The Benefits of the Service Object 129
Network Mobility: The Design Center of Java 131
The Resources Page 132

Chapter 5 The Java Virtual Machine 133

What Is a Java Virtual Machine? 134
The Lifetime of a Java Virtual Machine 134
The Architecture of the Java Virtual Machine 136
Data Types 139
Word Size 142
The Class Loader Subsystem 142
The Method Area 146
The Heap 154
The Program Counter 161
The Java Stack 162
The Stack Frame 163
Native Method Stacks 172
Execution Engine 173
Native Method Interface 186
The Real Machine 187
Eternal Math: A Simulation 188
On the CD-ROM 189
The Resources Page 190

Chapter 6 The Java Class File 191

What Is a Java Class File? 192
What Is in a Class File? 193
Special Strings 201
Fully Qualified Names 202
Simple Names 202
Descriptors 202
The Constant Pool 205
The CONSTANT_Utf8_info Table 205
The CONSTANT_Integer_info Table 207
The CONSTANT_Float_info Table 208
The CONSTANT_Long_info Table 208
The CONSTANT_Double_info Table 209
The CONSTANT_Class_info Table 209
The CONSTANT_String_info Table 210
The CONSTANT_Fieldref_info Table 211
The CONSTANT_Methodref_info Table 212
The CONSTANT_InterfaceMethodref_info Table 212
The CONSTANT_NameAndType_info Table 213

Fields 214
Methods 216
Attributes 218
Attribute Format 219
The Code Attribute 220
The ConstantValue Attribute 223
The Deprecated Attribute 224
The Exceptions Attribute 225
The InnerClasses Attribute 225
The LineNumberTable Attribute 229
The LocalVariableTable Attribute 231
The SourceFile Attribute 233
The Synthetic Attribute 234
Getting Loaded: A Simulation 234
On the CD-ROM 236
The Resources Page 236

Chapter 7 The Lifetime of a Type 237

Type Loading, Linking, and Initialization 238
Loading 240
Verification 241
Preparation 244
Resolution 245
Initialization 245
The Lifetime of an Object 253
Class Instantiation 254
Garbage Collection and Finalization of Objects 264
Unloading of Types 265
On the CD-ROM 268
The Resources Page 268

Chapter 8 The Linking Model 269

Dynamic Linking and Resolution 270
Resolution and Dynamic Extension 272
Class Loaders and the Parent-Delegation Model 276
Constant Pool Resolution 277
Resolution of CONSTANT_Class_info Entries 278
Resolution of CONSTANT_Fieldref_info Entries 287
Resolution of CONSTANT_Methodref_info Entries 288
Resolution of CONSTANT_InterfaceMethodref_info Entries 289
Resolution of CONSTANT_String_info Entries 290

- Resolution of Other Types of Entries 292
- Loading Constraints 292
- Compile-Time Resolution of Constants 294
- Direct References 296
- _quick Instructions 305
- Example: The Linking of the Salutation Application 306
- Example: The Dynamic Extension of the Greet Application 318
- Using a 1.1 User-Defined Class Loader 323
- Using a Version 1.2 User-Defined Class Loader 329
- Example: Dynamic Extension with forName( ) 333
- Example: Unloading Unreachable Greeters 336
- Example: Type Safety and Loading Constraints 343

On the CD-ROM 353
The Resources Page 353

Chapter 9 Garbage Collection 355

Why Garbage Collection? 356
Garbage Collection Algorithms 357
Reference Counting Collectors 358
Tracing Collectors 359
Compacting Collectors 359
Copying Collectors 360
Generational Collectors 362
Adaptive Collectors 362
The Train Algorithm 363
- Cars, Trains, and a Railway Station 364
- Collecting Cars 366
- Remembered Sets and Popular Objects 367

Finalization 368
The Reachability Life Cycle of Objects 370
- Reference Objects 371
- Reachability State Changes 373
- Caches, Canonicalizing Mappings, and Pre-Mortem Cleanup 376

Heap of Fish: A Simulation 378
- Allocate Fish 379
- Assign References 381
- Garbage Collect 382
- Compact Heap 383

On the CD-ROM 384
The Resources Page 384

Chapter 10 Stack and Local Variable Operations 385

Pushing Constants onto the Stack 386
Generic Stack Operations 389
Pushing Local Variables onto the Stack 389
Popping to Local Variables 390
The wide Instruction 392
Fibonacci Forever: A Simulation 394
On the CD-ROM 397
The Resources Page 397

Chapter 11 Type Conversion 399

The Conversion Opcodes 400
Conversion Diversion: A Simulation 402
On the CD-ROM 405
The Resources Page 405

Chapter 12 Integer Arithmetic 407

Two's-Complement Arithmetic 408
Inner Int: A Java int Reveals Its Inner Nature 409
Arithmetic Opcodes 409
Prime Time: A Simulation 412
On the CD-ROM 416
The Resources Page 416

Chapter 13 Logic 417

The Logic Opcodes 418
Logical Results: A Simulation 419
On the CD-ROM 421
The Resources Page 422

Chapter 14 Floating-Point Arithmetic 423

Floating-Point Numbers 424
Inner Float: A Java float Reveals its Inner Nature 427
Floating Point Modes 428
- Floating-Point Value Sets 429
- Floating-Point Value Set Conversion 430
- Implications of the Relaxed Rules 431

The Floating Point Opcodes 431
Circle of Squares: A Simulation 434

On the CD-ROM 436
The Resources Page 436

Chapter 15 Objects and Arrays 437

A Refresher on Objects and Arrays 438
Opcodes for Objects 438
Opcodes for Arrays 440
Three-Dimensional Array: A Simulation 443
On the CD-ROM 447
The Resources Page 447

Chapter 16 Control Flow 449

Conditional Branching 450
Unconditional Branching 453
Conditional Branching with Tables 453
Saying Tomato: A Simulation 455
On the CD-ROM 457
The Resources Page 458

Chapter 17 Exceptions 459

Throwing and Catching Exceptions 460
The Exception Table 464
Play Ball!: A Simulation 465
On the CD-ROM 468
The Resources Page 468

Chapter 18 Finally Clauses 469

Miniature Subroutines 470
Asymmetrical Invocation and Return 471
Hop Around: A Simulation 474
On the CD-ROM 477
The Resources Page 478

Chapter 19 Method Invocation and Return 479

Method Invocation 480
  Invoking a Java Method 481
  Invoking a Native Method 482
Other Forms of Method Invocation 482
The invokespecial Instruction 483
  invokespecial and <init>() 483

invokespecial and Private Methods 486
invokespecial and super 487
The invokeinterface Instruction 490
Invocation Instructions and Speed 490
Examples of Method Invocation 491
Returning from Methods 495
On the CD-ROM 496
The Resources Page 496

Chapter 20 Thread Synchronization 497

Monitors 498
Object Locking 503
Synchronization Support in the Instruction Set 505
Synchronized Statements 505
Synchronized Methods 508
Coordination Support in Class Object 511
On the CD-ROM 511
The Resources Page 512

Appendix A 513
Appendix B 649
Appendix C 659
Appendix D 667

Index 677

PREFACE

My primary goal in writing this book was to explain the Java virtual machine—and several core Java APIs closely related to the virtual machine—to Java programmers. Although the Java virtual machine incorporates technologies that have 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 encounter 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—might be new to many programmers. The aim of this book is to help programmers understand how all these things work, and in the process we hope to help them become more adept at Java programming.

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: they are dynamic (can evolve over time), they are interactive (especially if you embed Java applets in them), and they are interconnected (you can easily navigate from one to another). Besides the traditional text and figures, this book includes several Java applets (in a mini-Web site on the CD-ROM) that serve as interactive illustrations of the concepts presented in the text. In addition, I maintain a Web site at `artima.com` on the Internet that serves as a launching point for readers to find more (and more current) information about 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

INTRODUCTION

This book describes the Java virtual machine, the abstract computer on which all Java programs run, and several core Java APIs that have an intimate relationship with the virtual machine. 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 why Java is a likely candidate for this role is that Java's 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. Java's inherent memory safety—garbage collection, lack of pointer arithmetic, and run-time checks on the use of references—prevents most memory bugs from ever occurring in Java programs. Java's 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 more important than they were in the past.

The Java virtual machine is responsible for the memory safety, platform independence, and security features of the Java programming language. Although virtual machines have been around for a long time, prior to Java, they had not quite entered the mainstream. Given today's emerging

hardware realities, however, 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 features for the coming years of computing. This book will help you get to know this virtual machine and some closely related Java APIs. Armed with this knowledge, you will be better able to take maximum advantage of Java's unique architecture in your own endeavors.

Who Should Read the Book?

This book is aimed primarily at professional software developers and students who want to understand Java technology. I assume that you are familiar, although 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 Java's 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 and Appendixes A through C)
4. Interactive illustrations and example source code (on the CD-ROM)
5. The *Java Virtual Machine Resources Page* (`http://www.artima.com/insidejvm/resources/`)

An Introduction to Java's Architecture

Chapters 1 through 4 (Part I of this book) give an overview of Java's architecture, including the motivations behind (and the implications of) Java's

architectural design. These chapters show how the Java virtual machine relates to the other components of Java's 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 Java's architecture and a discussion of its inherent tradeoffs, see Chapter 1, "Introduction to Java's Architecture."
- For a discussion of what platform independence really means, how Java's architecture supports this feature, and seven steps to create a platform-independent Java program, see Chapter 2, "Platform Independence."
- For a description of the security model built into Java's core architecture, including an elaborate working example that demonstrates the fine-grained access control made possible by the Version 1.2 security framework, 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 and related core Java APIs. These chapters will help you understand how Java programs actually work. All of the material in Part II is presented in a tutorial manner with many 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 Type."
- For a thorough explanation of Java's linking model, including a tutorial and examples on using `forName()` and class loaders to dynamically extend Java applications with new types at run time, see Chapter 8, "The Linking Model."

- For a discussion of garbage collection and finalization, an explanation of soft, weak, and phantom references, and suggestions on how to use finalizers, see Chapter 9, "Garbage Collection."
- For a tutorial on the Java virtual machine's 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 machine's instruction set, and Appendixes A through C serve as a complete reference of the instruction set. If you need to look up something, check out these chapters and the appendixes.

Interactive Illustrations and Example Source Code

For most of this book's 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-Web site called the "*Interactive Illustrations Web Site*," which includes 15 Java applets that illustrate the concepts presented in the text. These interactive illustrations form an integral part of this book. Eleven of the applets simulate the Java virtual machine by executing bytecodes. The other applets illustrate garbage collection, twos-complement and IEEE 754 floating-point numbers, and the process of 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 *Interactive Illustrations Web Site* enables you to post the Web site on any network, including the Internet—providing that you adhere to a few simple rules. For example, you must post the Web site in its entirety (you cannot make any changes to it), and you cannot charge peo-

ple to look at the site. The full text of the copyright notice is given in the introduction to this book.

All of 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 for yourself.

Most of the example code is for illustrative purposes and is not likely to be of much practical use besides helping you 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) format. The full text of the copyright notice for the example source code is shown in the introduction.

The Java Virtual Machine Resources Page

To help you find more information and keep abreast of changes, I maintain several pages at artima.com with links to further reading about the material presented in this book. The main URL for these pages of links is the Java Virtual Machine Resources Page at `http://www.artima.com/insidejvm/resources/`.

Chapter-by-Chapter Summary

Part I: Java's Architecture

Chapter 1: Introduction to Java's Architecture This chapter gives an introduction to Java as a technology and gives an overview of Java's architecture, discusses why Java is important, and examines Java's pros and cons.

Chapter 2: Platform Independence This chapter shows how Java's architecture enables programs to run on any platform, discusses the factors that determine the true portability of Java programs, and examines the relevant tradeoffs.

Chapter 3: Security This chapter gives an in-depth overview of the security model built into Java's core architecture and traces the evolution of

Java's security model, from the basic sandbox of Version 1.0 through the code signing and authentication of Version 1.1, to the fine-grained access control of Version 1.2.

Chapter 4: Network Mobility This chapter examines the new paradigm of network-mobile software heralded by the arrival of Java and shows how Java's architecture makes this functionality possible.

Part II: Java Internals

Chapter 5: The Java Virtual Machine This chapter gives a detailed overview of the Java virtual machine's internal architecture. Accompanying the chapter on the CD-ROM is an applet called *Eternal Math*, which simulates the Java virtual machine by 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 for the Java class file format. Accompanying the chapter on the CD-ROM is an applet called *Getting Loaded*, which simulates the process of 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 type's initial entrance into the virtual machine to its ultimate exit. The chapter discusses the processes of loading, linking, and initialization; object instantiation, garbage collection, and finalization; and type unloading.

Chapter 8: The Linking Model This chapter takes an in-depth look at Java's linking model and describes the parent-delegation model of class loaders, constant pool resolution, name spaces, and loading constraints. The chapter also shows how to use `forName()` and 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, including a discussion of the train algorithm and soft, weak, and phantom references. Accompanying this chapter on

the CD-ROM is an applet called *Heap of Fish*, which 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 called *Fibonacci Forever*, which 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 called *Conversion Diversion,* which simulates the Java virtual machine's execution of a method that performs type conversion.

Chapter 12: Integer Arithmetic This chapter describes integer arithmetic in the Java virtual machine, explains twos-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, called *Inner Int*, enables you to view and manipulate a twos-complement number. The other applet, called *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 called *Logical Results*, which simulates the Java virtual machine's execution of a method that 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 specification. Accompanying this chapter on the CD-ROM are two applets that interactively illustrate the material presented in the chapter. One applet, called *Inner Float*, enables you to view and manipulate the individual components that make up a floating-point number. The other applet, called *Circle of Squares*,

simulates the Java virtual machine's execution of a method that uses several 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 called *Three-Dimensional Array*, which simulates the Java virtual machine's execution of 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 called *Saying Tomato*, which simulates the Java virtual machine's execution of 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 and 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 called *Play Ball!*, which 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 and describes the relevant instructions with examples of their use. The chapter also describes some surprising behavior 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 called *Hop Around*, which 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. The chapter also shows how one aspect of monitors, the locking and unlocking of data, is supported in the instruction set.

Appendix A: Instruction Set by Opcode Mnemonic This appendix lists the opcodes alphabetically by mnemonic. For each opcode, you are given the mnemonic, 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 instruction's execution. 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 in which 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, you are given the mnemonic.

Appendix D: Slices of Pi: A Simulation of the Java Virtual Machine This final appendix describes one final applet, *Slices of Pi*, that is part of the *Interactive Illustrations Web Site*. This applet simulates the Java virtual machine calculating pi.

Copyright Notices

Here is the text of the copyright notice that appears in each of the example source files (any item on the CD-ROM that is not in either the `applets` or `jdk` directories):

> Copyright© 1997-1999 Bill Venners. All rights reserved.
>
> Source code file from the book "Inside the Java 2 Virtual Machine," by Bill Venners, published by McGraw-Hill, 1997-1999, ISBN: 0-07-135093-4.
>
> This source file may not be copied, modified, or redistributed EXCEPT as allowed by the following statements: You may freely use this file for your own work, including modifications and distribution in compiled (class files, native executable, etc.) form only. You may not copy and distribute this file. You may not remove this copyright notice. You may not distribute modified versions of this source file. You may not use this file in printed media without the express permission of Bill Venners.
>
> BILL VENNERS MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WAR-

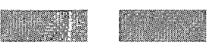
RANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSE, OR NON-INFRINGEMENT. BILL VENNERS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY A LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.

The HTML pages (including the applets) and Java source files for the *Interactive Illustrations Web Site* (stored in the `applets` directory of the CD-ROM) all bear the following copyright notice:

All the web pages and Java applets delivered in the `applets` directory of the CD-ROM, consisting of ".html," ".gif," ".class," and ".java" files, are copyrighted © 1996, 1997 by Bill Venners, and all rights are reserved. This material may be copied and placed on any commercial or non-commercial web server on any network (including the internet) provided that the following guidelines are followed:

a. All the web pages and Java Applets (".html," ".gif," ".class," and ".java" files), including the source code, that are delivered in the `applets` directory of the CD-ROM that accompanies the book must be published together on the same web site.
b. All the web pages and Java Applets (".html," ".gif," ".class," and ".java" files) must be published "as is" and may not be altered in any way.
c. All use and access to this web site must be free, and no fees can be charged to view these materials, unless express written permission is obtained from Bill Venners.
d. The web pages and Java Applets may not be distributed on any media, other than a web server on a network, and may not accompany any book or publication.

BILL VENNERS MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSE, OR NON-INFRINGEMENT. BILL VENNERS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY A LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.



Some Terminology

In this book, I have attempted to use terminology that is consistent with the Java language and Java virtual machine specifications. In case you

are not familiar with this terminology, I would like to clarify a few terms up front.

First of all, in this book I have attempted to be meticulous about my usage of the terms type and class. In Java jargon, variables and expressions have *type*. Objects and arrays have *class*. Every variable and expression in a Java program has a type that is known at compile time: either a primitive type (`int`, `long`, `float`, `double`, etc.) or a reference type (a class, interface, or array). The type of a variable or expression determines the range and kind of values it can have, the operations it supports, and the meaning of those operations.

At run time, every object and array has a class. Although an object is an instance of its class and all of its superclasses, it only *has* one class. An object's class can be any of the following:

- The class mentioned in the class instance creation expression that created the object
- The class represented by the `Class` object upon which `newInstance()` was invoked to create the object
- The class of the object upon which `clone()` was invoked to create the object
- The class of an object that was created by deserializing a previously serialized object

Array classes have names such as `[D` or `[[[[I`, which are not valid identifiers in the Java language. (Array class names are described in Chapter 6, "The Java Class File.") If at run time a variable that has a reference type is not `null`, then that variable refers to an object whose *class* is compatible with the *type* of the variable.

To complicate the terminology situation a bit more, the specifications contain one other usage of the term *type*. Because a variable can declare a class or interface as its type, classes and interfaces define new types for the program to use. (The capability to define new types is, of course, one of the fundamental concepts of object-oriented programming.) Throughout this book, I attempt to use the term *classes* to mean just classes (not classes and interfaces). Likewise, I use the term *interfaces* to mean just interfaces. When I want to refer to both, I sometimes say *classes and interfaces*, but often I just say *types*. For example, when I say, "When the class loader loads a new *type* . . . ," I mean, "When the class loader loads a new *class or interface*." In this sense, type is not referring to the compile-time notion of a variable's type; rather, it refers to the new type that each class and interface definition represents.

Another set of terminology I would like to clarify up front are the terms used by the Java specifications to describe the relationships between types (classes and interfaces) in an inheritance hierarchy. Consider the inheritance hierarchy shown in the Figure 0-1. In this figure, class `CockerSpaniel` extends class `Dog`, which extends class `Animal`, which extends class `Object`. In addition, interface `Friendly` extends interface `Happy`, and class `Dog` implements interface `Friendly`.

In Java terminology, classes higher than a class in an inheritance hierarchy are *superclasses*; classes beneath a class are *subclasses*. In the following figure, `Dog`'s superclasses are `Animal` and `Object`, and `Dog` is a subclass of both `Animal` and `Object`. The superclass that is directly higher than a class in the inheritance hierarchy is the class's *direct superclass*. A subclass directly beneath a class is its *direct subclass*. For exam-

Figure 0-1
An inheritance hierarchy

ple, `Animal` is the direct superclass of `Dog`, and `CockerSpaniel` is a direct subclass of `Dog`.

This sub- and super-terminology can be applied to interfaces, as well. For example, interfaces `Happy` and `Friendly` are *superinterfaces* of both `Dog` and `CockerSpaniel`. Interface `Friendly` is a *direct superinterface* of `Dog` and a *direct subinterface* of `Happy`.

One last way to use the sub- and super-terminology is by grouping both classes and interfaces under the name *type*. In Figure 0-1, `Friendly`, `Dog`, and `CockerSpaniel` are *subtypes* of `Happy`. `Object`, `Animal`, `Happy`, `Friendly`, and `Dog` are all *supertypes* of `CockerSpaniel`.

Font Conventions Used in this Book

Throughout this book, I use a fixed-width font for Java code and Java virtual machine opcode mnemonics. In the text, I use fixed-width font for Java language keywords only in certain cases—in an attempt to maximize readability. For example, I say public method instead of `public` method, because in this case, public is being used as a regular English adjective (in a sense of the word that is understood in Java circles)—not necessarily as the Java keyword.

Java Versions and Specification Editions

The text of this book is current to the Java 2 SDK Version 1.2 and the second edition of the Java virtual machine specification. Although little of the material covered by this book changed between Versions 1.0 and 1.1, a great deal changed between Versions 1.1 and 1.2. Moreover, the second edition of the Java virtual machine specification clarified many issues contained in the first edition of the specification, as well as making a few amendments.

One change that occurred in JDK Version 1.0.2 was a change in the semantics of the `invokespecial` instruction, which is described in Chap-

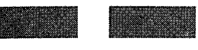
ter 19, "Method Invocation," and in Appendix A. Two attributes were added to the class-file format in Version 1.1 to support inner classes, and an attribute was added to support the `@deprecated` `javadoc` tag. They are described in Chapter 6, "The Java Class File." Also, the API of the `ClassLoader` class was extended in Version 1.1. Chapter 8, "The Linking Model," demonstrates the use of the 1.1 version of this API.

Several of the Java APIs described in this book underwent significant changes between Version 1.1, which was covered by the first edition of this book, and Version 1.2, which is covered by this second edition. Perhaps the most significant API changes that affected this book are the many API changes that support the Version 1.2 security model. All of the components of the Version 1.2 security model—the basic sandbox, code signing and authentication, policies and policy files, permissions, code sources, protection domains, and the stack inspection algorithms of the access controller—are described in detail in Chapter 3, "Security." The `strictfp` keyword added to the Java language in Version 1.2 and the corresponding access flag added to the Java class-file format are explained in Chapter 6, "The Java Class File." The class-loader parent-delegation model introduced in Version 1.2—and several new methods introduced in Version 1.2 to classes `java.lang.Class` and `java.lang.ClassLoader`—are described in Chapter 8, "The Linking Model." Soft, weak, and phantom references, which were added as the `java.lang.ref` package of the Version 1.2 Java API, are described in Chapter 9, "Garbage Collection."

Aside from several new APIs introduced to Java in Version 1.2, this book incorporates the many clarifications and amendments to the original Java virtual machine specification that were printed in the second edition of the specification. For example, the second edition of the Java virtual machine specification documented a new set of loading constraints that ensure type-safe linking in the presence of multiple class loaders. These loading constraints are described and are demonstrated by a code example in Chapter 8, "The Linking Model." The revised floating-point rules for Java virtual machines given in the second edition of the specification are explained in Chapter 14, "Floating-Point Arithmetic." This second edition of this book also incorporates many corrections and clarifications to the specification of class-file version numbers, method invocation, and the loading, linking, and initialization of types.

The bytecode examples shown throughout this book were generated by the `javac` compiler from various incarnations of Sun's JDK Version 1.1. Keep in mind that there is more than one way to compile a class. Different compilers, even different versions of the same compiler, could generate different results.

The source code of the simulation applets (the interactive illustrations) adhere to Java Version 1.0. As I discuss in Chapter 2, "Platform Independence," one of the realities of Java's platform-independence promise is that you have to decide when a version of the Java Platform has been distributed widely enough to make it worthwhile to target that version. Although I had a 1.1 version of the Java virtual machine simulator applets working in 1997, when it came time to deliver the CD-ROM material for the first edition of this book to the publisher, I opted to drop the code back to Version 1.0. At the time, neither Netscape Communicator nor Microsoft Internet Explorer fully supported Version 1.1. Because these applets are not example source but are software products in their own right, I felt that it did not make sense to release them in Version 1.1. As a consequence, the applets will work in browsers that support either Versions 1.0, 1.1, or 1.2—and hopefully many versions into the future.



Request for Comments

If you have a suggestion on how to improve this book or wish to report a bug or error, please visit `http://www.artima.com/insidejvm/feedback.html`. This page will give you instructions on how to submit your comment.

ACKNOWLEDGMENTS

I'd like to thank Michael O'Connell, the former editor-in-chief of JavaWorld, for giving me the opportunity to write about Java. I'd also like to thank Jill Steinberg, the current editor-in-chief, and the rest of the gang at JavaWorld for all their help with my original JavaWorld column, "Under the Hood."

Thanks to my agent, Laura Belt, of Adler & Robin, who—after reading one of my JavaWorld columns back in 1996, e-mailed me and inquired whether I'd like to write a book—and then phoned me within five minutes of my pressing the reply button.

As this second edition owes much to the first, I'd like to once again thank some people who helped me with the first edition. Thanks again to Judy Brief, my editor at McGraw-Hill, for the first edition. She helped me reach the finish line the first time around. Thanks to Tim Lindholm and Jeff Rice, both of whom navigated through the manuscript of the first edition in search of technical bugs. Thanks also to the local reviewers of my first edition: Siew Chuah, Terrin Eager, Peter Eldredge, Steve Engle, Matt Gerrans, Mark Johnson, Barbara Laird, Steve Schmidt, and Anil Somani—all of whom read partial drafts of the manuscript. Your feedback and moral support were invaluable. Also, a special thanks goes to the Coffee Society of Cupertino, who cheerily welcomed my reviewers and me on many a Wednesday night, although none of us were young and few of us were pierced.

Thanks to everyone who submitted errata reports for the first edition, especially Antoine Trux, who found the majority of them.

Thanks to Kurt and Heidi Sohn, who let me invade their vacation home in Adrazhofen, Germany, where I completed the updates to Chapters 8 and 14.

Thanks goes also to Li Gong of Sun Microsystems, who reviewed the new security chapter.

Thanks very much to Kee Yong Chuah, who set me up with an Internet dial-up account that worked in Malaysia, where I happened to be when the page proofs needed author review.

Thanks to those at D&G Limited, who helped form my manuscript into the book you're holding: Alan Harris, project manager; Kelly Dobbs, production manager; and Claudia Bell, layout technician.

Finally, thanks to Simon Yates, my editor at McGraw-Hill, and Jennifer Perillo, managing editor, for all their assistance and patience as I worked to complete this project.

CHAPTER 1

Introduction to Java's Architecture

At the heart of Java technology lies the Java virtual machine—the abstract computer on which all Java programs run. Although the name "Java" is generally used to refer to the Java programming language, there is more to Java than just the language. The Java virtual machine, Java *Application Programming Interface* (API), and Java class file work together with the language to make Java programs run.

The first four chapters of this book (collectively called "Part I: Java's Architecture") show how the Java virtual machine fits into the big picture. These chapters show how the virtual machine relates to the other components of Java's 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, offers an overview of Java's architecture, discusses why Java is important, and examines Java's 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, these microprocessors 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, cellular phones, personal digital assistants, pagers, and wristwatches all have been enhanced with microprocessors, and 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 for software developers. Java is well suited to help software developers meet challenges and seize opportunities presented by the emerging computing environment, because 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 software developers by the increasingly network-centric hardware 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 cre-

ation 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 customized degrees of security.

One aspect of security is simple program robustness. Like devious code written by malicious programmers, bug-filled code written by well-meaning programmers can potentially destroy information, monopolize compute cycles, or cause systems to crash. Java's architecture guarantees a certain level of program robustness by preventing certain types of pernicious bugs, such as memory corruption, from ever occurring in Java programs. This architecture establishes trust that downloaded code will not inadvertently (or intentionally) crash, but it also has an important benefit unrelated to networks: the architecture makes programmers more productive. Because Java prevents many types of bugs from ever occurring, Java programmers do not need to 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. This transmission can also simplify version control. Because the most recent version of a Java program can be delivered on demand across a network, you do not need to worry about which version your end-users are running. They will always get the most recent version each time they use your program.

Mobile code gives rise to another opportunity: mobile objects, the transmission of both code and state across the network. Java realizes the promise of object mobility in its APIs for object serialization and *Remote Method Invocation* (RMI). Built on top of Java's underlying architecture, object serialization and RMI provide an infrastructure that enables the various components of distributed systems to share objects. The network mobility of objects makes possible new models for distributed systems programming, effectively bringing the benefits of object-oriented programming to the network.

Platform independence, security, and network mobility are three facets of Java's architecture that work together to make Java suitable for the emerging network computing environment. Because Java programs are platform independent, network mobility of code and objects is more practical. The same code can be sent to all the computers and devices that the network interconnects. Objects can be exchanged between various components of a distributed system, which can be running on different kinds of hardware. Java's built-in security framework also helps make network mobility of software more practical. By reducing risk, the security framework helps to build trust in a new paradigm of network-mobile software.

The Architecture

Java's architecture arises from four distinct (but interrelated) technologies:

- the Java programming language
- the Java class file format
- the Java API
- the Java virtual machine

When you write and run a Java program, you are tapping into 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 API. As your program runs, it fulfills your program's 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.

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* (or, starting with version 1.2, the *Java 2 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.

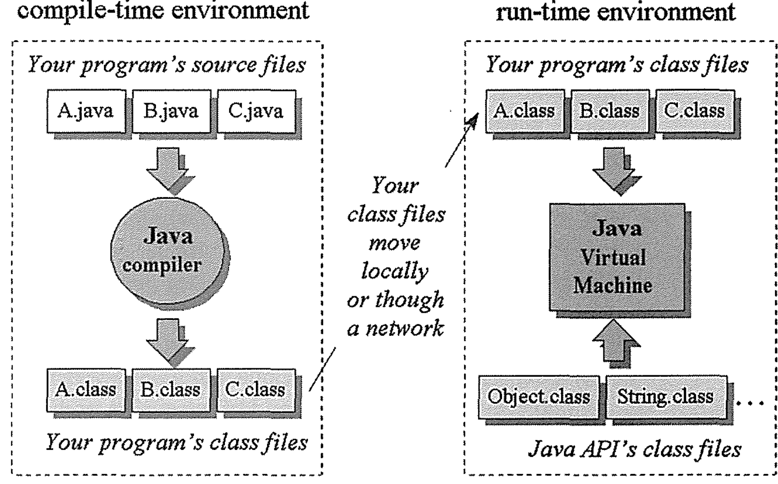


Figure 1-1 The Java programming environment

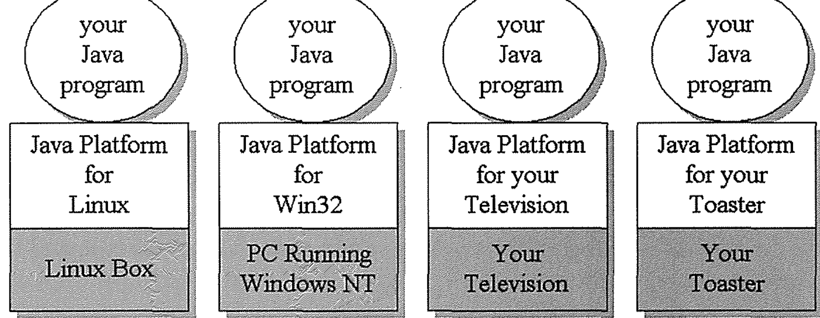


Figure 1-2 Java programs run on top of the Java Platform.

The Java Virtual Machine

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

The Java virtual machine is an abstract computer. Its specification defines certain features every Java virtual machine must have but leaves many choices to the designers of each implementation. For example, although all Java virtual machines must be able to execute Java bytecodes,

they may use any technique to execute them. Also, the specification is flexible enough to enable a Java virtual machine to be implemented either completely in software—or to varying degrees in hardware. The flexible nature of the Java virtual machine's specification enables it to be implemented on a wide variety of computers and devices.

A Java virtual machine's main job is to load class files and execute the bytecodes they contain. As you can see in Figure 1-3, the Java virtual machine contains a *class loader*, which loads class files from both the program and the Java API. Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine. The bytecodes are executed in an *execution engine*.

The execution engine 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 the code can be reused the next time that same method is invoked. A third type of execution engine is an *adaptive optimizer*. In this approach, the virtual machine starts by interpreting bytecodes but monitors the activity of the running program and identifies the most heavily used areas of code. As the program runs, the virtual machine compiles to native machine code and optimizes only these heavily used areas. The rest of the of code, which is not heavily used, remains as bytecodes—which the virtual machine continues to interpret. This adaptive optimization approach enables a Java virtual machine to spend typically 80 percent to

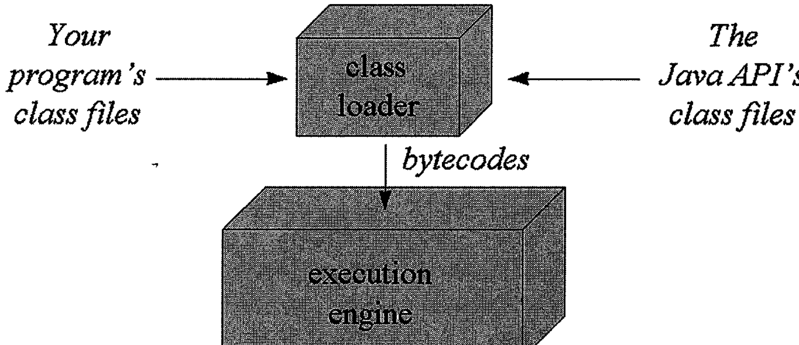


Figure 1-3
A basic block diagram of the Java virtual machine

90 percent of its time executing highly optimized native code, while requiring it to compile and optimize only the 10 percent to 20 percent of the code that really matters for performance. Lastly, 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.

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" may seem to imply that a virtual machine is interpreting bytecodes, the term "interpreter" is really being used in a different sense in this case. When talking about execution techniques, interpreting is a particular technique known for its easy implementation and slow execution. But "Java interpreter" just means "Java virtual machine," and says nothing about execution technique.

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 is 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 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.

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, 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* (JNI), enables native methods to work with any Java Platform implementation on a particular host computer. Vendors of the Java Platform, however, are not necessarily required to support JNI. They may provide their own proprietary native method interfaces in addition to JNI (or, depending on their contract, 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 access the system resources of the underlying operating system only through the Java API.

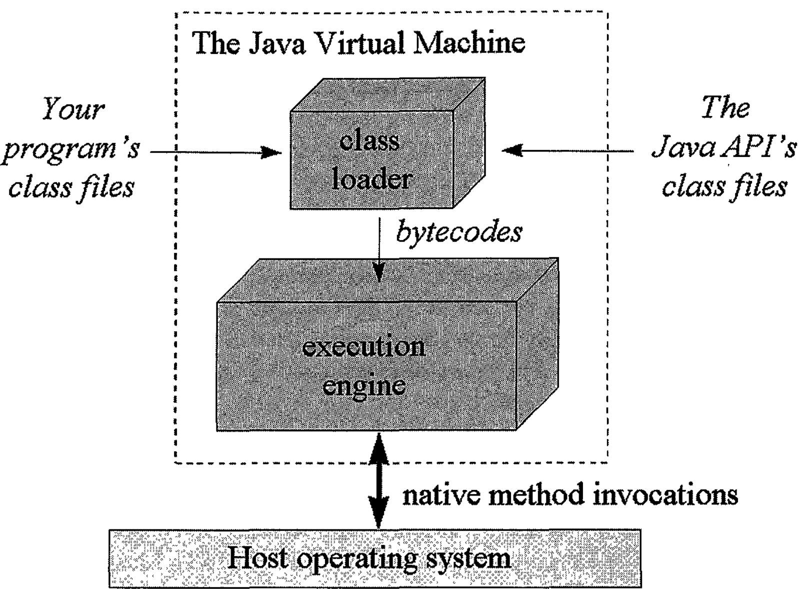


Figure 1-4 A Java virtual machine implemented in software on top of a host operating system

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 (shown previously), a single mysterious cube identifies itself as "the class loader." In reality, though, 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 enables a Java application to load classes in custom ways.

A Java application can use two types of class loaders: a "bootstrap" class loader and user-defined class loaders. The bootstrap class loader (there is only one of them) is 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 bootstrap class loader will be part of that C program. The bootstrap class loader loads classes, including the classes of the Java API, in some default way, usually from the local disk. (The bootstrap class loader has also been called the primordial class loader, system class loader, or default class loader. In

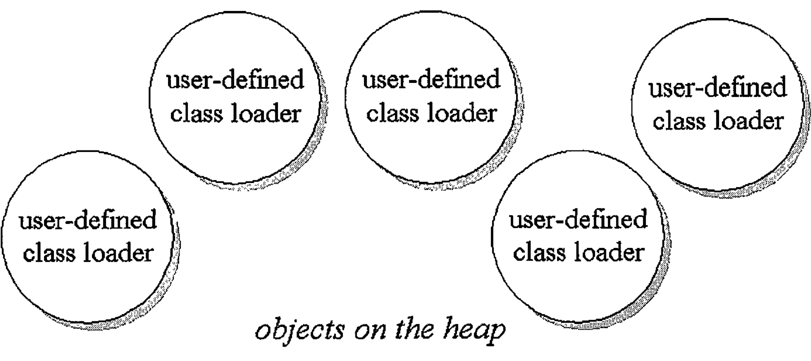
version 1.2, the name "system class loader" was given a new meaning (described in Chapter 3, "Security").

At run time, a Java application can install user-defined class loaders that load classes in custom ways, such as by downloading class files across a network. While the bootstrap class loader is an intrinsic part of the virtual machine implementation, user-defined class loaders are not. Instead, user-defined class loaders 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.

Because of user-defined class loaders, at compile time you do not have to know all the classes that may ultimately take part in a running Java application. User-defined class loaders enable you to dynamically extend a Java application at run time. As the application runs, it can determine what extra classes are needed and load them through one or more user-defined class loaders. Because you write the class loader in Java, you can load classes in any manner expressible in Java code. 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 bootstrap or user-defined—loaded the class. When

Figure 1-5
Java's class loader architecture



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 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, by default classes can only see other classes that were loaded by the same class loader. In this way, Java's architecture enables you to create multiple *name-spaces* inside a single Java application. Each class loader in your running Java program has its own name-space, which is populated by the names of all the classes it has loaded.

A Java application can instantiate multiple user-defined class loaders either from the same class or from multiple classes. The application can, therefore, create as many (and as many different kinds of) user-defined class loaders as necessary. Classes loaded by different class loaders are in different name-spaces and cannot gain access to each other, unless the application explicitly permits this access. When you write a Java application, you can segregate classes loaded from different sources into different name-spaces. In this way, you can use Java's class loader architecture to control any interaction between code loaded from different sources. In particular, you can prevent hostile code from gaining access to and subverting friendly code.

One example of dynamic extension is the Web browser, which uses user-defined class loaders to download the class files for applets across a network. A Web browser fires off a Java application that installs a user-defined class loader—usually called an *applet class loader*—that knows how to request class files from a *HyperText Transport Protocol* (HTTP) server. Applets are an example of dynamic extension, because at startup, the Java application does not know 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 user-defined class loader for each location on the network from which it retrieves class files. As a result, class files from different sources are loaded by different user-defined class loaders. This action 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 enabling you to instantiate user-defined class loaders that know how to download class files across a network, Java's class loader architecture supports network mobility. Java also supports security by enabling you to load class files from different sources through different user-defined class loaders. This feature puts the class files from different sources into different name-spaces, which enables 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. This form is expected by the Java virtual machine but is independent of underlying host platforms. This approach breaks with the tradition followed by languages such as C or C++, because 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 will not work on another platform. 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, which are 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—regardless of which platform generated the file and independent of whatever platforms may eventually use the file.

In addition to its support for platform independence, the Java class file plays a critical role in Java's 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 can be extended dynamically, 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-user's wait time can be kept to a minimum.

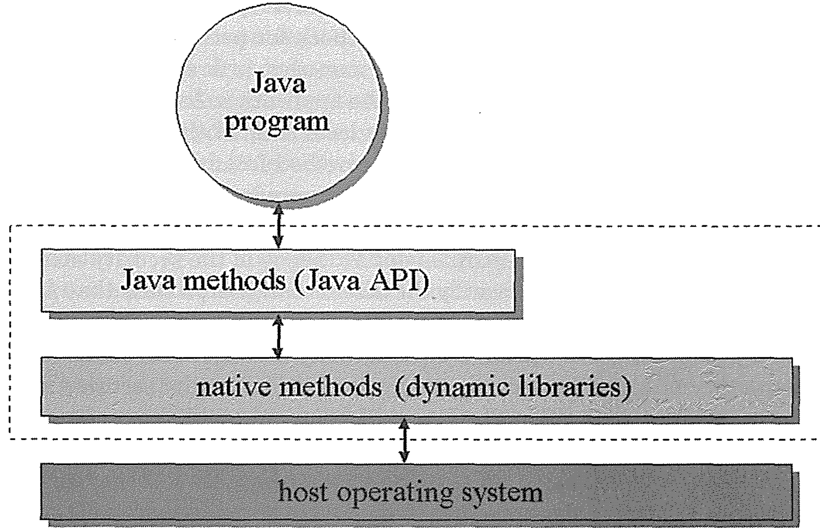
The Java API

The Java API helps make Java suitable for networks through its support for platform independence and security. The Java API is a set of run-time libraries that give you a standard way to access the system resources of a host computer. When you write a Java program, you assume that 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 assumption is relatively safe, 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 program's 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 API's functionality must be implemented expressly for a particular platform before that platform can host Java programs. To access the native resources of the host, the Java API calls native methods. As you can see in Figure 1-6, the class files of the Java API invoke native methods so your Java program doesn't have to do this task. In this manner, the Java API's 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.

The internal design of the Java API is also geared towards platform independence. For example, the *graphical user interface* (GUI) libraries of the Java API, the *Abstract Windows Toolkit* (AWT), and Swing are designed to facilitate the creation of user interfaces that work on all platforms. Creating platform-independent user interfaces is inherently diffi-

Figure 1-6
A platform-independent Java program



cult, given that the native look and feel of user interfaces vary greatly from one platform to another. The AWT library's architecture does not coerce implementations of the Java API to give Java programs a user interface that looks exactly the same everywhere. Instead, the architecture encourages implementations to adopt the look and feel of the underlying platform. The Swing library offers even more flexibility, enabling the look and feel to be chosen by the programmer. Also, because the size of fonts, buttons, and other user-interface components will vary from platform to platform, the AWT and Swing include *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 a dialog box, for example, the layout manager positions the coordinates when your dialog box is displayed. With the aim of making the dialog look its best on each platform, the layout manager will likely position the dialog box elements slightly different on different platforms. In these ways and in many others, the internal architecture of the Java API is aimed at facilitating the platform independence of the Java programs that use the application.

In addition to facilitating platform independence, the Java API contributes to Java's security model. Before they perform any action that

could potentially be harmful (such as writing to the local disk), the methods of the Java API check for permission by querying the *security manager*. The security manager is a special object 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. In Java 1.2, the job of the security manager was in many ways taken over by the *access controller*, a class that performs stack inspection to determine whether the operation should be permitted. (For backwards compatibility, the security manager still exists in Java 1.2.) By enforcing the security policy established by the security manager and access controller, 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. Platform 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 obtain 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 enables 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 had already been tried and proven in other languages. These concepts and techniques make the Java programming language a powerful, general-purpose tool that you can apply to a variety of situations—independent of whether 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 potentially slower execution speed.

First and foremost, Java is an object-oriented language. One promise of object-orientation is that it promotes the reuse of code, resulting in better productivity for developers. This feature may make Java more attractive than a procedural language such as C but does not add much value to Java over C++. Yet, compared to C++, Java has some significant differences that can improve a developer's productivity. This productivity boost comes mostly from Java's 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 cannot cast the reference to type `Lava` and manipulate the memory as if it were a `Lava`, nor can you simply add an arbitrary offset to the reference (as pointer arithmetic permits you to do in C++). In Java, you can 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 the program. As a result, you can never create certain kinds of bugs in Java programs that regularly harass C++ programmers and hamper their productivity.

Another way that 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 situation results in memory leaks, where unused objects are never freed, and memory corruption, where the same object is accidentally freed multiple times. Both kinds of memory troubles cause C++ programs to crash, but tracking down the exact source of the problem is difficult. You can be more productive in Java primarily because you do not have to chase down memory corruption bugs. You can also be more productive, however, because when you no longer have to worry about explicitly freeing memory, program design becomes easier.

A third way that 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++ enables you to declare an array of 10 items, then write to the 11th item (although that action 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 10 items in Java and try to write to the 11th, Java will throw an exception. Java will not 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).

Architectural Tradeoffs

Although Java's network-oriented features are desirable, especially in a networked environment, they do 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. Java is suitable for solving problems that involve networks and has functionality in many problems that do not involve networks, but its architectural tradeoffs will disqualify it for certain types of jobs.

One of the prime costs of Java's network-oriented features is a potential reduction in program execution speed compared to other technologies such as C++. Indeed, achieving satisfactory performance was one of the most frustrating struggles for Java developers in the first few years of Java's existence. Nevertheless, although the early experience with Java may have encouraged the developer community to conclude that Java is slow, this was not necessarily the right conclusion. Although Java can be slow, it is not *inherently* slow. As virtual machine technology has advanced, great strides have been made in performance—even so far as to bring Java performance on par with natively compiled C.

The first Java virtual machine that appeared in 1995 executed bytecodes with an interpreter—a simple technique that yields poor performance. Before long, just-in-time compilers appeared that greatly improved Java's performance compared to interpreters, but they still left Java performance well behind natively compiled C++. With the most recent advances in virtual machine technology, however, Java's speed penalty is diminishing significantly, if not vanishing altogether. Advanced techniques such as adaptive optimization have enabled Java programs to run at speeds comparable to natively compiled C.

Although the recent advances in Java performance are good news, they do not necessarily signal the end of developer frustrations about Java performance. The trouble for developers is that although certain Java virtual machine implementations may yield stunning performance, developers cannot always select the virtual machine on which their Java programs will run. One of the promises of Java's architecture is that a Java program will run "anywhere," and that also means on any Java virtual machine. If you are writing a server application in Java intended for in-house use, you may be able to select the virtual machine implementation on which your application will run. But as soon as you have multiple customers for your Java program, you will likely need to get your program to have acceptable performance on many virtual machine implementations. Also, in a world consisting of the kind of distributed systems encouraged by Java's architecture (with code and objects flying over the network from one virtual machine to another),

developers basically lose all control over the virtual machine implementations on which their programs will run.

Ultimately, whether or not performance will be a problem for you and how you would go about dealing with that problem depends on what exactly you are trying to do. Fortunately, Java is a flexible tool, giving you many ways to deal with potential performance troubles. If, for example, what you need to provide is a monolithic executable (such as a word processor or server process), you could do the following tasks:

- Deliver a virtual machine along with your program.
- Implement time-critical sections of your program as native methods.
- Compile the whole program to a monolithic executable in the tradition of C and C++.
- Compile to a monolithic executable at the end-user's machine at install time.

Probably the most powerful way to manage performance of a monolithic application is by being able to pick the virtual machine yourself. Executing part or all of your program natively may be the best approach in some situations, however.

Compiling a Java program to a monolithic executable, which is sometimes referred to as "ahead-of-time compiling," can help improve performance—but usually at the cost of making it impossible for the program to use Java's dynamic extension capabilities. Ahead-of-time compiling performs static, not dynamic, linking and yields fully linked, monolithic native executables that do not usually have the capability to bring in and dynamically link to new types at run time. For Java programs that would not use dynamic extension anyway, however, ahead-of-time compiling should yield an executable program that behaves exactly like the program would if executed on a traditional virtual machine. Because many embedded systems have no need for dynamic extension and usually have resource constraints, ahead-of-time compiling is often used to compile a Java program to a native executable image that can be burned into *Read-Only Memory* (ROM) for an embedded system. Ahead-of-time compiling can also be used for a desktop application, as long as it does not use dynamic extension. If you are struggling to solve performance problems of a relatively stand-alone Java program that does not use dynamic extension, ahead-of-time compiling may be able to help.

Managing performance becomes more difficult, however, when you are developing not a monolithic application but a distributed system—especially one in which code and objects will be moving from virtual

machine to virtual machine. This kind of object-oriented network programming is, after all, one of the big promises of Java's architecture. In such cases, the best way to manage performance is in the way you design your system. Here, you must resort to traditional mechanisms for improving performance, such as minimizing network traffic and selecting the best algorithm, and to other standard approaches for performance tuning in any language.

Although program speed is a concern when you use Java, there are ways you can address this issue. By appropriate use of the various techniques for developing, delivering, and executing Java programs, you can often satisfy end-users' 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.

Besides performance, another tradeoff of Java's network-oriented architecture is the lack of control of memory management and thread scheduling. Garbage collection can help make programs more robust, which is a valuable security guarantee in a networked environment. But garbage collection adds a level of uncertainty to the run-time performance of the program. You cannot always be sure when or if a garbage collector will decide it is time to collect garbage or how long the process will take. In addition, the Java virtual machine specification discusses thread scheduling in only general terms. This looseness in the specification of thread behavior helps make it easier to port the Java virtual machine to many different kinds of hardware. Although virtual machine portability is important in a networked environment, the vague specification of thread scheduling leaves programmers with little knowledge and no control of how their threads will be scheduled. This lack of control of memory management and thread scheduling makes Java a questionable candidate for software problems that require a real-time response to events.

Still another tradeoff arises from Java's 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 the feature anyway. This task 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 users on the affected operating systems.

What is worse, not only does the lowest-common-denominator problem afflict the designers of a platform-independent API, but 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. Nevertheless, you might find it difficult to design a user interface in which the components interact in a way that *feels* native on every platform, although 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. The Swing library gives you more options, but ultimately you still have to wrestle with differences in end-user expectations when you design a cross-platform user interface.

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 field's name and *descriptor* (the field's type) are also specified. If the reference is to a method, the method's name and descriptor (the method's return type and 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, but 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 file's 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 feature, 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 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.

Conclusion

So, what is the main point of Java's architecture? As shown in this chapter, the Java programming language is a general-purpose tool that has distinct advantages over other technologies. In particular, Java can yield better programmer productivity and improved program robustness—with, for the most part—acceptable performance, compared to older programming technologies such as C and C++. Yet, the main focus of the design of Java's architecture was not just to make programmers more productive and programs more robust, but to provide a tool for the emerging network-centric computing environment. Java's architecture paves the way for new network-oriented software architectures that take full advantage of Java's support for network mobility of code and objects.

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/resources`.

CHAPTER 2

Platform Independence

The previous chapter showed how Java's architecture makes it a useful tool for developing software in a networked environment. The next three chapters take a closer look at how Java's architecture accomplishes its suitability for networks. This chapter examines platform independence in detail, shows how Java's 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 why 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 feature is important in a networked environment, because networks usually interconnect many different kinds of computers and devices. In a typical enterprise environment, for example, a network might connect Macintoshes in the art department, UNIX workstations in engineering, and PCs running Windows everywhere else. 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 computer's type, make the system administrator's job simpler—especially if those programs can actually be delivered across the network.

In addition, the emerging proliferation of network-enabled, embedded devices represents another environment in which Java's platform independence is useful. In the workplace, for example, various kinds of embedded devices, such as printers, scanners, and fax machines, are typically connected to the internal network. Network-connected, embedded devices have also appeared in consumer domains, such as in the home and in the car. In the embedded world, Java's platform independence can also help simplify system administration. Jini technology, which aims to bring plug and play to the network, simplifies the task of administering a dynamic environment of network-connected, embedded devices for consumers at home and for systems administrators at work. Once a device is plugged into the network, it can access other devices attached to the network. Other devices can access it, as well. To achieve this ease of connectivity, Jini-enabled devices exchange objects across the network—a technique that would be impossible without Java's support for platform independence.

From the developer's perspective, Java can reduce the cost and time required to develop and deploy applications on multiple platforms. Although historically, many (or most) applications have been supported on only one platform, often the reason was that the cost involved in supporting multiple platforms was not worth the added return. Java can help make multi-platform support affordable for more types of programs.

On the other hand, Java's platform independence can act as a disadvantage, as well as an advantage, for software developers. If you are developing and selling a software product, Java's support for platform independence can

help you compete in more markets. Instead of developing a product that runs only on Windows, for example, you can write one that runs on Windows, OS/2, Solaris, and Linux. 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 Solaris. 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.

But perhaps most significantly for developers, the fact that Java code can run unchanged on multiple platforms gives the network a homogeneous execution environment that enables new kinds of distributed systems built around network-mobile objects. APIs such as object serialization, *Remote Method Invocation* (RMI), and Jini take advantage of this underlying capability to bring object-oriented programming out of the virtual machine and onto the network. (More information on Jini is given in Chapter 4, "Network Mobility.")

Java's Architectural Support for Platform Independence

Support for platform independence, like support for security and network mobility, is spread throughout Java's 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

Java's 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, while the API gives the program access to the underlying computer's resources. No matter where a Java program goes, it only needs to interact with the Java Platform. The program does not need to worry about the underlying hardware and operating system. As a result, the application can run on any computer that hosts a Java Platform.

The Java Language

The Java programming language reflects Java's 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 statement means that a C++ program might have a 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 two's 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 Java's 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.

Although 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 and consumer devices, not for 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 and consumer systems markets. Microprocessors had been appearing in device after device for years—in audio-video equipment, cellular phones, printers, fax machines, and 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 run-time 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. These constraints mean that embedded and consumer systems usually do not have the need or the memory to support the full Java API.

To address the special requirements of embedded and consumer systems, Sun Microsystems, Inc. created several incarnations of the Java Platform with smaller API requirements for embedded and consumer systems:

- the Java Personal Platform (for consumer devices)
- the Java Embedded Platform (for embedded devices)
- the Java Card Platform (for SmartCards)

These Java Platforms are composed of a Java virtual machine and a smaller shell of run-time libraries that are available in the standard Java Platform. The difference between the standard and the Personal Platform, therefore, is that the Personal Platform guarantees the availability of fewer Java API run-time libraries. The Embedded Platform guarantees fewer APIs than the Personal Platform, and the Card Platform guarantees fewer than the Embedded. Yet, although each platform addresses a progressively smaller execution environment with progressively tighter constraints on resources, the APIs are not necessarily subsets of each other. Each API subset is geared towards a particular target and therefore includes just the APIs that make sense for that target.

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.

Although Sun attempted to address the special API needs of the embedded and consumer markets with these three subsets, the special API needs of these markets turned out to be a bit too heterogeneous for the three API subsets to adequately address. Because of the special constraints of embedded systems, especially the small memory footprint and lack of disk storage, vendors of embedded systems are often under tremendous economic pressure to pick and choose APIs. Because of the low price points for embedded devices, vendors often simply cannot afford to include APIs that are not directly needed by their device. Despite the three subsets defined by Sun, vendors still felt the need to define and support their own API subsets.

Eventually, Sun recognized that its three subsets would not suffice and changed its approach to defining API standards for the embedded and consumer worlds. Instead of trying to define "one-API-fits-all" subsets, such as Personal and Embedded Java, Sun defined a minimal API set that it called the *Java 2 Platform, Micro Edition* (J2ME). On top of J2ME, Sun planned to facilitate the definition of API subsets by individual industry segments appropriate for its market niche (such as automobile, TV set-top box, screen phone, wireless pagers and cellular phones, personal digital assistants, etc.). Sun called these API subsets "profiles." The old Personal and Embedded platforms become profiles in the new approach.

Because the Java Platform is compact, it can be implemented on a wide variety of embedded and consumer systems. The potential 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. Although, in Java's early years, Java Virtual Machine implementation had scaling difficulties on the server side. Virtual machines were tuned for servers, and now many implementations yield good performance on the server side. At this end of the spectrum, Sun has defined an API superset: the *Java 2 Enterprise Edition* (J2EE). In addition to the standard Java APIs, the J2EE includes other APIs that are useful in enterprise server environments, such as servlets and Enterprise JavaBeans.

In the end, Sun's revised approach to defining APIs yielded three basic API sets, which demonstrate the scaleability of the Java Platform:

- *Enterprise Edition* (J2EE)
- *Standard Edition* (J2SE)
- *Micro Edition* (J2ME)

At the high end, the existence of the Enterprise Edition signifies the utility of the Java Platform in high-end servers. In the middle, the Stan-

dard Edition carries on the tradition started by applets in browsers of the Java Platform on the desktop. At the low end, the Micro Edition, augmented with industry profiles, shows that the Java Platform can scale down and mold itself to meet the requirements of a great variety of consumer and embedded environments.

Factors that Influence Platform Independence

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

The degree of platform independence of any Java program 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 the program.

Java Platform Deployment

The most basic factor determining a Java program's 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 Alicia's particular type of hardware and operating system. Once the port has been done by some Java Platform vendor, that port must in some way be installed on Alicia's 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 and moving on to desktop, workstation, network operating systems, and many different kinds of consumer and embedded devices. Therefore, it is increasingly likely that your friend Alicia will have a Java Platform implementation on her computer or device.

The Java Platform Version and Edition

The deployment of the Java Platform is a bit more complicated, however, because not all standard run-time 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 *Standard API*. Sun calls a 1.2 Java virtual machine accompanied by the class files that constitute the standard API the *Java 2 Platform, Standard Edition*. This edition of the Java Platform has the minimum set of Java API libraries that you can assume will be available at desktop computers and workstations. But, as described earlier, Sun also defines API sets for the Micro and Enterprise Editions of the Java 2 Platform and encourages the development of API profiles to augment the Micro Edition in various consumer and embedded industry segments. In addition, Sun defines some standard run-time libraries that it considers optional for the Standard Edition and calls these *Standard Extension APIs*. These libraries include services such as telephony and commerce and media such as audio, video, or 3D. If your program uses libraries from the Standard Extension API, the program will run anywhere those standard extension API libraries are available. But the program will not run on a computer that implements only the basic Standard Edition Platform. Some of the Standard Extension APIs, on the other hand, are guaranteed to be available at any implementation of the Enterprise Edition. Given the variety of API editions and profiles, the Java 2 Platform hardly represents a single, homogeneous execution environment that will—in all cases—enable code that is written once to run anywhere.

Another complicating factor is that in a sense, the Java Platform is a moving target because it evolves over time. Although the Java virtual machine is likely to evolve gradually, the Java API will probably change more frequently. Over time, features will be added to and removed from both the Standard Edition and Standard Extension APIs, and parts of the Standard Extension API may migrate into the Standard Edition. The changes made to the Java Platform should, for the most part, be backwards compatible, meaning that they will not break existing Java programs, but some changes may not be backwards compatible. As obsolete features are removed in a new version of the Java Platform, existing Java programs that depend upon those features will not run on the new version. Also, changes may not be forwards compatible, meaning programs that are compiled for a new version of the Java Platform will not 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 2 Platform Standard Edition, as long as you depend only upon the run-time libraries in the standard API. In practice, however, new versions of the standard API will take time to percolate everywhere. When your program depends on newly added features of the latest version of the standard API, there may be some hosts that cannot run the program because they have an older version. This problem is not new to software developers. Programs written for Windows 95, for example, did not work on the previous version of the operating system, Windows 3.1. Because Java enables the network delivery of software, however, this incompatibility 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 the same piece of binary Java code can be sent across the network and run on any computer or device.

As a developer, you cannot control the release cycles or deployment schedules of the Java Platform, but you can choose the Java Platform edition and version upon which your programs depend. 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 to not directly or indirectly invoke any native methods that are not 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.

Calling native methods directly is appropriate in situations where you do not 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 is not written in Java
- For speeding up the performance of a program by implementing time-critical code as native methods

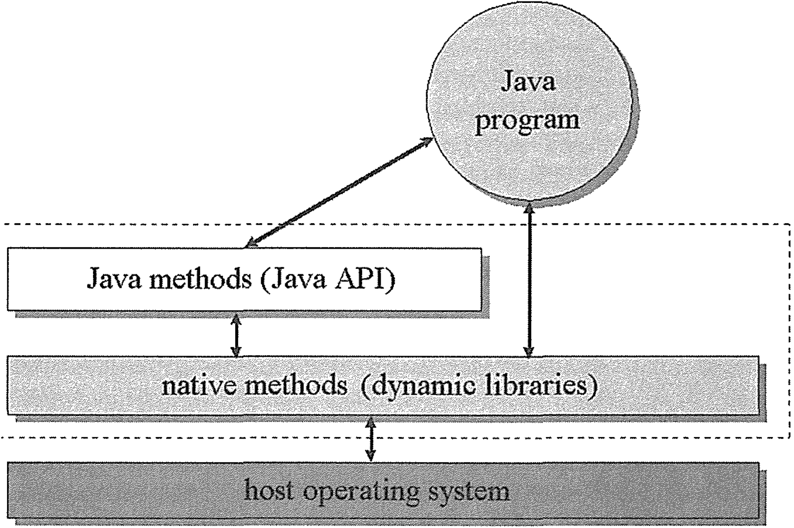


Figure 2-1
A platform-specific Java program

If you need to use native methods and also need your program to run on several platforms, you will have to port the native methods to all the required platforms. This porting must be done the old-fashioned way, and once you have done this task, you will have to figure out how to deliver the platform-specific, native method libraries to the appropriate hosts. Because Java's 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 Run-Time Libraries

Native methods are not inherently incompatible with platform independence. What is 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 the method will be available everywhere. You do not care if the method is implemented as a native method in some places.

Java Platform implementations can come from a variety of vendors, and although every vendor must supply the standard run-time 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 run-time libraries you use call native methods. Non-standard libraries that do not call native methods do not degrade your program's platform independence. Using non-standard libraries that do call native methods, however, yields the same result as calling native methods directly. This usage renders your program platform specific.

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 as follows:

1. Do not depend upon timely finalization for program correctness.
2. Do not 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 machines 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 feature, 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 are 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 prohibit lower-priority threads from running when higher-priority threads are not blocked, however. On some virtual machine implementations, therefore, lower-priority threads may get some CPU time—even when the higher-priority threads are not blocked. If your program depends on correctness of this behavior, however, it may work on some virtual machine implementations but not on others.

To keep your multi-threaded Java program platform independent, you must rely on synchronization—not prioritization—to coordinate interactivity between threads.

User Interface Dependencies

Another major variation between different Java Platform implementations is the 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. The Swing library gives you advanced components that do not map directly to native components. From this raw material, you must build an interface with which end-users on many different platforms will feel comfortable. This task is not always easy.

End-users on different platforms are accustomed to different ways of interacting with their computers. The metaphors are different. The components are different. The interaction between the components is different. Although the AWT and Swing libraries make it fairly easy to create a user interface that runs on multiple platforms, they do not necessarily make it easy to devise an interface that keeps end-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. If so, you can attempt to find a way to work around this problem.

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 (if possible) test your Java programs on all plat-

forms on which you are claiming that the program runs. Java programs are not platform independent to a great enough extent that you only need to 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 on which you claim your program runs. 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

Java's architecture enables 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 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 program's portability:

1. Choose a set of host computers and devices that you will claim your program runs on (your "target hosts").
2. Choose an edition and 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 run times").
4. Write your program so that it accesses the host computer only through the standard run-time libraries of the Java API. (Do not invoke native methods or use vendor-specific libraries that invoke native methods.)
5. Write your program so that it does not 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 run times and all of your target hosts.

If you follow the seven steps outlined here, 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 good chance that you program will run in many other places, as well.

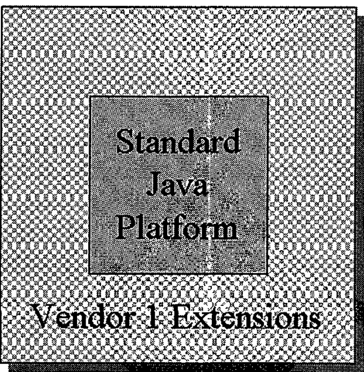
If you wish, you can have your program certified as "100% Pure Java." There are several reasons that you may wish to do this task if you are writing a program that you want to be platform independent. For example, if your program is certified 100% Pure, you can brand your program with the "100% Pure 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 on the platform independence of your program. In this case, you have the option of just running "100% Pure" verification tools that you can download for free. These tools will report problems with your program's "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 end-users' expectations are fulfilled on multiple platforms. The "100% Pure" testing process does not attempt to measure end-user fulfillment; rather, it only checks to make certain your program depends only on the standard APIs. You could write a Java program that passes the "100% Pure" tests but still does not work well on all platforms from the end-user's perspective. 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.

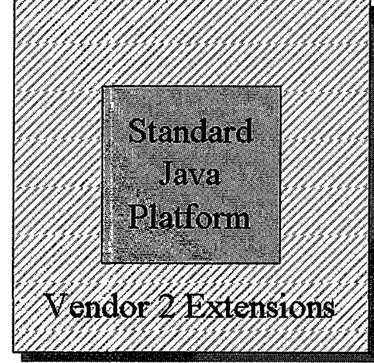
The Politics of Platform Independence

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

Figure 2-2
Java Platform implementations from different vendors



Java Platform from
Vendor 1



Java Platform from
Vendor 2

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 earlier, in the way vendors are allowed to implement threads, garbage collection, and user interface look and feel. If Sun's plans occur as scheduled, 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.

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, the program 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 enables 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.

Java's 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 probably run that same operating system. You are "locked into" 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. Instead, if much of your software is written to the Java API and is 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. Continuing this status quo is in Microsoft's strategic interest, so Microsoft is encouraging developers to use Java as a language to write programs that run only on Microsoft platforms. Weakening Microsoft's lock on the operating system market is in just about every other operating system vendor's strategic interest, so the other players are encouraging developers to write Java programs that are platform independent. For example, Sun, Netscape, IBM, and many others banded together to promote Sun's "100% Pure Java initiative," through which they hoped to educate and persuade developers to go the platform-independence route.

Microsoft's approach to Java is to make Windows the best platform on which to develop and run Java programs. They want developers to use Microsoft's tools and libraries, whether the developer chooses platform independence or not. Still, in the "spin" Microsoft gives to Java in promotional material to developers, Microsoft strongly favors the platform-specific Windows path. Microsoft extols 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 Microsoft's 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 Microsoft's way, you get to brand your product with a Windows 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 should 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.

Platform Independence and Network-Mobile Objects

As mentioned previously in this chapter, the original design target for Java technology was embedded devices. This target was chosen in part because given that the desktop was controlled by Microsoft and Intel, embedded devices represented the most open market. But also, embedded devices were targeted because they were destined to play a role in a coming hardware revolution—the proliferation of diskless, embedded devices connected to high-bandwidth (often, wireless) networks.

Three years after Java was first released by Sun, Sun announced the emergence of Jini. Jini is an attempt at defining an architecture for the "computer" represented by the emerging environment of embedded and consumer devices connected to a ubiquitous network. The Jini architecture relies heavily on network-mobile objects. In a world of Jini-enabled devices, objects fly across the network between Java Platform implementations in embedded and consumer devices, desktop computers, and servers. The Java Platform implementations that will host these network-mobile objects will reside in a great variety of devices and computer hardware, which will be manufactured by many different vendors. This architecture significantly raises the bar for platform independence.

For Jini to work in the real world, objects written by one device vendor will have to execute properly in Java Platform execution environments provided by other device vendors. Testing your network-mobile code on all platforms it will eventually run on, as recommended by the Seven Steps to Platform Independence presented earlier in this chapter, will be basically impossible. Because so many vendors will be producing so many different kinds of devices, with new devices appearing at an ever-increasing rate, it will be generally impossible to predict all the places where

network-mobile code embedded in any particular device will execute. Thus, other approaches to testing will have to be developed, such as compatibility test suites for network-mobile code. In addition, for Jini to work in the real world, the homogeneity of execution environments must be realized to the greatest extent possible. Lastly, programmers will likely need to consider the possibility of differences in execution environments when they write network-mobile code, and they will need to program defensively.

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/resources`.

CHAPTER 3

Security

Aside from platform independence, which we discussed in the previous chapter, the other major technical challenge that a network-oriented software technology must deal with is security. Because networks enable computers to share data and distribute processing, they can potentially serve as a way to break into a computer system—enabling someone to steal information, alter or 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, Java's architecture comes with an extensive, built-in security model that has evolved with each major release of the Java platform. This chapter gives an overview of the security model built into Java's core architecture and traces its evolution.

Why Security?

Java's 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 that is hooked to them. This concern becomes especially strong in an environment in which software is downloaded across the network and is executed locally, as is done, for example, with Java applets and Jini service objects. 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. Similarly, the class files for a Jini service object are downloaded from a codebase specified by the service provider when it registers its service with the Jini lookup service. Because Jini enables spontaneous networking in which users entering a new environment can look up and access locally available services, users of Jini services will more than likely encounter service objects from untrusted sources. Without any security, these automatic code download schemes would be a convenient way to distribute malicious code. Thus, Java's security mechanisms help make Java suitable for networks, because they establish a needed trust in the safety of executing network-mobile code.

Java's security model is focused on protecting end-users from hostile programs (and bugs in otherwise benevolent programs) that are downloaded across a network from untrusted sources. To accomplish this goal, Java provides a customizable "sandbox" in which untrusted Java programs can be placed. The sandbox restricts the activities of the untrusted program. The program can do anything within the boundaries of its sandbox but cannot take any action outside those boundaries. For example, the original sandbox for untrusted Java applets in Version 1.0 prohibited many activities, including the following:

- Reading or writing to the local disk
- Making a network connection to any hosts except the host from which the applet came
- Creating a new process
- Loading a new dynamic library

By making it impossible for downloaded code to perform certain actions, Java's security model protects the end-user from the threats of hostile and buggy code.

Because the sandbox security model imposes strict controls on what untrusted code can and cannot do, users are able to run untrusted code with relative security. Unfortunately for the programmers and users of 1.0 systems, however, the original sandbox was so restrictive that well-meaning (but untrusted) code was often unable to do useful work. In Version 1.1, the original sandbox model was augmented with a trust model based on code signing and authentication. The signing and authentication capability enables the receiving system to verify that a set of class files (in a JAR file) has been digitally signed (in effect, blessed as trustworthy) by some entity and that the class files have not been altered since they were signed. This process enables end-users and system administrators to ease the restrictions of the sandbox for code that has been digitally signed by trusted parties.

Although the security APIs released with Version 1.1 include support for authentication, they do not offer much help in establishing anything more than an all-or-nothing trust policy (in other words, either code is completely trusted or completely untrusted). Java's next major release, Version 1.2, provided APIs to assist with establishing fine-grained security policies based on authentication of digitally signed code. The remainder of this chapter will trace the evolution of Java's security model from the basic sandbox of Version 1.0, through the code signing and authentication of Version 1.1, to the fine-grained access control of Version 1.2.

The Basic Sandbox

In the world of personal computers, you traditionally had to trust software before running it. You achieved security by being careful only to use software from trusted sources and by regularly scanning for viruses. Once software gained access to your system, it had full reign. If the software was malicious, it could do a great deal of damage—because there were no restrictions placed on it by the run-time 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 do not fully trust. Instead of approaching security by requiring you to prevent any code that you do not trust from ever making its way into your computer, the sandbox model enables you to welcome code from any source. As code from an untrusted source runs, however, the

sandbox restricts the code from taking any actions that could possibly harm your system. You do not need to figure out what code you can and cannot trust. You do not need to scan for viruses. The sandbox itself prevents any viruses or other malicious, buggy code that you might invite into your computer from doing any damage to your system.

If you have a properly skeptical mind, you will need to be convinced that a sandbox has no leaks before you trust it to protect your system. To make sure that the sandbox has no leaks, Java's security model involves every aspect of its architecture. If there were areas in Java's architecture where security was not considered, a malicious programmer (known as a cracker) could likely exploit those areas to circumvent the sandbox. To understand the sandbox, therefore, you must look at several different parts of Java's architecture and understand how they work together.

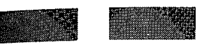
The fundamental components responsible for Java's sandbox are as follows:

- 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 Java's sandbox security model is that two of these components—the class loader and security manager—are customizable. By customizing these components, you can create a customized security policy for a Java application. Unfortunately, this capability for customization does not come for free, because the flexibility of the architecture creates some risks of its own. Class loaders and security managers are complicated enough that the mere act of customization can potentially produce errors that open security holes.

In each major release of the Java API, changes were made to make the task of creating a custom security policy less error prone. The most significant change occurred in Version 1.2, which introduced a new and more elaborate architecture for access control. In Versions 1.0 and 1.1, access control, which involves both the specification of a security policy and the enforcement of that policy at run time, is the responsibility of an object called the security manager. To establish a custom policy in Versions 1.0 and 1.1, you have to write your own custom security manager. In Version 1.2, you can take advantage of a security manager supplied with the Java 2 platform. This ready-made security manager enables you to specify a security policy in an ASCII *policy file* separate from the program. At run time, the ready-made security manager enlists the help of a class called the access controller to enforce the security policy specified in the policy file. The access control infrastructure introduced in Version 1.2 provides

a flexible and easily customized default implementation of the security manager that should suffice for the majority of your security needs. For backwards compatibility and to enable parties with special security needs to override the default functionality provided by the ready-made security manager, Version 1.2 applications can still install their own security manager. Using the ready made security manager (and the extensive access control infrastructure that comes with it) is optional.



The Class Loader Architecture

In Java's sandbox, the class loader architecture is the first line of defense. After all, the class loader brings code into the Java virtual machine—code that could be hostile or buggy. The class loader architecture contributes to Java's sandbox in three ways:

1. Preventing malicious code from interfering with benevolent code
2. Guarding the borders of the trusted class libraries
3. Placing code into categories (called *protection domains*) that will determine which actions the code can take

The class loader architecture prevents malicious code from interfering with benevolent code by providing separate name spaces for classes loaded by different class loaders. A *name space* is a set of unique names —one name for each loaded class—that the Java virtual machine maintains for each class loader. 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 the 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, cannot even detect each other's presence unless you explicitly provide a mechanism that enables 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 should not know or could interfere with the proper execution of your program.

Figure 3-1 shows the name spaces associated with two class loaders, both of which have loaded a type named `Volcano`. Each name in a name space is associated with the type data in the method area that defines the type with that name. Figure 3-1 shows arrows from the names in the name spaces to the types in the method area that define the type. The class loader on the left, which is shown in dark gray, has loaded the two dark-gray types named `Climber` and `Volcano`. The class loader on the right, which is shown in light gray, has loaded the two light-gray types named `BakingSoda` and `Volcano`. Because of the nature of name spaces, when the `Climber` class mentions the `Volcano` class, it refers to the dark-gray `Volcano`—the `Volcano` loaded in the same name space. The class has no way of knowing that the other `Volcano`, which is sitting in the same virtual machine, even exists. For details about how the class loader architecture achieves its separation of name spaces, see Chapter 8, "The Linking Model."

The class loader architecture guards the borders of the trusted class libraries by making it possible for trusted packages to be loaded with different class loaders than untrusted packages. Although you can grant special access privileges between types belonging to the same package by

Figure 3-1 Class loaders and name spaces

giving members protected or package access, this special access is granted to members of the same package at runtime—only if they were loaded by the same class loader.

Often, a user-defined class loader relies on other class loaders—at the least, upon the class loaders created at virtual machine startup—to help it fulfill some of the class-load requests that come its way. Prior to Version 1.2, class loaders had to explicitly ask for the help of other class loaders. A class loader could ask another user-defined class loader to load a class by invoking `loadClass()` on a reference to that user-defined class loader. Or, a class loader could ask the bootstrap class loader to attempt to load a class by invoking `findSystemClass()`, a static method defined in class `ClassLoader`. In Version 1.2, the process by which one class loader asks another class loader to try to load a type was formalized into a parent-delegation model. Starting with Version 1.2, each class loader except the bootstrap class loader has a "parent" class loader. Before a particular class loader attempts to load a type in its custom way, by default it "delegates" the job to its parent—asking its parent to try to load the type. The parent, in turn, asks its parent to try to load the type. The delegation process continues all the way to the bootstrap class loader, which is (in general) the last class loader in the delegation chain. If a class loader's parent class loader can load a type, the class loader returns that type. Otherwise, the class loader attempts to load the type itself.

In most Java virtual machine implementations prior to Version 1.2, the built-in class loader (which was then called the primordial class loader) was responsible for loading locally available class files. Such class files usually included the class files that made up the Java application being executed, plus any libraries needed by the application (including the class files of the Java API). Although the manner in which the class files for requested types were located was implementation specific, many implementations searched directories and JAR files in an order specified by a class path.

In Version 1.2, the job of loading locally available class files was parceled out to multiple class loaders. The built-in class loader, previously called the primordial class loader, was renamed the "bootstrap" class loader to indicate that it was now responsible for loading *only* the class files of the core Java API. The name bootstrap class loader comes from the idea that the class files of the core Java API are the class files required to "bootstrap" the Java virtual machine.

The responsibility for loading other class files, such as the class files for the application being executed, class files for installed or downloaded standard extensions, class files for libraries discovered in the class path, and so on, was issued in Version 1.2 to user-defined class loaders. When

a Version 1.2 Java virtual machine starts its execution, therefore, it creates at least one and probably more user-defined class loaders before the application even starts. All of these class loaders are connected in one chain of parent-child relationships. At the top of the chain is the bootstrap class loader, and at the bottom of the chain is what came (in Version 1.2) to be called the "system class loader." Prior to Version 1.2, the name "system class loader" was sometimes used to refer to the built-in class loader, which was also called the primordial class loader. In Version 1.2, the name system class loader was more formally defined to mean the default delegation parent for new user-defined class loaders created by a Java application. This default delegation parent will usually be the user-defined class loader that loaded the initial class of the application, but it might be any user-defined class loader decided upon by the designers of the Java platform implementation.

For example, imagine that you write a Java application that installs a class loader whose particular manner of loading class files is by downloading them across a network. Imagine that you run this application on a virtual machine that instantiates two user-defined class loaders on startup: an "installed extensions" class loader and a "class path" class loader. These class loaders are connected in a parent-child relationship chain, along with the bootstrap class loader (as shown in Figure 3-2). The class path's class loader's parent is the installed extensions class loader whose parent is the bootstrap class loader. As shown in Figure 3-2, the class path class loader is designated as the system class loader—the default delegation parent for new user-defined class loaders instantiated by the application. Assume that when your application instantiates its network class loader, it specifies the system class loader as its parent.

Imagine that during the course of running the Java application, a request is made of your class loader to load a class named `Volcano`. Your class loader would first ask its parent, the class path class loader, to find and load the class. The class path class loader, in turn, would make the same request of its parent, the installed extensions class loader. This class loader would also first delegate the request to its parent, the bootstrap class loader. Assuming that class `Volcano` is not a part of the Java API, part of an installed extension, or on the class path, all of these class loaders would return without supplying a loaded class named `Volcano`. When the class path class loader indicates that neither it nor any of its parents can 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 that your class loader could download class `Volcano`, that `Volcano` class could then play a role in the application's future course of execution.

Figure 3-2
A parent-child class loader delegation chain

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 `java.util.HashMap` from the Java API. Because it is the first time that the reference was used by the running program, the virtual machine asks your class loader (the one that loaded `Volcano`) to load `java.util.HashMap`. As before, your class loader first passes the request to its parent class loader, and the request becomes delegated all the way to the bootstrap class loader. In this case, however, the bootstrap class loader can return a `java.util.Hashmap` class back to your class loader. Because the bootstrap class loader can find the class, the installed extensions class loader does not attempt to look for the type in the installed extensions. The class path class loader does not attempt to look for the type on the class path. Also, your class loader does not attempt to download the type from the network. All of these class loaders merely return the `java.util.HashMap` class returned by the bootstrap class loader. From that point forward, the virtual machine uses that `java.util.HashMap` class whenever class `Volcano` references a class named `java.util.HashMap`.

Given this background information about how class loaders work, you are now ready to look at how class loaders can be used to protect trusted libraries. The class loader architecture guards the borders of the trusted class libraries by preventing untrusted classes from pretending to be trusted. If a malicious class could successfully trick the Java virtual machine into believing that 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 run time.

Given the parent-delegation model, the bootstrap class loader can attempt to load types before the standard extensions class loader, which can attempt to load types before the class path class loader, which can attempt to load types before your network class loader. Thus, given the manner in which the parent-child delegation chain is built, the most trusted library—the core Java API—is checked first for each type. Next, the standard extensions are checked. Then, local class files that are sitting on the class path are checked. Therefore, if some mobile code loaded by your network class loader wants to download a type across the network with the same name as an item in the Java API, such as `java.lang.Integer`, its action will fail. If a class file for `java.lang.Integer` exists in the Java API, the bootstrap class loader will load it. The network class loader will not attempt to download and define a class named `java.lang.Integer`. Rather, it will simply use the type returned by its parent—the one loaded by the bootstrap class loader. In this way, the class loader architecture prevents untrusted code from replacing trusted classes with their own versions.

Consider a different situation, however. What if the mobile code, rather than trying to replace a trusted type, wants to insert a brand-new type into a trusted package? Imagine what would happen if your network class loader from the previous example was requested to load a class named `java.lang.Virus`. As before, this request would first be delegated all the way up the parent-child chain to the bootstrap class loader. Although the bootstrap class loader is responsible for loading the class files of the core Java API, which includes a package named `java.lang`, it is unable to find a member of the `java.lang` package with the name `Virus`. Assuming that this class was also not found among the installed extensions or on the local class path, your class loader would proceed to attempt to download the type across the network.

Assume that your class loader is successful in the download attempt and defines the type named `java.lang.Virus`. Java permits classes in the same

package to grant each other special access privileges that are not granted to classes outside the package. Therefore, because your class loader loaded a class (`java.lang.Virus`) that (by its name) brazenly declares itself to be part of the Java API, you might expect that it could gain special access to the trusted classes of `java.lang` and could possibly use that special access for devious purposes. The class loader mechanism thwarts this code from gaining special access to the trusted types in the `java.lang` package, because the Java virtual machine only grants that special package access between types loaded into the same package by the same class loader. Because the trusted class files of the Java API's `java.lang` package were loaded by the bootstrap class loader, and the malicious `java.lang.Virus` class was loaded by your network class loader, these types do not belong to the same *runtime package*. The term runtime package, which first appeared in the second edition of the Java Virtual Machine Specification, refers to a set of types that belong to the same package and were all loaded by the same class loader. Before enabling access to package-visible members (members declared with protected or package access) between two types, the virtual machine makes sure not only that the two types belong to the same package, but that they belong to the same runtime package (that they were loaded by the same class loader). Thus, because `java.lang.Virus` and the members of `java.lang` from the core Java API do not belong to the same runtime package, `java.lang.Virus` cannot access the package-visible members and types of the Java API's `java.lang` package.

This concept of a runtime package is one motivation for using different class loaders to load different kinds of classes. The bootstrap class loader loads the class files of the core Java API. These class files are the most trusted. An installed extensions class loader loads class files from any installed extensions. Installed extensions are quite trusted, but they do not need to be trusted to the extent that they can gain access to package-visible members of the Java API by simply inserting new types into those packages. Because installed extensions are loaded with a different class loader than the core API, they cannot gain this access. Likewise, code found on the class path by the class path class loader cannot gain access to package-visible members of the installed extensions or to the Java API.

Another way that class loaders can be used to protect the borders of trusted class libraries is by simply prohibiting the loading of certain forbidden types. For example, you might have installed some packages that contain classes that you want your application to load through your network class loader's parent—the class path class loader—but not through your own network class loader. Assume that you have created a package named `absolutepower` and have installed it somewhere on the local

class path where it is accessible by the class path class loader. Also assume that you do not 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 first thing the loader does is make sure that the requested class does not declare itself a member of the absolutepower package. If such a class is requested, your class loader—rather than passing the class name to its parent class loader—would throw a security exception.

The only way that a class loader can know whether or not a class is from a forbidden package, such as absolutepower, is by the class's name. Thus, a class loader must have a list of the names of forbidden packages. Because the name of class absolutepower.FancyClassLoader indicates that it is part of the absolutepower package—and the absolutepower package is on the list of forbidden packages—your class loader should absolutely throw a security exception.

Besides shielding classes in different name spaces and protecting the borders of trusted class libraries, class loaders play one other security role: they must place each loaded class into a protection domain, which defines what permissions the code will be given as it runs. More information about this vitally important security job of class loaders will be given later in this chapter.

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 and that they are consistent. 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 cannot determine how a particular class file was created. Because a class file is simply a sequence of bytes, a virtual machine cannot know whether a particular class file was generated by a well-meaning Java compiler or by shady crackers who were 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 class files to make sure that the types they define 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 invoked—cause the virtual machine to crash. Thus, for the sake of robustness, the virtual machine should verify the integrity of the bytecodes it imports.

The class file verifier of the Java virtual machine does most checking before bytecodes are executed. Rather than checking every time it encounters a jump instruction as it executes bytecodes, for example, it 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 that 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.

The class file verifier operates in four distinct passes. During pass one, which takes place as a class is loaded, the class file verifier checks the internal structure of the class file to make sure that it is safe to parse. During passes two and three, which take place during linking, the class file verifier makes sure that the type data obeys the semantics of the Java programming language, including verifying the integrity of any bytecodes it contains. During pass four, which takes place as symbolic references are resolved in the process of dynamic linking, the class file verifier confirms the existence of symbolically referenced classes, fields, and methods.

Pass One: Structural Checks on the Class File

During pass one, the class file verifier makes certain that the sequence of bytes it will attempt to import as a type conform to the basic structure of a Java class file. The verifier performs many checks during this pass. For example, every class file must start with the same four bytes (the magic number): `0xCAFEBABE`. The purpose of the magic number is to make it easy for the class file parser to reject files that were either damaged or that were never intended to be class files in the first place. Thus, the first thing a class file verifier probably checks is that the imported file does indeed begin with `0xCAFEBABE`. The verifier also makes sure that the major and minor version numbers declared in the class file are within the range supported by that implementation of the Java virtual machine.

Also during pass one, the class file verifier checks to make sure that 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, the verifier can make sure that the imported file has a length consistent with its internal contents.

The point of pass one is to ensure that the sequence of bytes that supposedly define a new type adhere sufficiently to the Java class file format to enable them to be parsed into implementation-specific, internal data structures in the method area. Passes two, three, and four take place not on the binary data in the class file format, but on the implementation-specific data structures in the method area.

Pass Two: Semantic Checks on the Type Data

Pass two of the class file verifier performs checking that can be done without looking at the bytecodes and without examining (or loading) any other types. During this pass, the verifier looks at individual components to make sure that 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 that the verifier performs on individual components is to make sure that 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. Also during pass two, the verifier makes sure that final classes are not subclassed and that final methods are not overridden. In addition, it checks that constant pool entries are valid and that all indexes into the constant pool refer to the correct type of constant pool entry. Thus, the class file verifier checks some of the Java language rules at run time that should have been enforced at compile time. Because the verifier has no way of knowing whether the class file was generated by a benevolent, bug-free compiler, it checks each class file to make sure that the rules are followed.

Pass Three: Bytecode Verification

Once the class file verifier has successfully completed the pass two checks, it turns its attention to the bytecodes. During this pass, which is com-

monly 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 bytecodes and frames.

The bytecode streams that represent Java methods are a series of one-byte instructions called *opcodes*, each of which can be followed by one or more *operands*. The operands supply extra data needed by the Java virtual machine to execute the opcode instruction. 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 receives its own frame, which we can define as a section of memory where it stores local variables and intermediate results of computation (among other items). The part of the frame in which a method stores intermediate results is called the method's *operand stack*. An opcode and its (optional) operands might refer to the data stored on the operand stack or in the local variables of the method's frame. Thus, the virtual machine can 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, from checking to make sure that no matter which 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. The bytecode verifier also checks to make sure that no local variable is accessed before it is known to contain a proper value. The bytecode 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 are on the operand stack. These are just a few of the many checks performed by the bytecode verifier, which can (through all of its checking) verify that a stream of bytecodes is safe for the Java virtual machine to execute.

The bytecode verifier does not attempt to detect all safe programs. If it did, it would encounter the Halting Problem. The Halting Problem, a well-known theorem in computer science, states that you cannot write a program that can determine whether any program fed to it as input will halt when it is executed. Whether or not a program will halt is called an "undecidable" property of the program, because you cannot write a program that can tell you 100 percent of the time whether or not any given program has this property. The undecideability of the Halting Problem

extends to many properties of computer programs, including whether or not a set of Java bytecodes would be safe for a Java virtual machine to execute.

The way the bytecode verifier circumvents the Halting Problem is by not attempting to pass all safe programs. Although you cannot write a program that can determine whether or not any given program will halt, you can write a program that recognizes some programs that will halt. For example, if the first instruction of a program is halted, that program will halt. If a program has no loops in it, it will halt, and so on. Similarly, although you cannot write a verifier that will pass all bytecode streams that are safe for the virtual machine to execute, you can write a verifier that will pass some of them. That task is what Java's bytecode verifier does. The verifier checks to make sure that a certain set of rules are followed by each set of bytecodes fed to it. If a set of bytecodes obeys all of the rules, then the verifier knows that the bytecodes are safe for the virtual machine to execute. If not, the bytecodes might or might not be safe for the virtual machine to execute. Thus, the verifier avoids the Halting Problem by recognizing some, but not all, safe bytecode streams. Given the nature of the constraints checked by the bytecode verifier, any program that can be written in the Java programming language can be compiled to bytecodes that will pass the verifier. Some programs that could not possibly be expressed in the Java programming language will pass the verifier. And some programs (also not expressible in Java source code) that would otherwise be safe for the virtual machine to execute will not pass the verifier.

Passes one, two, and three of the class file verifier make sure that the imported class file is properly formed, is 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 conditions are not true, it throws an error, and the program never uses the class file.

Pass Four: Verification of Symbolic References

Pass four of the class file verifier takes place when the symbolic references contained in a class file are resolved in the process of dynamic linking. During pass four, the Java virtual machine follows the references from the class file being verified to the referenced class files to make sure that the references are correct. Because pass four must examine other classes that

are external to the class file being checked, pass four might require that new classes are loaded. Most Java virtual machine implementations will likely delay loading classes until the program actually uses them. 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 cannot find a certain referenced class, it does not throw a `NoClassDefFoundError` error until (and unless) the referenced class is used for the first time by the running program. Thus, if a Java virtual machine performs early linking, pass four could happen shortly after pass three. In Java virtual machines that resolve each symbolic reference the first time they are used, however, pass four will happen much later than pass three as bytecodes are executed.

Pass four 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, while symbolic references to the fields of other classes give the class name, field name, and field descriptor; and 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. Finding the class being referenced (and loading it if necessary)
2. Replacing 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 spending time resolving the symbolic reference again.

When the Java virtual machine resolves a symbolic reference, pass four of the class file verifier makes sure that the reference is valid. If the reference is not valid—for instance, if the class cannot be loaded or if the class exists but does not contain the referenced field or method—then 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, then the name and descriptor of the method in Lava are included as part of the binary data in the class file for Volcano. When Volcano's method first invokes Lava's method during the course of execution, the Java virtual machine makes sure that 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 now on. But if the symbolic reference from class Volcano does not match any method in class Lava, pass four verification fails, and the Java virtual machine throws a NoSuchMethodError.

Binary Compatibility

The reason why pass four of the class file verifier must look at classes that refer to one another to make sure that 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 doing so, they will detect any incompatibility at compile time. There might be times, however, when your compiler does not 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 likely cause a recompilation of affected classes within that same package—but not necessarily in any other package. Moreover, if you are using someone else's packages, especially if your program downloads class files from someone else's package across a network as it runs, then it might be impossible for you to check for compatibility at compile time. For this reason, pass four of the class file verifier must check for compatibility at run time.

As an example of incompatible changes, imagine that you compiled class Volcano (from the previous example) with a Java compiler. Because a method in Volcano invokes a method in another class called Lava, the Java compiler would look for a class file or a source file for class Lava to make sure that there was a method in Lava with the appropriate name, return type, and number and types of arguments. If the compiler could not find any Lava classes, or if it encountered a Lava class that did not contain the desired method, the compiler would then 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 was not already compatible with class Lava.

The converse, however, is not necessarily true. The Java compiler could conceivably generate a class file for Lava that is not compatible with Volcano. If the Lava class does not refer to Volcano, you could potentially change the name of the method that 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 you still used the old version of Volcano that was not recompiled since you made your change to Lava, then the Java virtual machine would (as a result of pass four class-file verification) throw a NoSuchMethodError 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 might arise when you update a library you have been using and your existing code is not 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 enable you to make many kinds of changes to a class that do not require recompilation of classes that depend upon the language. The changes you can 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 are 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 essence) deleted the old method and added a new one. If you had instead added the new method and then rewritten the old method so that it calls the new method, 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 has performed passes one through three of class-file verification, the bytecodes are ready

to be executed. Besides the verification of symbolic references (pass four of class-file verification), the Java virtual machine has several other built-in security mechanisms operating as bytecodes are executed. These mechanisms, most of which are elements of Java's type safety, are listed in Chapter 1 as features of the Java programming language that make Java programs robust. Not surprisingly, these features are also part of the Java virtual machine:

- Type-safe reference casting
- Structured memory access (no pointer arithmetic)
- Automatic garbage collection (cannot explicitly free allocated memory)
- Array bounds checking
- Checking references for `null`

By granting a Java program-only type safe, which provides structured ways to access memory, the Java virtual machine makes Java programs more robust, but it also makes their execution more secure. 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, for example, it is critical that the process does not crash. This level of robustness is also important in embedded systems, such as a cellular phone, which people do not usually expect to have to reboot. Another reason why unrestrained memory access would be a security risk is because a 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, the cracker could assign a pointer to that memory and manipulate the class loader's data. By enforcing structured access to memory, the Java virtual machine yields programs that are robust but that also frustrate 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 for structured memory access—is the unspecified manner in which the run-time 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 will not 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. What is worse (for the cracker) is that the cracker cannot determine anything about memory layout by reading the Java virtual machine specification. 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 run-time 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 machine's memory access restrictions, they would next be faced with the difficult task of finding something to subvert by searching the structure.

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 firm barrier against the malicious manipulation of memory.

There is a way, however, to penetrate the security barriers erected by the mechanisms that support type safety in a Java virtual machine. Although the bytecode instruction set does not give you an unsafe, unstructured way to access memory, there is a way you can avoid bytecodes: native methods. Basically, when you call a native method, Java's security sandbox becomes dust in the wind. First of all, the robustness guarantees do not hold for native methods. Although you cannot corrupt memory from a Java method, you can from a native method. Most importantly, however, native methods do not go through the Java API (they are used to circumvent the Java API), so the security manager is not checked before a native method attempts to do something that could be potentially damaging. (This process is often how the Java API itself gets anything done, of course. 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 does not apply anymore to that thread—as long as that thread continues to execute the native method. For this reason, the security manager includes a method that establishes whether or not a program can load dynamic libraries, which are necessary for invoking native methods. Untrusted

applets, for example, are not permitted to load a new dynamic library; therefore, they cannot install their own new native methods. They can, however, call methods in the Java API—methods which might be native, but 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 the method.

One final mechanism 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 might result in the death of the offending thread but should not 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 situation is usually a major inconvenience to a running Java program but will not necessarily result in termination of the entire program. If the program has other threads doing useful tasks, those threads might have the capacity to carry on without their recently departed colleague. Throwing an exception, on the other hand, could result in the death of the thread but is often 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

The first three prongs of Java's security model—the class loader architecture, the class file verifier, and the safety features built into Java—all work together to achieve a common goal: protecting the internal integrity of a Java virtual machine instance and the application it is running from malicious or buggy code that it might load. By contrast, the fourth prong of the security model—the security manager—is geared towards protecting assets that are external to the virtual machine from malicious or buggy code running within the virtual machine. The security manager is a single object that serves as the central point for access control—the controlling of access to external assets—within a running Java virtual machine.

The security manager defines the outer boundaries of the sandbox. Because it is customizable, the security manager enables a custom security policy to be established 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. To ask the security manager for permission, the methods of the Java API invoke check methods on the security manager object. These methods are called check methods because their names all begin with the substring *check*. For example, the security manager's `checkRead()` method determines whether or not a thread can read to a specified file. The `checkWrite()` method determines whether or not a thread can write to a specified file. The implementation of these methods is what defines the custom security policy of the application.

Because the Java API always checks with the security manager before it performs a potentially unsafe action, the Java API will not perform any action that is forbidden under the security policy established by the security manager. If the security manager forbids an action, the Java API will not perform that action.

When a Java application starts, it has no security manager. The application, however, can install one by passing a reference to an instance of `java.lang.SecurityManager` or one of its subclasses to `setSecurityManager()`, a static method of class `java.lang.System`. If an application does not install a security manager, there are no restrictions placed on any activities requested of the Java API; rather, the Java API will do whatever it is asked. (For this reason, 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 in Version 1.0 or Version 1.1, that security manager will be in charge for the entire remainder of the lifetime of that application. This security manager cannot 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 Version 1.2, however, the currently installed security manager can be replaced by code that has permission to replace it by invoking `System.setSecurityManager()` with a reference to a different security manager object.

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, there are two steps involved in the procedure that a Java API method generally follows when it is about to perform a potentially unsafe activity. First, the Java API code checks to determine whether a security manager has been installed. If not, it skips

step two and continues with the potentially unsafe action. Otherwise, in 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, then the check method will simply return. In this case, the Java API method carries on and performs the potentially unsafe action.

As mentioned earlier in this chapter, the security manager is responsible for two items: specifying a security policy, and enforcing that policy. The security policy, which outlines the kind of code that will be permitted to take a certain kind(s) of action(s), is defined by the code of the security manager's check methods. The policy is enforced by the behavior of the check methods when they are invoked.

Prior to Version 1.2, `java.lang.SecurityManager` was an abstract class. To establish a custom security policy in Version 1.0 or Version 1.1, you had to write your own security manager by subclassing `SecurityManager` and implementing its check methods. Your application would instantiate and install the security manager, and from that point forward (for the remainder of the life of the application), the security manager would enforce the security policy that you defined in the code of its check methods.

Although the customizability of the security manager was one of the greatest strengths of Java's security model, it was also a potential weak point. Writing a security manager is a complicated and error-prone task. Any mistakes made when implementing the check methods of a security manager could potentially translate into security holes at run time. To help make it easier and less error prone for developers and end-users to establish fine-grained security policies based on signed code, the `java.lang.SecurityManager` class in Version 1.2 is a concrete class that provides a default implementation of the security manager. (In the remainder of this book, this default implementation of the security manager provided with Version 1.2 will be called the concrete `SecurityManager`.) Your application can instantiate and install this security manager explicitly or can install it automatically. In Sun's Java 2 SDK Version 1.2, for example, you can specify that the concrete `SecurityManager` is installed by using the `-Djava.security.manager` option on the command line.

The concrete `SecurityManager` class enables you to define your custom policy not in Java code, but in an ASCII file called a *policy file*. In the policy file, you grant *permissions* to *code sources*. Permissions are defined in terms of classes that are subclasses of `java.security.Permission`. For example, `java.io.FilePermission` represents permission to read,

write, execute, or delete a file. Code sources are composed of a codebase URL from which the code was loaded and a set of signers that vouched for the code. When the security manager is created, it parses the policy file and creates `CodeSource` and `Permission` objects. These objects are encapsulated in a single `Policy` object that expresses the policy at run time. Only one `Policy` object can be installed at any time.

Class loaders place types into protection domains, which encapsulate all the permissions granted to the code source represented by the loaded type. Each type loaded into a Version 1.2 virtual machine belongs to one (and only one) protection domain. The virtual machine remembers the protection domain and uses it when deciding whether or not the code can take potentially unsafe actions.

When the check methods of the concrete `SecurityManager` are invoked, most of them pass the request on to a class called the `AccessController`. The `AccessController`, using the information contained in the protection domain objects of the classes whose methods are on the call stack, performs stack inspection to determine whether the action should be permitted.

The security manager has undergone quite a bit of change in Version 1.2. In Versions 1.0 and 1.1, each check method indicates what is being checked in its method name. To check whether or not it is acceptable to read a certain file, the Java API invokes the `checkRead()` method on the security manager and passes the path name of the file to read as a parameter. For example, before attempting to read a file called `/tmp/finances.dat`, the security manager invokes `checkRead("/tmp/finances.dat")` on the security manager.

The security manager declares 28 of these check methods—which, in the remainder of this chapter, will be referred to as legacy check methods. Although new methods were added to the security manager in Version 1.2 that would otherwise render these legacy check methods obsolete, to maintain backwards compatibility, the Java API continues to call the legacy check methods just as it did in prior releases.

The 28 legacy check methods are listed here, along with the potentially unsafe action that triggers their invocation by the code of the Java API:

- `checkConnect(String host, int port)`—Opens a socket connection to the specified host and port number
- `checkConnect(String host, int port, Object context)`—Opens a socket connection to the specified host and port number under the passed security context
- `checkAccept(String host, int port)`—Accepts a socket connection from the specified host and port number

- `checkCreateClassLoader()`—Creates a new class loader
- `checkAccess(Thread t)`—Modifies a thread (changes its priority, stops it, etc.)
- `checkAccess(ThreadGroup t)`—Modifies a thread group (adds a new thread, sets daemons, etc.)
- `checkExit()`—Causes the application to exit
- `checkLink()`—Loads a dynamic library that contains native methods
- `checkRead(FileDescriptor fd)`—Reads from the specified file
- `checkRead(String file)`—Reads from the specified file
- `checkRead(String file, Object context)`—Reads from the specified file under the passed security context
- `checkWrite(FileDescriptor fd)`—Writes to the specified file
- `checkWrite(String file)`—Writes to the specified file
- `checkDelete(String file)`—Deletes the specified file
- `checkListen(int port)`—Waits for a connection on the specified local port number
- `checkMulticast(InedAddress maddr)`—Joins, leaves, sends, or receives IP multicast
- `checkMulticast(InedAddress maddr, byte ttl)`—Joins, leaves, sends, or receives IP multicast
- `checkPropertiesAccess()`—Accesses or modifies system properties in general
- `checkPropertiesAccess(String key)`—Accesses or modifies the specified system property
- `checkTopLevelWindow(Object Window)`—Brings up the specified window without any warning
- `checkPrintJobAccess()`—Initiates a print job request
- `checkSystemClipboardAccess()`—Accesses the system's clipboard
- `checkAWTEventQueueAccess()`—Accesses the AWT event queue
- `checkPackageAccess(String pkg)`—Accesses types from the specified package (used by class loaders)
- `checkPackageDefinition(String pkg)`—Adds a new class to the specified package (used by class loaders)
- `checkSetFactory()`—Sets the socket factory that `ServerSocket` or `Socket` uses or sets the URL stream handler that `URL` uses
- `checkMemberAccess()`—Accesses class information via the reflection API

In Version 1.2, a set of permission classes was defined whose instances represent the actions that code can take. A new pair of check methods were added in Version 1.2 to class `java.lang.SecurityManager`, both named `checkPermission()`:

- `checkPermission(Permission perm)`—Takes an action that requires the specified permission
- `checkPermission(Permission perm, Object context)`—Takes an action that requires the specified permission under the passed security context

The `checkPermission()` methods accept a reference to a `Permission` object, which indicates the action that is being requested. Thus, this method provides an alternative way to ask the security manager whether it is acceptable to perform a potentially unsafe action. For example, to determine whether it is acceptable to read file `/tmp/finances.dat`, the Java API in Version 1.2 could take either of two approaches. The Java API could take the old-fashioned approach and invoke the legacy method `checkRead()`, passing the `String` `"/tmp/finances.dat"` as a parameter, or it could take the fresh, new approach of creating a `java.io.FilePermission` object and passing `Strings` `"/tmp/finances.dat"` and `"read"` to the `FilePermission` constructor. The Java API could then pass this `Permission` object to the security manager's `checkPermission()` method.

Both the old-fashioned approach of invoking a legacy check method and the fresh new approach of creating a permission object and invoking `checkPermission()` should yield the same result. To maintain backwards compatibility with security managers that were written for Versions 1.0 or 1.1, however, the Version 1.2 Java API continues to take the old-fashioned approach. The Version 1.2 Java API continues to call the 28 legacy check methods. Nevertheless, in the concrete `SecurityManager` class, the legacy methods are (for the most part) implemented in terms of the new `checkPermission()` method. So, by invoking the legacy method on the concrete `SecurityManager`, the Java API is indirectly invoking the `checkPermission()` method anyway. For example, the `checkRead()` method implementation in the concrete `SecurityManager` simply instantiates a new `FilePermission` object, passing the path name `String` passed to it to the `FilePermission`'s constructor, along with the `String` `"read"`. The `checkRead()` method then invokes `checkPermission()`, passing a reference to the `FilePermission` object.

At times, the Java API might also invoke `checkPermission()` directly. For new concepts of potentially unsafe actions introduced in Version 1.2 and later, no legacy check methods exist. Thus, in some situations, the Java API might create a new `Permission` object for which no

relevant check methods exist and pass that `Permission` object directly to the security manager's `checkPermission()` method.

In the concrete `SecurityManager` class, the `checkPermission()` method also delegates the job of deciding whether or not to permit another method to perform the action. The concrete `SecurityManager`'s `checkPermission()` method simply invokes the static `checkPermission()` method of class `java.security.AccessController`, passing along the permission object. The `AccessController` class, therefore, is the actual entity responsible for enforcing the security policy when you use the concrete `SecurityManager`.

All of these changes in Version 1.2 are backwards compatible with Versions 1.1 and 1.0. In other words, if you created a security manager for Version 1.1, it should still work as expected in Version 1.2. You can still create a custom security manager in Version 1.2 as well, which enables anyone with special security needs that are not adequately addressed by the concrete `SecurityManager` implementation to create a different kind of security infrastructure. Most people's security needs, however, will more than likely be met by taking advantage of the flexibility and extensibility built into the concrete `SecurityManager`.

Code Signing and Authentication

A critical piece of Java's security model is the support for authentication introduced in Java 1.1 in the `java.security` package and its subpackages. The authentication capabilities expand your ability to establish multiple security policies by enabling you to implement a sandbox that varies depending on who vouched for the code. Authentication enables you to verify that a set of class files was blessed as trustworthy by some party —and that the class files were not altered en route to your virtual machine. Thus, to the extent that you trust the party who vouched for the code, you can ease the restrictions placed on the code by the sandbox. You can establish different security restrictions for code that is signed by different parties.

To vouch for, or sign, a piece of code, you must first generate a public/private key pair. You should keep the private key private, but you can make the public key public. At the least, you must somehow get the public key to anyone who wants to establish a security policy based on your signature. (As illustrated later in this section, distributing public keys is not necessarily as easy as it might seem.) Once you have a public/private

key pair, you must place the class files and any other files you want to sign into a JAR file. You then use a tool, such as `jarsigner` from the Version 1.2 SDK, to sign the entire JAR file. The signer tool will first perform a one-way hash calculation on the contents of the JAR file to generate a hash. The tool will then sign the hash with your private key and add the signed hash to the JAR file. The signed hash represents your digital signature of the contents of the JAR file. When you distribute the JAR file that contains the signed hash, anyone with your public key can verify two things about the JAR file: that you indeed signed the JAR file, and that the contents of the JAR file were not in any way altered since you attached your signature.

The first step in the digital signing process is the one-way hash calculation, which takes a big number as input and generates a small number (called the hash). In the case of a JAR file, the big-number input to the calculation is the stream of bytes that make up the contents of the JAR file. The one-way hash calculation is called one-way because given just the hash (the small number), it is impossible to calculate the input (the big number). In other words, the hash value does not contain enough information about the input to enable the input to be regenerated from the hash. The calculation goes just one way, from big to small and from input to hash.

The hash, which is also called a message digest, serves as a kind of fingerprint for the input. Although different inputs can produce the same hash, the hash is considered unique enough in practice to represent the input from which it was generated. Much like a fingerprint can be used to identify the individual who made the fingerprint, a hash can be used to identify the input that caused the one-way hash algorithm to produce the hash. The hash is used during the authentication process to verify that the input is identical to the input that produced the original hash. In other words, the hash verifies that the input was not changed en route to its destination.

Given that it is impossible to reconstruct the input given just the hash, a hash is only useful if the input is also available. Thus, you normally transmit both the input and the hash together. By themselves, the combination of an input and its hash is not secure, however, because even an extremely unimaginative cracker could simply replace both the input and the hash. To prevent this scenario from occurring, you encrypt the hash with your private key before sending the hash. The reason why you encrypt the hash rather than simply encrypting the entire JAR file is because private-key encryption is a time-consuming process. In general,

you will find it much faster to calculate a one-way hash from the JAR file contents and encrypt the hash with a private key than to encrypt the entire JAR file with the private key. A cracker will only have the capability to replace both an input and an encrypted hash if the cracker has your private key, which you are supposed to keep secret. Thus, the combination of input and encrypted hash is more frustrating to a potential cracker than the mere combination of input and hash, because in theory, the cracker does not have your private key.

Anything encrypted with your private key can be decrypted with your public key. With public/private key pairs, it is difficult to generate the private key if you only have the public key. If you can keep your private key out of the hands of crackers, therefore, their best option is to try to replace the input with a different input that yields the same hash value. If the cracker wishes to replace one class file in your JAR file with a different class file that performs some devious act, for example, the odds are extremely high that the revised JAR file (the one that contains the devious class file) will produce a different hash. But the cracker could add random data to the JAR file until the one-way hash calculation on the altered JAR file produces the same hash value as the original. If the cracker can produce such an alternative input—one that both helps the cracker achieve his or her nefarious goals and generates the same hash as your original input—then the cracker would not need your private key. Because the cracker's input generates the same hash value as your original input, and you have already signed that hash value with your private key, then the cracker can simply place your signed hash in a JAR file with his or her input. What prevents a cracker from taking this approach? Unfortunately for the cracker, such an approach would more than likely take too much time to be feasible.

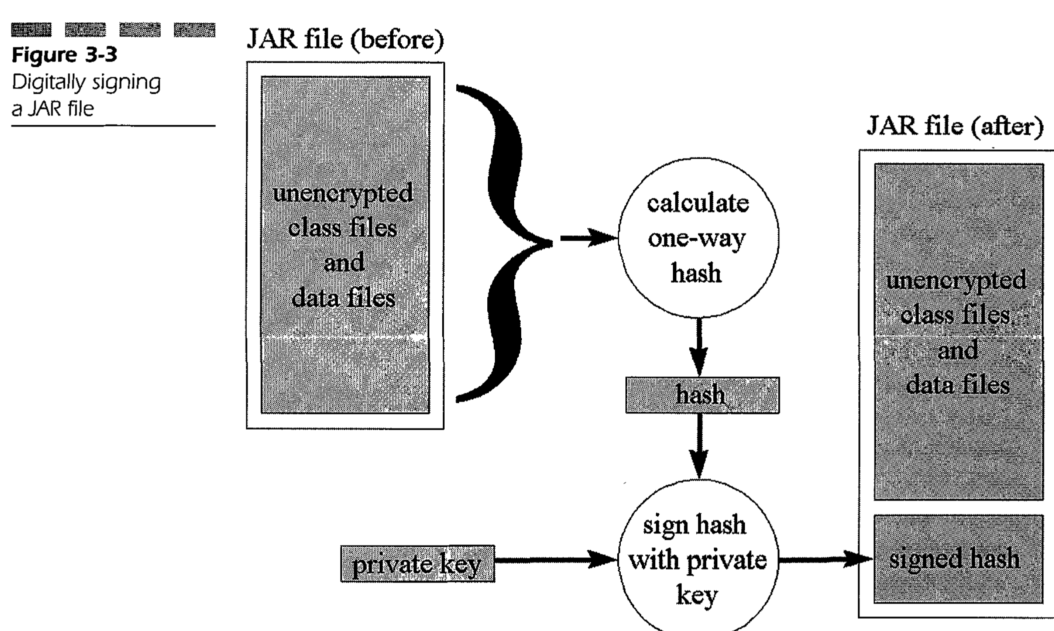
Because one-way hash algorithms generate a small number (the message digest or hash) from a big number (the input), different inputs can produce the same hash. One-way hash algorithms tend to spread out the inputs that produce the same hash in a sufficiently random manner, so the likelihood of getting the same hash value depends primarily on the size of the hash. For example, if you use a hash value that is eight bits wide, your one-way hash algorithm will only have 256 unique hash values from which to choose. If you have a JAR file that produces the hash value 100 and you start calculating the eight-bit hash with this algorithm on other JAR files, you should not be surprised if every 256 times or so you receive the hash value 100. The more bits contained in the hash, of course, the less often the algorithm will produce the same hash. In practice, 64- and 128-bit hash values are common and are considered large

enough to render the process of finding a different input that produces the same hash computationally unfeasible. The main barrier preventing a cracker from replacing your benevolent input with a malicious input that serves the cracker's evil purposes *and* produces the same hash, therefore, is the time and resources that the cracker would have to devote to searching for that malicious input.

The last step in the digital signing process, after you have generated the hash value and encrypted it with your private key, is to add the encrypted hash value to the same JAR file that contains the files from which you generated the hash value originally. A signed JAR file, therefore, contains the input—the class and data files you wanted to vouch for—plus the hash value (generated from the input) encrypted with your private key. The encrypted hash represents your digital signature of the class and data files contained in the same JAR file. The process of signing a JAR file is shown graphically in Figure 3-3.

To authenticate a JAR file that you have purportedly signed, the recipient must decrypt the signed hash with your public key. The result should be equal to the original hash that you calculated on the contents of the

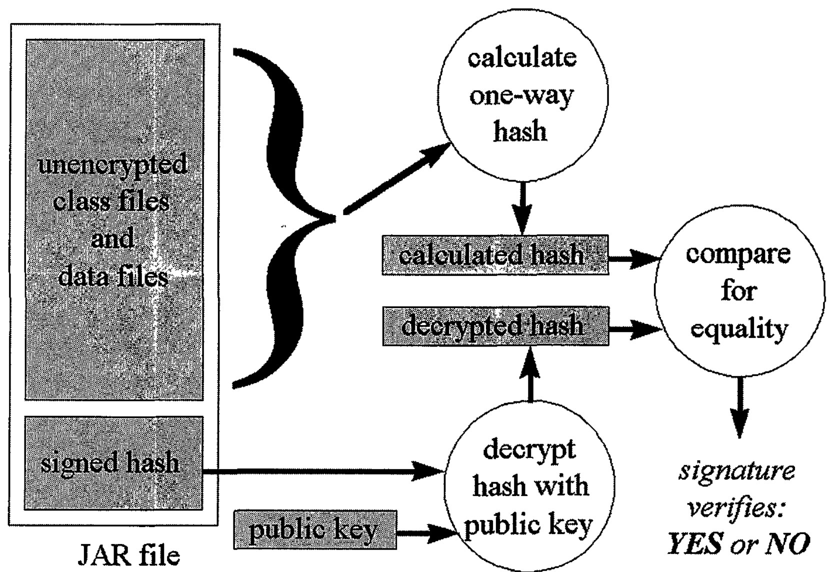
Figure 3-3
Digitally signing a JAR file



JAR file. To verify that the JAR file contents were not changed since you signed them, the recipient simply applies the one-way hash algorithm on the contents of the JAR file, just as you did during the signing process. (You never encrypted the contents of the JAR file, so anyone can see them. You only added a digital signature to the JAR file.) If the hash value generated by the algorithm matches the decrypted hash value, the recipient concludes that you did indeed vouch for this JAR file and that the contents of the JAR file did not change since you added your signature. The code contained in the JAR file can be placed inside a relaxed sandbox that represents the trust that the recipient places in your signature. The process of verifying a digitally signed JAR file is shown in Figure 3-4.

Although the authentication technology first introduced in Java Version 1.1 is firmly founded in reliable mathematics, the math does not solve every problem. In fact, Java's authentication technology raises several questions. For example, the authentication technology says nothing about who you should trust and to what extent you should trust them. To what extent do you trust some small company that you do not recognize? To what extent do you trust a big company whose name is a household word? To what extent do you trust a different department in your own company? What are the chances that any particular company (or department) has

Figure 3-4 Authenticating a digitally signed JAR file



a rogue employee who managed to slip a time bomb into a JAR file that the company signed? No cryptographic algorithm can answer these questions for you.

Another security issue stems from the assumption that is inherent in the authentication technology that private keys will be kept under lock and key. If private keys are not kept private, the entire authentication scheme is reduced to strenuous mathematical activity that is not only ineffective but dangerous, because it can give a false sense of security. You are responsible for keeping your own private keys private. You can only hope that any entity on whose signature authority you grant code access to your system has kept their private keys private. For any party, establishing a key-management scheme that prevents private keys from being leaked (remember those rogue employees?) can be a challenging task.

Another question raised by the technology involves the distribution of public keys. Although it might seem surprising at first, the assumption inherent in the authentication technology that public keys will be made public creates some security issues of its own. For example, imagine that you want to relax your sandbox for code vouched for by a guy named Evan. To do so, you need Evan's public key. But how exactly do you obtain his public key? If you know Evan personally, you can invite him over for coffee and ask him to bring his public key so he can give it to you in person. But what if you do not know Evan personally? You might think that you could simply visit Evan's Web site and grab his public key from his Web page. Alternatively, perhaps you could phone Evan and ask him to send you his public key in an e-mail. Evan should have no problem sending you—a stranger—his public key, because public keys are designed to be public. Evan does not need to worry about who gets his public key. He could hire a biplane to write his public key on the sky over Silicon Valley and still feel confident he was operating within the rules delineated by Java's authentication technology. So, what is the problem? The problem is that although Evan does not need to worry about your identity when he sends you his public key, you need to worry about his. Evan will be happy to send you his public key, but how do you know that the public key you receive is really the one that Evan sent?

The difficulty of public key distribution is that no matter what means of communication you use, the message—the public key—could potentially be tampered with in transit. When you visit Evan's Web page, it is possible that the Web page is intercepted and changed en route to your browser, perhaps by Dastardly Doug, a cracker of international repute. When you think you are copying Evan's public key from his Web page, you could actually be copying Dastardly Doug's. Doug could also have intercepted Evan's

e-mail and replaced Evan's beneficent public key with his own dastardly public key. Doug could even have donned one of his many clever disguises and piloted the biplane high above Silicon Valley, inscribing his public key among the clouds in place of Evan's. If Doug can successfully replace Evan's public key with his own, Doug can pretend to be Evan and take advantage of the trust you place in Evan's signature to break into your system.

But wait a minute. Isn't the difficulty of public key distribution just another authentication problem—the kind of problem the authentication technology itself is designed to address? The answer to this question is yes. By turning authentication back on itself, Evan can make it far more difficult for Dastardly Doug to replace Evan's public key with his own.

To address the difficulties of public key distribution, several *Certificate Authorities* (CAs) have been established for the purpose of vouching for public keys. Evan, for example, could go to a certificate authority and present his credentials (birth certificate, driver's license, passport, and so on) and his public key. Once convinced that Evan is who he says he is, the CA would sign Evan's public key with the certificate authority's private key. The resulting sequence of numbers is called a *certificate*. Instead of distributing his public key, then, Evan would distribute his certificate.

You could grab Evan's certificate from his Web page, from an e-mail, or via any other unsecured communications medium. When you get the certificate, you decrypt it with the CA's public key and receive Evan's public key. The certificate scheme makes it much less likely that Doug will be able to swap his public key for Evan's. To do so, Doug would need the CA's private key.

Although certificates improve the public key distribution situation immensely, some issues still remain. First of all, how exactly do you obtain the CA's public key? You need this public key to authenticate the public keys of anyone else. Well, if you know any employees in the CA personally, you could invite them over for coffee and ask them to bring their public key to give to you in person. But what if you do not know any employees in the CA? Then, there is the nagging question of why you should trust the CA. A CA can pretend to be anyone. Isn't a CA just as susceptible as the next company to the vagaries of rogue employees?

Despite all of these concerns, the code-signing capabilities introduced in Java Version 1.1 generally offer you *enough* security to enable you to relax your sandbox when necessary. Although the authentication technology does not eliminate all risks associated with relaxing the sandbox, it can help minimize the risks. Security is a tradeoff between cost and risk: the lower the security risk, the higher the cost of security. You must

weigh the costs associated with any computer or network security strategy against the costs of the theft or destruction of the information or computing resources being protected. The nature of your computer or network security strategy should be shaped by the value of the assets you are trying to protect. Java's authentication technology is a useful tool that, in concert with Java's sandbox, can help you manage the costs and risks of running network-mobile code on your systems.

A Code-Signing Example

For an example of code signing with the `jarsigner` tool of the Java 2 SDK Version 1.2, consider the following types: `Doer`, `Friend`, and `Stranger`. The first type, `Doer`, defines an interface that the other two types, classes `Friend` and `Stranger`, implement:

```
// On CD-ROM in file
// security/ex2/com/artima/security/doer/Doer.java
package com.artima.security.doer;

public interface Doer {

    void doYourThing();
}
```

`Doer` declares just one method: `doYourThing()`. Class `Friend` and class `Stranger` implement this method in the exact same way. In fact, besides their names, the two classes are identical:

```
// On CD-ROM in file
// security/ex2/com/artima/security/friend/Friend.java
package com.artima.security.friend;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Friend implements Doer {

    private Doer next;
    private boolean direct;

    public Friend(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }
```

```
    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}

// On CD-ROM in file
// security/ex2/com/artima/security/stranger/Stranger.java
package com.artima.security.stranger;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Stranger implements Doer {

    private Doer next;
    private boolean direct;

    public Stranger(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}
```

These types—Doer, Friend, and Stranger—are designed to illustrate the stack inspection mechanism of the access controller. The motivation behind their design will be made clear later in this chapter, when we give you several examples of stack inspection. At this point, however, the class files generated by compiling Friend and Stranger must be signed to prepare them for the upcoming stack inspection examples. The class files generated from Friend.java will be signed by a party referred to fondly as "friend." The class files generated from Stranger.java will be signed by a party referred to somewhat suspiciously as "stranger." The class file generated by Doer will not be signed.

To prepare the class files for signing, they must first be placed into JAR files. Because the class files for Friend and Stranger need to be signed by two different parties, they will be collected into two different JAR files. The two class files generated by compiling Friend.java—Friend.class and Friend$1.class—will be placed into a JAR file called friend.jar. Similarly, the two class files generated by compiling Stranger.java—Stranger.class and Stranger$1.class—will be placed into a JAR file called stranger.jar. (Note that although all of the files in these examples are in the security/ex2 directory of the CD-ROM, to repeat any of the commands that generate files, you will have to copy the entire security/ex2 directory hierarchy to a writeable media, such as a hard disk. You probably knew that already, however.)

Friend.java's class files are dropped by the javac compiler in the security/ex2/com/artima/security/friend directory. Because class Friend is declared in the com.artima.security.friend package, Friend.java's class files must be placed in the JAR file in the com/artima/security/friend directory. The following command, executed in the security/ex2 directory, will place Friend.class and Friend$1.class into a newly created JAR file called friend.jar, which is placed in the current directory, security/ex2:

```
jar cvf friend.jar com/artima/security/friend/*.class
```

Once the previous command completes, the class files for Friend.java must be removed so they will not be found by the Java virtual machine when it runs the access control examples:

```
rm com/artima/security/friend/Friend.class
rm com/artima/security/friend/Friend$1.class
```

Filling a JAR file with Stranger.java's class files, which are dropped by javac in the security/ex2/com/artima/security/stranger

directory, requires a similar process. From the security/ex2 directory, the following command must be executed:

```
jar cvf stranger.jar com/artima/security/stranger/*.class
rm com/artima/security/stranger/Stranger.class
rm com/artima/security/stranger/Stranger$1.class
```

To sign a JAR file with the jarsigner tool from the Java 2 SDK Version 1.2, a public/private key pair for the signer must already exist in a *keystore file*, which is a file for storing named, password-protected keys. The keytool program of the Java 2 SDK Version 1.2 can be used to generate a new key pair, to associate the key pair with a name or *alias*, and to protect it with a password. The alias, which is unique within each keystore file, is used to identify a particular key pair in a particular keystore file. The password for a key pair is required to access or change the information contained in the keystore file for that key pair.

The access control examples expect a keystore file named ivjmkeys in the security/ex2 directory containing key pairs for the aliases "friend" and "stranger." The following command, executed from the security/ex2 directory, will generate the key pair for the alias friend with the password friend4life. In the process, the command will create the keystore file named ijvmkeys:

```
keytool -genkey -alias friend -keypass friend4life
-validity 10000 -keystore ijvmkeys
```

The -validity 10000 command-line argument of the previous keytool command indicates that the key pair should be valid for 10,000 days. Over the course of 27 years, then, the key pair will likely outlive the product life cycle of this edition of this book. When the command runs, it will prompt you for a keystore password, which is a general password required for making any kind of access to or change in the keystore file. The keystore password given to ijvmkeys is "ijvm2ed".

The key pair for stranger can be generated with a similar command:

```
keytool -genkey -alias stranger -keypass stranger4life
-validity 10000 -keystore ijvmkeys
```

Now that the keystore file ijvmkeys contains key pairs for friend and stranger—and the JAR files friend.jar and stranger.jar contain the appropriate class files—the JAR files can finally be signed. The following jarsigner command, executed from the examples/ex2 directory, will sign the class files contained in friend.jar using friend's private key:

```
jarsigner -keystore ijvmkeys -storepass ijvm2ed -keypass
friend4life friend.jar friend
```

A similar command will sign the class files contained in `stranger.jar` with `stranger`'s private key:

```
jarsigner -keystore ijvmkeys -storepass ijvm2ed -keypass
stranger4life stranger.jar stranger
```

Whew, that was a lot of work just to sign two JAR files. Keep in mind that in the real world, you would have to make sure that no one with bad intent got hold of your private keys and that you kept track of them. In other words, do not lose the keystore file, remember the passwords, etc. In addition, you will have to give your public keys to anyone who will use your signature to give your code access to their system.

Policy

As mentioned previously, one of the greatest advantages of Java's sandbox security model is that the sandbox can be customized. The code signing and authentication technology introduced in Java Version 1.1 enables your running application to differentiate code to which you attribute different degrees of trust. By customizing the sandbox, trusted code can be given more access to system resources than untrusted code. This feature prevents untrusted code from accessing the system but enables trusted code to access the system and do useful work. The real power of Java's security architecture, however, lies in the capability to grant varying degrees of trust to code that has different levels of *partial* access to the system.

Microsoft offers an authentication technology similar to Java's for ActiveX controls, but ActiveX controls do not run inside a sandbox. Thus, with ActiveX, a chunk of mobile code is either completely trusted or completely untrusted. If untrusted, the ActiveX control is denied the opportunity to run. If trusted, the ActiveX control is enabled to run and is given full access to the system. While this functionality is a big improvement from no authentication at all, if some malicious or buggy code becomes authenticated, then the dangerous code has full access to the system. One of the strengths of Java's security architecture is that code can be given access only to the resources it needs. If some malicious or buggy code becomes authenticated, it has fewer opportunities to do damage. For example, instead of being able to delete all files on a local hard disk, the

malicious or buggy code might only have the capability to delete the files in a particular directory set aside just for the malicious code.

One major goal of the Version 1.2 security infrastructure is to make it easier and less error prone to establish fine-grained access control policies based on signed code. To assign different system-access privileges to different units of code, Java's access control mechanism must have the capability to ascertain which privileges should be given to each individual piece of code. To facilitate this process, each piece of code (each class file) loaded into a Version 1.2 or higher Java virtual machine is associated with a *code source*. The code source basically indicates where the code came from and who, if anyone, has vouched for the code by signing the code. In the Version 1.2 security model, permissions (system-access privileges) are assigned to code sources. Thus, if a piece of code requests access to a particular system resource, the Java virtual machine will grant the code access to that resource only if such access is a privilege associated with that code's code source.

In the Version 1.2 security infrastructure, an access control policy for an entire Java application is represented by a single instance of a subclass of the abstract class `java.security.Policy`. Each application has just one `Policy` object in effect at any given time. Code that has permission can replace the current `Policy` object with a new one by invoking `Policy.setPolicy()` and by passing a reference to the new `Policy` object. Class loaders consult the `Policy` object to help them decide which privileges to grant code as they import the code into the virtual machine.

A security policy is a mapping from a set of properties that characterize running code to the permissions granted to the code. In the Version 1.2 security infrastructure, the properties that characterize running code are collectively called the code source. A code source is represented by a `java.security.CodeSource` object, which contains a `java.net.URL` to represent the codebase and an array of zero or more certificate objects to represent the signers. Certificate objects are instances of subclasses of the abstract class `java.security.cert.Certificate`. A `Certificate` is an abstraction that represents a binding of a principal to a public key and another principal (the certificate authority mentioned previously) that vouches for that binding. The `CodeSource` object contains an array of `Certificate` objects, because the same code can be signed (vouched for) by more than one party. The signatures are usually obtained from a JAR file.

All of the tools and access control infrastructure that accompanies the concrete `SecurityManager` in Version 1.2 work only with certificates. None work with bare public keys. If you do not have a certificate author-

ity handy, you can sign your own public key with your private key and generate a self-signed certificate. The `keytool` program from the Java 2 SDK Version 1.2 always generates a self-signed certificate when it generates keys. In the code-signing example given earlier in this chapter, for instance, the `keytool` created not only public/private key pairs but also created self-signed certificates for the aliases `friend` and `stranger`.

A permission is represented by an instance of a subclass of the abstract class `java.security.Permission`. A permission object has three properties: a type, a name, and an optional action. A permission's type is indicated by the name of the permission class. Some examples of permission types are: `java.io.FilePermission`, `java.net.SocketPermission`, and `java.awt.AWTPermission`. A permission's name is encapsulated inside the `Permission` object. For example, the name of a `FilePermission` might be: `"/my/finances.dat"`; the name of a `SocketPermission` might be `"applets.artima.com:2000"`; and the name of an `AWTPermission` might be `"showWindowWithoutBannerWarning"`. The third property of a `Permission` object is its action. Not all permissions have an action. An example of an action for a `FilePermission` is "`read,write`", and an example for a `SocketPermission` is "`accept,connect`". A `FilePermission` with the name `/my/finances.dat` and action `read,write` represents permission to read and write to the file `/my/finances.dat`. Both name and action are represented by `Strings`.

The Java API has a large hierarchy of permissions that represent potentially dangerous actions that code might wish to take. You can also create your own permission classes to represent custom permissions that you can use for your own purposes. For example, you could create permission classes that represent permission to access particular records of your proprietary database. Defining custom permission classes is one way that you can extend the Version 1.2 security mechanism to reflect your own needs. If you create your own `Permission` classes, you can use them like any of the built-in `Permission` classes from the Java API.

In the `Policy` object, each `CodeSource` is associated with one or more `Permission` objects. The `Permission` objects with which a `CodeSource` is associated are encapsulated in an instance of a subclass of `java.security.PermissionCollection`. Class loaders can invoke `Policy.getPolicy()` to get a reference to the policy object currently in effect. They can then invoke `getPermissions()` on the `Policy` object, passing a `CodeSource` to get a `PermissionCollection` of `Permission` objects for the passed `CodeSource`. A class loader can then use the `PermissionCollection` retrieved from the `Policy` object to help it decide which permissions the code it is about to import will be granted.

Policy File

`java.security.Policy` is an abstract class. One of the implementation details of concrete `Policy` subclasses is how an instance of the subclass learns what the policy should be. Subclasses can take various approaches, such as deserializing a previously serialized policy object, extracting the policy from a database, or reading the policy from a file. The concrete policy subclass supplied by Sun with the Version 1.2 Java platform takes the latter approach: it enables you to express your security policy in a context-free grammar in an ASCII *policy file*.

A policy file consists of a series of *grant clauses*, each of which grants a code source a set of permissions. As mentioned previously, a code source consists of a codebase, which is a URL from which code was loaded, and a set of signers. In the policy file, signers are designated with the alias with which the signer's public key is stored in a keystore file. The keystore can be explicitly specified in the policy file in a `keystore` statement.

As an example of a policy file, consider the `policyfile.txt` file from the `security/ex2` directory of the CD-ROM:

```
keystore "ijvmkeys";

grant signedBy "friend" {
     permission java.io.FilePermission "question.txt",
     "read";
     permission java.io.FilePermission "answer.txt", "read";
};

grant signedBy "stranger" {
     permission java.io.FilePermission "question.txt",
     "read";
};

grant codeBase "file:${com.artima.ijvm.cdrom.home}/
security/ex2/*" {
     permission java.io.FilePermission "question.txt",
     "read";
     permission java.io.FilePermission "answer.txt", "read";
};
```

The first statement in the `policyfile.txt` file is a `keystore` statement:

```
keystore "ijvmkeys";
```

This keystore statement indicates that the key aliases mentioned in the rest of the policy file refer to certificates stored in a file called `"ijvmkeys"`. Because this filename includes no path, the file must be

located in the current directory—the directory in which the Java application using this policy file is started.

The second statement in the policy file is a grant statement:

```
grant signedBy "friend" {
    permission java.io.FilePermission "question.txt",
    "read";
    permission java.io.FilePermission "answer.txt", "read";
};
```

This statement grants two permissions to any code signed by the entity with the alias "friend." The granted permissions are as follows: permission to read a file named question.txt, and permission to read a file named answer.txt. Because these filenames appear with no path, both files must be in the current directory—the directory in which the application is started. Because no codebase is mentioned in the grant clause, code signed by friend can come from any codebase. All code signed by friend, regardless of codebase, will be awarded permission to read question.txt and answer.txt.

The third statement in policyfile.txt is another grant statement, similar in form to the first:

```
grant signedBy "stranger" {
    permission java.io.FilePermission "question.txt",
    "read";
};
```

This statement grants one permission to any code signed by the entity with the alias "stranger": permission to read a file named question.txt. This file must be sitting in the current directory—the directory in which the application is started. Because no codebase is mentioned in the grant clause, code signed by stranger can come from any codebase and will still be awarded permission to read question.txt. Note that although stranger can read the question contained in question.txt, stranger is not permitted to peek at the answer contained in answer.txt. This situation contrasts with the privileges awarded to friend, which can read both the question and the answer.

The fourth and final statement in this policy file is yet another grant statement:

```
grant codeBase "file:${com.artima.ijvm.cdrom.home}/
security/ex2/*" {
    permission java.io.FilePermission "question.txt",
    "read";
    permission java.io.FilePermission "answer.txt", "read";
};
```

This final grant statement grants two permissions to any code that was loaded from a particular directory: permission to read a file named question.txt, and permission to read a file named answer.txt. Both files must be in the current directory—the directory in which the application is started. Note that this grant statement does not mention any signers. The code can be signed by anyone or by no one. As long as it is loaded from the indicated directory, the code will be granted the listed permissions.

The codebase URL in this grant statement takes the form of a file: URL that includes a property, ${com.artima.ijvm.cdrom.home}. If you run the AccessControl example programs described later in this chapter, you will have to set the com.artima.ijvm.cdrom.home property to the path of the CD-ROM that comes with this book (or to whichever directory you have moved the security subdirectory from the CD-ROM). The Policy object that is instantiated based on the contents of policyfile.txt will take the com.artima.ijvm.cdrom.home property into account when it constructs the URL for the CodeSource for this grant clause.

Protection Domains

As class loaders load types into the Java virtual machine, they assign each type into a *protection domain*. A protection domain defines all the permissions that are granted to a particular code source. (A protection domain corresponds to one or more grant clauses in a policy file.) Each type loaded into a Java virtual machine belongs to one and only one protection domain.

The class loader knows the codebase and the signers of any class or interface it loads and uses that information to create a CodeSource object. The class loader passes the CodeSource object to the getPermissions() method of the currently used Policy object to obtain an instance of a subclass of the abstract class java.security.PermissionCollection. The PermissionCollection holds references to all Permission objects granted to the given code source by the current policy. With both the CodeSource that it created and the PermissionCollection it got from the Policy object, it can instantiate a new ProtectionDomain object. It places the code into a protection domain by passing the appropriate ProtectionDomain object to the defineClass() method, an instance method of class ClassLoader that user-defined class loaders call to import type data into the Java virtual machine. This assigning classes into protection domains is a critical job which, as mentioned earlier in this chap-

ter, is one of three ways the class loader architecture supports Java's sandbox security model.

Although the Policy object represents a global mapping from code sources to permissions, in the end the class loader is the responsible party that decides which permissions the code will receive when it runs. A class loader could, for example, completely ignore the current policy and just assign permissions randomly. Or, a class loader could add permissions to those returned by the policy object's getPermissions() method. For example, a class loader for loading applet code could add a permission to make a socket connection back to the host, from which the applet came to the permissions granted to the code by the policy (if any). As you can see, the class loader plays a crucial security role as it loads classes.

For a graphical depiction of protection domains, code sources, and permissions, consider Figure 3-5. In this figure, the method area and heap are shown after the code inside friend.jar is loaded under the policy defined by policyfile.txt. friend.jar is a JAR file in the security/ex2/jars directory of the CD-ROM, and policyfile.txt is an ASCII policy file in the security/ex2 directory. The friend.jar file contains two class files: Friend.class and Friend$1.class. As described in the code-signing example earlier in this chapter, both of these class files have been signed by friend. When these classes are defined by the class loader, they are placed into a protection domain whose CodeSource object indicates two things. First, the CodeSource indicates that the class files were loaded from a local jar file whose URL is file:///f|/security/ex2/jars/friend.jar. Second, the CodeSource indicates that the class files were signed by friend, an alias associated with a certificate in the local keystore. The ProtectionDomain object encapsulates a reference to the CodeSource object and to a java.security.Permissions object. java.security.Permissions, a concrete subclass of the abstract java.security.PermissionCollection class, represents a heterogeneous collection of permissions. The Permissions object holds references to two java.io.FilePermission objects. These two FilePermissions grant the privilege to read files named question.txt and answer.txt in the current directory.

When a class loader imported Friend and Friend$1 into the method area shown in Figure 3-5, the class loader passed a reference to the ProtectionDomain object to defineClass(), along with the bytes of the class files. The defineClass() method associated the type data in the method area for Friend and Friend$1 with the passed ProtectionDomain object. This association is shown graphically in Figure 3-5, which includes arrows that represent references to the

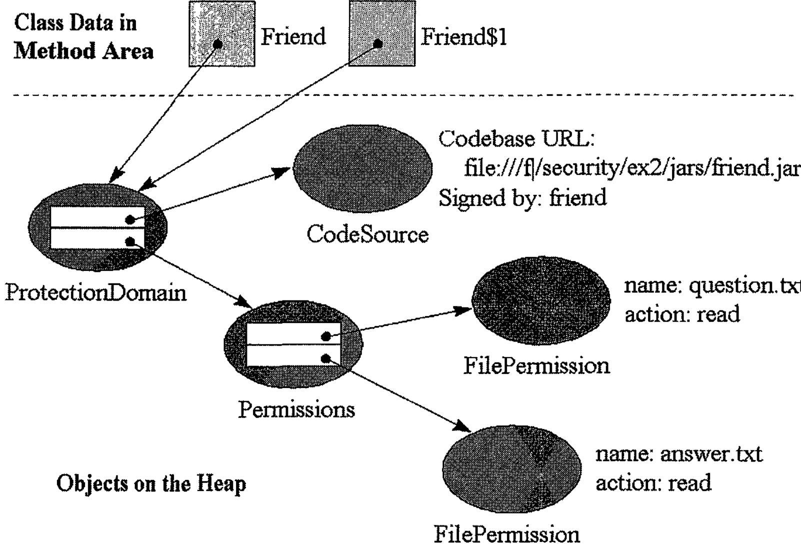


Figure 3-5 Protection domains, code sources, and permissions

`ProtectionDomain` object held as part of the type data in the method area for `Friend` and `Friend$1`.

The Access Controller

Class `java.security.AccessController` provides a default security policy enforcement mechanism that uses stack inspection to determine whether or not potentially unsafe actions should be permitted. The access controller cannot be instantiated, because it is not an object. Rather, the access controller is a bundle of static methods collected in a single class. The central method of the `AccessController` is its static `checkPermission()` method, which decides whether or not a particular action is permitted. This method returns `void` and takes a reference to a `Permission` object as its only parameter. Similar to the check methods of the security manager, if the `AccessController` decides that the operation should be permitted, its `checkPermission()` method simply returns silently. But if the `AccessController` decides that an operation should be forbidden, its `checkPermission()` method completes abruptly by throwing an `AccessControlException` or by throwing one of its subclasses.

As mentioned previously, the concrete `SecurityManager`'s implementation of the legacy check methods (such as `checkRead()` and `checkWrite()`)

simply instantiate an appropriate Permission object and invoke the concrete SecurityManager's checkPermission() method. The concrete SecurityManager's checkPermission() method simply invokes checkPermission() on the AccessController. Thus, if you install the concrete SecurityManager, the AccessController is the ultimate entity that decides whether or not potentially unsafe actions will be permitted.

The basic algorithm implemented by the AccessController's checkPermission() method makes certain that every frame on the call stack has permission to perform the potentially unsafe action. Each stack frame represents some method that has been invoked by the current thread. Each method is defined in some class, and each class belongs to some protection domain. Also, each protection domain contains a set of permissions, so each stack frame is indirectly associated with a set of permissions. For an action represented by the Permission object passed to the AccessController's checkPermission() method to be enabled, the basic algorithm of the AccessController requires that the permissions associated with each frame on the call stack must include or imply that the Permission is passed to checkPermission().

The AccessController's checkPermission() method inspects the stack from the top down. As soon as it encounters a frame that does not have permission, it throws an AccessControlException. By throwing the exception, the AccessController indicates that the action should not be permitted. On the other hand, if the checkPermission() method reaches the bottom of the stack without encountering any frames that do not have permission to perform the potentially unsafe action, checkPermission() returns silently. By returning rather than throwing an exception, the AccessController indicates that the action should be permitted.

The actual algorithm implemented by the AccessController's checkPermission() method is a bit more complex than the basic algorithm described here. By invoking any of several doPrivileged() methods of class AccessController, programs can (in effect) cause the AccessController to stop its frame-by-frame search before it reaches the bottom of the stack. We will describe the doPrivileged() method later in this chapter.

The implies( ) Method

To determine whether or not the action represented by the Permission object passed to the AccessController's checkPermission() method is included among (or implied by) the permissions associated with the

code on the call stack, the AccessController makes use of an important method called implies(). The implies() method is declared in class Permission, as well as in classes PermissionCollection and ProtectionDomain. implies() takes a Permission object as its only parameter and returns a Boolean true or false. The implies() method of class Permission determines whether the permission represented by one Permission object is naturally implied by the permission represented by a different Permission object. The implies() methods of PermissionCollection and ProtectionDomain determine whether the passed Permission is included among or implied by the collection of Permission objects encapsulated in the PermissionCollection or ProtectionDomain.

For example, a permission to read all of the files in the /tmp directory would naturally imply a permission to read /tmp/f—a specific file in the /tmp directory. The converse of this statement, however, is not true. If you asked a FilePermission object that represents the permission to read any file in the /tmp directory if it implies the permission to read file /tmp/f, the implies() method should return true. But if you ask a FilePermission object representing the permission to read /tmp/f if it implies the permission to read any file in the /tmp directory, the implies() method should return false.

The Example1 application from the security/ex1 directory of the CD-ROM demonstrates this meaning of implies():

```
import java.security.Permission;
import java.io.FilePermission;
import java.io.File;

// On CD-ROM in file security/ex1/Example1.java
class Example1 {

    public static void main(String[] args) {

        char sep = File.separatorChar;

        // Read permission for "/tmp/f"
        Permission file = new FilePermission(
            sep + "tmp" + sep + "f", "read");

        // Read permission for "/tmp/*", which
        // means all files in the /tmp directory
        // (but not any files in subdirectories
        // of /tmp)
        Permission star = new FilePermission(
            sep + "tmp" + sep + "*", "read");

        boolean starImpliesFile = star.implies(file);
        boolean fileImpliesStar = file.implies(star);
```

```
            // Prints "Star implies file = true"
            System.out.println("Star implies file = "
                + starImpliesFile);

            // Prints "File implies star = false"
            System.out.println("File implies star = "
                + fileImpliesStar);
        }
    }
```

The Example1 application creates two FilePermission objects: one that represents read permission for a particular directory, and another that represents read permission for a particular file in that same directory. The FilePermission object referenced from local variable star represents the permission to read any file in /tmp. The FilePermission object referenced from local variable file represents the permission to read file /tmp/f. When executed, this application prints the following lines:

```
Star implies file = true
File implies star = false
```

The implies() method is used by the AccessController to determine whether a thread has permission to take actions. If the checkPermission() method of the AccessController is invoked to determine whether that thread has permission to read file /tmp/f, for example, the AccessController can invoke the implies() method on the ProtectionDomain objects associated with each frame of that thread's call stack. To each implies() method, the AccessController can pass the FilePermission object representing permission to read file /tmp/f that was passed to its checkPermission() method. The implies() method of each ProtectionDomain object can invoke implies() on the PermissionCollection it encapsulates, passing along the same FilePermission. Each PermissionCollection can, in turn, invoke implies() on the Permission objects it contains—once again passing along a reference to the same FilePermission object. As soon as a PermissionCollection's implies() method encounters one Permission object whose implies() method returns true, the PermissionCollection's implies() method returns true. Only if none of the implies() methods of the Permission objects contained in a PermissionCollection return true does the PermissionCollection return false. The ProtectionDomain's implies() method simply returns what the PermissionCollection's implies() method returns. If the AccessController receives a true from the implies() method of a ProtectionDomain associated with a particular stack frame, the code represented by that stack frame has permission to perform the potentially unsafe action.

Stack Inspection Examples

The next few sections give several examples to illustrate the manner in which the AccessController performs stack inspection. In the upcoming examples, code signed by both friend and stranger will be trusted to some extent, but friend code will be trusted more than stranger code. In particular, code signed by both friend and stranger will be given permission to read a file named question.txt, which contains a question. Although code signed by friend will be given permission to read a file named answer.txt, which contains the answer to the question asked in question.txt, code signed by stranger will not. These permissions granted to friend and stranger are those outlined in the policyfile.txt file from the security/ex2 directory of the CD-ROM, which was described earlier in this chapter. Each of the upcoming examples will take their policy from policyfile.txt.

The stack inspection examples all make use of classes that implement the Doer interface:

```
// On CD-ROM in file
// security/ex2/com/artima/security/doer/Doer.java
package com.artima.security.doer;

public interface Doer {

    void doYourThing();
}
```

To be a Doer, a class must provide an implementation for one method: doYourThing(). Classes that implement Doer can do whatever they feel like in their doYourThing() method. For example, here is a class named TextFileDisplayer that implements Doer. This class displays the contents of a text file.

```
// On CD-ROM in file security/ex2/TextFileDisplayer.java

import com.artima.security.doer.Doer;
import java.io.FileReader;
import java.io.CharArrayWriter;
import java.io.IOException;

public class TextFileDisplayer implements Doer {

    private String fileName;

    public TextFileDisplayer(String fileName) {
        this.fileName = fileName;
    }
```

```
        public void doYourThing() {

            try {
                FileReader fr = new FileReader(fileName);

                try {
                    CharArrayWriter caw = new
                    CharArrayWriter();

                    int c;
                    while ((c = fr.read()) != -1) {
                        caw.write(c);
                    }

                    System.out.println(caw.toString());
                }
                catch (IOException e) {
                }
                finally {
                    try {
                        fr.close();
                    }
                    catch (IOException e) {
                    }
                }
            }
            catch (IOException e) {
            }
        }
    }
```

When you create a TextFileDisplayer object, you must pass a file path name to its constructor. The TextFileDisplayer constructor stores the passed path name in the filename instance variable. When you invoke doYourThing() on the TextFileDisplayer object, it will attempt to open and read the contents of the file and print them at the standard output.

Another example of a doYourThing() method comes from classes Friend and Stranger, which appeared earlier in this chapter in the code-signing example and are shown again here to refresh your memory:

```
// On CD-ROM in file
// security/ex2/com/artima/security/friend/Friend.java
package com.artima.security.friend;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Friend implements Doer {

    private Doer next;
```

```
    private boolean direct;

    public Friend(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}

// On CD-ROM in file
// security/ex2/com/artima/security/stranger/Stranger.java
package com.artima.security.stranger;
import com.artima.security.doer.Doer;
import java.security.AccessController;
import java.security.PrivilegedAction;

public class Stranger implements Doer {

    private Doer next;
    private boolean direct;

    public Stranger(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
```

```
                    }
                }
            );
        }
    }
}
```

Friend and Stranger have much in common. They have identical instance variables, constructors, and doYourThing() methods. They differ only in their package and simple names. When you create a new Friend or Stranger object, you must pass to the constructor a Boolean value and a reference to another object whose class implements the Doer interface. The constructor stores the passed Doer reference in the instance variable, next, and the Boolean value in the instance variable, direct. When doYourThing() is invoked on either a Friend or Stranger object, the method invokes doYourThing()—either directly or indirectly—on the Doer reference contained in next. If direct is true, Friend or Stranger's doYourThing() just invokes doYourThing() directly on next. Otherwise, Friend or Stranger's doYourThing() invokes doYourThing() on next indirectly, by way of a doPrivileged() call.

A Stack Inspection That Says "Yes"

As the first stack inspection example, consider the Example2a application:

```
// On CD-ROM in file security/ex2/Example2a.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;

// This succeeds because everyone has permission to
// read answer.txt
class Example2a {

    public static void main(String[] args) {

        TextFileDisplayer tfd = new TextFileDisplayer
        ("question.txt");

        Friend friend = new Friend(tfd, true);

        Stranger stranger = new Stranger(friend, true);

        stranger.doYourThing();
    }
}
```

The Example2a application creates three Doer objects: a TextFileDisplayer, a Stranger, and a Friend. The TextFileDisplayer constructor is passed the String, "question.txt". When its doYourThing() method is invoked, it will attempt to open a file named question.txt in the current directory for reading and will attempt to print its contents to the standard output. The Friend object's constructor is passed a reference to the TextFileDisplayer object (a Doer) and the Boolean value true. Because the passed Boolean value is true, when Friend's doYourThing() method is invoked, it will directly invoke doYourThing() on the TextFileDisplayer object. The Stranger object's constructor is passed a reference to the Friend object (also a Doer) and to the Boolean value true. Because the passed Boolean value is true, when Stranger's doYourThing() method is invoked, it will directly invoke doYourThing() on the Friend object. After creating these three Doer objects and hooking them together as described, Example2a's main() method invokes doYourThing() on the Stranger object. Now, the fun begins.

When the Example2a program invokes doYourThing() on the Stranger object referenced from the stranger variable, the Stranger object invokes doYourThing() on the Friend object, which invokes doYourThing() on the TextFileDisplayer object. TextFileDisplayer's doYourThing() method attempts to open and read a file called "question.txt" in the current directory (the directory in which the Example2a application was started) and print its contents to the standard output. When TextFileDisplayer's doYourThing() method creates a new FileReader object, the FileReader's constructor creates a new FileInputStream whose constructor checks to see whether or not a security manager has been installed. In this case, the concrete SecurityManager has been installed, so the FileInputStream's constructor invokes checkRead() on the concrete SecurityManager. The checkRead() method instantiates a new FilePermission object representing permission to read file question.txt and passes that object to the concrete SecurityManager's checkPermission() method, which passes the object on to the checkPermission() method of the AccessController. The AccessController's checkPermission() method performs the stack inspection to determine whether this thread should be permitted to open file question.txt for reading.

Figure 3-6 shows the call stack when the AccessController's checkPermission() method is invoked. In this figure, each frame of the call stack is represented by a horizontal row that is composed of several elements. The leftmost element in each stack frame row, which is labeled

class, is the fully qualified name of the class in which the method represented by that stack frame is defined. The next element to the right, which is labeled *method*, gives the name of the method. The next element, which is labeled *protection domain*, indicates the protection domain with which each frame is associated. Farthest to the right is an arrow that shows the progression of the `AccessController's` `checkPermission()` method as it checks to see whether each stack frame has permission to perform the requested action. Just to the left of the arrow is a number for each stack frame. Similar to all images of the stack shown in this book, the top of the stack appears at the bottom of the picture. Thus, in Figure 3-6, the top of the stack is the frame numbered 10.

The protection domain column of the stack diagram shown in Figure 3-6 shows each frame associated with one of four protection domains, called "FRIEND," "STRANGER," "CD-ROM," and "BOOTSTRAP." Three of these protection domains correspond to grant clauses in `policyfile.txt`. The FRIEND protection domain corresponds to the grant clause that gives permission to any code signed by `friend` to read `question.txt` and `answer.txt`. The STRANGER protection domain corresponds to the grant clause that gives permission to any code signed by `stranger` to read `question.txt`. The CD-ROM protection domain corresponds to the grant clause that gives permission to any code loaded from the "${com.artima.ijvm.cdrom.home}/security/ex2/" directory to read `question.txt` and `answer.txt`. The fourth and final protection domain, called BOOTSTRAP, does not correspond to any grant clause in `policyfile.txt`. Rather, the BOOTSTRAP protection domain represents the permissions granted to any code loaded by the bootstrap class loader, which is responsible for loading the class files of the Java API. Code in the BOOTSTRAP

Figure 3-6
Stack inspection for `Example2a`, where all frames have permission

class	method	protection domain	
Example2b	main()	CDROM	1
com.artima.security.stranger.Stranger	doYourThing()	STRANGER	2
com.artima.security.friend.Friend	doYourThing()	FRIEND	3
TextFileDisplayer	doYourThing()	CDROM	4
java.io.FileReader	<init>()	BOOTSTRAP	5
java.io.FileInputStream	<init>()	BOOTSTRAP	6
java.lang.SecurityManager	checkRead()	BOOTSTRAP	7
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	8
java.security.AccessController	checkPermission()	BOOTSTRAP	9
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	10

protection domain is granted java.lang.AllPermission, which gives it permission to do any action.

To get the Example2a application to demonstrate stack inspection as intended, you must start the application with an appropriate command. When using the java program from the Java 2 SDK Version 1.2, you will find that the appropriate command takes the following form:

```
java -Djava.security.manager -Djava.security.policy=
policyfile.txt -Dcom.artima.ijvm.cdrom.home=d:\books\
InsideJVM\manuscript\cdrom -cp
.;jars/friend.jar;jars/stranger.jar Example2a
```

This command, which is contained in the ex2a.bat file in the security/ex2 directory of the CD-ROM, is an example of the kind of command that you will need to get the example to work. By defining the java.security.manager property on the command line, you indicate that you want the concrete SecurityManager to be automatically installed. Because the Example2a application does not install a security manager explicitly, if you neglect to define the java.security.manager property on the command line, no security manager will be installed—and the code will have the capacity to do any task. The -cp argument sets up the class path, which causes the virtual machine to look for class files in the current directory and in the friend.jar and stranger.jar files in the jars subdirectory. The com.artima.ijvm.cdrom.home property indicates the directory in which Doer, Example2a, and TextFileDisplayer are located. The third grant clause in policyfile.txt uses this property and corresponds to the protection domain called "CD-ROM." As a result, types Doer, Example2a, and TextFileDisplayer will be loaded into the CD-ROM protection domain and will be granted permission to read to both question.txt and answer.txt. To execute Example2a on your own system, you must set the com.artima.ijvm.cdrom.home property to the security/ex2 directory of your CD-ROM or to whichever directory you might have copied the security/ex2 directory from the CD-ROM.

When the AccessController performs its stack inspection, it starts at the top of the stack—frame 10—and heads down to frame one, which is the frame for the first method invoked by this thread, main() of class Example2a. In the case of the Example2a application, every frame on the call stack has permission to perform the action—to read the file "question.txt". This situation occurs because all four protection domains represented on the call stack—FRIEND, STRANGER, CD-ROM, and BOOTSTRAP—include or imply a FilePermission for reading question.txt in the current directory. When the

AccessController's checkPermission() method reaches the bottom of the stack without having encountered any frames that do not have permission to read the file, it returns normally without throwing an exception. The FileInputStream opens the file for reading. The Example2a application reads the contents of question.txt and prints them to the standard output, which looks similar to the following:

```
To what extent does complexity threaten security?
```

A Stack Inspection That Says "No"

As an example of a stack inspection that results in denied permission, consider the Example2b application from the security/ex2 directory of the CD-ROM:

```
// On CD-ROM in file security/ex2/Example2b.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;

// This fails because the Stranger code doesn't have
// permission to read file question.txt

class Example2b {

    public static void main(String[] args) {

        TextFileDisplayer tfd = new
TextFileDisplayer("answer.txt");

        Friend friend = new Friend(tfd, true);

        Stranger stranger = new Stranger(friend, true);

        stranger.doYourThing();
    }
}
```

The only difference between Example2b and the previous example, Example2a, is that whereas Example2a passes the filename "question.txt" to the TextFileDisplayer constructor, Example2b passes the filename "answer.txt". This small change to the application makes a big difference on the outcome of the program, however, because one of the methods on the stack does not have permission to access "answer.txt".

When the Example2b program invokes doYourThing() on the Stranger object referenced from the stranger variable, the Stranger object invokes doYourThing() on the Friend object, which invokes

doYourThing() on the TextFileDisplayer object. TextFileDisplayer's doYourThing() method attempts to open and read a file called "answer.txt" in the current directory (the directory in which the Example2b application was started) and print its contents to the standard output. When TextFileDisplayer's doYourThing() method creates a new FileReader object, the FileReader constructor creates a new FileInputStream whose constructor checks to see whether or not a security manager has been installed. In this case, the concrete SecurityManager has been installed, so the FileInputStream's constructor invokes checkRead() on the concrete SecurityManager. The checkRead() method instantiates a new FilePermission object representing the permission to read file answer.txt and passes that object to the concrete SecurityManager's checkPermission() method, which passes the object on to the checkPermission() method of the AccessController. The AccessController's checkPermission() method performs the stack inspection to determine whether this thread should be permitted to open the file answer.txt for reading.

The call stack to be inspected in Example2b, which is shown in Figure 3-7, looks identical to the call stack that was inspected in Example2a. The only difference is that this time, rather than making sure that every frame on the stack has permission to read file q uestion.txt, the AccessController will make sure that every frame on the stack has permission to read answer.txt. As always, stack inspection starts at the top of the stack and proceeds down the stack towards frame one. But this time, the inspection process never actually reaches frame one. When the AccessController reaches frame two, it discovers that the code of the Stranger class, to whom the doYourThing() method of frame two belongs, does not have permission to read "answer.txt". Because all frames of the stack must have permission, the stack inspection process do not need to go farther than frame two. The AccessController's checkPermission() method throws an AccessControl exception.

To get the Example2b application to work as intended, you must start the application with an appropriate command. When using the java program from the Java 2 SDK Version 1.2, the appropriate command takes the following form:

```
java -Djava.security.manager -Djava.security.policy=
policyfile.txt -Dcom.artima.ijvm.cdrom.home=d:\books\
InsideJVM\manuscript\cdrom -cp .;jars/friend.jar;jars/
stranger.jar Example2b
```

Figure 3-7 Stack inspection for `Example2b`, where frame two does not have permission

class	method	protection domain	
Example2b	main()	CDROM	1
com.artima.security.stranger.Stranger	doYourThing()	STRANGER	2
com.artima.security.friend.Friend	doYourThing()	FRIEND	3
TextFileDisplayer	doYourThing()	CDROM	4
java.io.FileReader	<init>()	BOOTSTRAP	5
java.io.FileInputStream	<init>()	BOOTSTRAP	6
java.lang.SecurityManager	checkRead()	BOOTSTRAP	7
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	8
java.security.AccessController	checkPermission()	BOOTSTRAP	9
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	10

This command, which is contained in the `ex2b.bat` file in the `security/ex2` directory of the CD-ROM, is an example of the kind of command that you will need to get the example to work. As before, to execute `Example2b` on your own system, you must set the `com.artima.ijvm.cdrom.home` property to the `security/ex2` directory of your CD-ROM—or to whichever directory you might have copied the `security/ex2` directory from the CD-ROM. When you run this program, you should see the following output:

```
Exception in thread "main" java.security.
AccessControlException: access denied (java.io.
FilePermission answer.txt read)
     at java.security.AccessControlContext.
     checkPermission(AccessControlContext.java:195)
     at java.security.AccessController.checkPermission
     (AccessController.java:403)
     at java.lang.SecurityManager.checkPermission
     (SecurityManager.java:549)
     at java.lang.SecurityManager.checkRead
     (SecurityManager.java:873)
     at java.io.FileInputStream.<init>(FileInputStream.
     java:65)
     at java.io.FileReader.<init>(FileReader.java:35)
     at TextFileDisplayer.doYourThing(TextFileDisplayer.
     java, Compiled Code)
     at com.artima.security.friend.Friend.doYourThing
     (Friend.java:21)
     at com.artima.security.stranger.Stranger.doYourThing
     (Stranger.java:21)
     at Example2b.main(Example2b.java:18)
```

The doPrivileged( ) Method

The basic algorithm illustrated so far in this chapter, in which the AccessController inspects the stack from top to bottom—stubbornly requiring that every frame should have permission to perform an action —prevents less-trusted code from hiding behind more-trusted code. Because the AccessController looks all the way down the call stack, it will eventually find any method that is not trusted to perform the requested action. For example, although the untrusted Stranger object of Example2b places the trusted code of Friend and TextFileDisplayer between it and the Java API method that attempts to open file answer.txt, the untrusted Stranger code cannot hide behind that trusted code. As shown in Figure 3-7, although the AccessController must look through eight frames that have permission to read answer.txt before it encounters frame two, it eventually reaches frame two. Once it arrives at frame two, it discovers the doYourThing() method of class Stranger, whose associated protection domain does not have permission to read answer.txt. As a result of this discovery, the AccessController throws an AccessControllerException, thereby prohibiting the read.

The basic AccessController algorithm prevents any code from performing (or causing to perform) any action that the code is not trusted to carry out. Methods belonging to a less-powerful protection domain, therefore, cannot gain privileges by invoking methods belonging to more powerful protection domains. The basic algorithm also implies that methods belonging to more powerful protection domains must give up privileges when calling methods belonging to less powerful protection domains. Although the basic algorithm provides behavior that is desirable in general, the AccessController's stubborn insistence that all frames on the call stack have permission to perform the requested action can be a bit restrictive at times.

Sometimes code farther up the call stack (closer to the top of the stack) might wish to perform an action that code farther down the call stack might not be permitted to do. For example, imagine that an untrusted applet asks the Java API to render a string of text in bold Helvetica font on its applet panel. To fulfill this request, the Java API might need to open a font file on the local disk to load a bold Helvetica font with which to render the text on behalf of the applet. Because it belongs to the Java API, the class making the explicit request to open the font file more than likely has permission to open the file. The code of the untrusted applet, however, which is represented by a stack frame farther down the call stack, more

than likely does not have permission to open the file. Given the basic algorithm, the AccessController would prevent the opening of the font file because the code for the untrusted applet, sitting somewhere on the stack, does not have permission to open the file.

To enable trusted code to perform actions for which less-trusted code farther down the call stack might not have permission, the AccessController class offers four overloaded static methods called doPrivileged(). Each of these methods accepts as a parameter an object that implements either the java.security.PrivilegedAction or java.security.PrivilegedExceptionAction interface. Both of these interfaces declare one method called run() that takes no parameters and returns void. The only difference between these two interfaces is that whereas PrivilegedExceptionAction's run() method declares Exception in its throws clause, PrivilegedAction declares no throws clause. To perform an action despite the existence of less-trusted code farther down the call stack, you create an object that implements one of the PrivilegedAction interfaces whose run() method performs the action and pass that object to doPrivileged().

When you invoke doPrivileged(), as when you invoke any method, a new frame is pushed onto the stack. In the context of a stack inspection by the AccessController, a frame for a doPrivileged() method invocation signals an early termination point for the inspection process. If the protection domain associated with the method that invoked doPrivileged() has permission to perform the requested action, the AccessController returns immediately. The protection domain permits the action even if code farther down the stack does not have permission to perform the action.

If an untrusted applet asks the Java API to render a test string on its applet panel, therefore, the Java API code can open the local font file by wrapping the file open action in a doPrivileged() call. The AccessController will enable such a request, although the untrusted applet code does not have permission to open the file. Because the frame for the untrusted applet code is beneath the frame for the doPrivileged() invocation by the Java API code, the AccessController will not even consider the permissions of the untrusted applet code.

For an example of a doPrivileged() method invocation, consider again the doYourThing() method of class Friend:

```
// On CD-ROM in file
// security/ex2/com/artima/security/friend/Friend.java
package com.artima.security.friend;
import com.artima.security.doer.Doer;
import java.security.AccessController;
```

```
import java.security.PrivilegedAction;

public class Friend implements Doer {

    private Doer next;
    private boolean direct;

    public Friend(Doer next, boolean direct) {
        this.next = next;
        this.direct = direct;
    }

    public void doYourThing() {

        if (direct) {

            next.doYourThing();
        }
        else {
            AccessController.doPrivileged(
                new PrivilegedAction() {
                    public Object run() {
                        next.doYourThing();
                        return null;
                    }
                }
            );
        }
    }
}
```

If the `direct` instance variable is `false`, then `Friend`'s `doYourThing()` method will simply invoke `doYourThing()` directly on the `next` reference. But if `direct` is `true`, `doYourThing()` will wrap the invocation of `doYourThing()` on the `next` reference in a `doPrivileged()` call. To do so, `Friend` instantiates an anonymous inner class that implements `PrivilegedAction`, whose `run()` method invokes `doYourThing()` on `next` and passes that object to `doPrivileged()`.

To see `Friend`'s `doPrivileged()` invocation in action, consider the `Example2c` application from the `security/ex2` directory of the CD-ROM:

```
// On CD-ROM in file security/ex2/Example2c.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;

// This succeeds because Friend code executes a
// doPrivileged() call. (Passing false as
// the second arg to Friend constructor causes
// it to do a doPrivileged().)
```

```
class Example2c {

    public static void main(String[] args) {

        TextFileDisplayer tfd = new TextFileDisplayer
        ("answer.txt");

        Friend friend = new Friend(tfd, false);

        Stranger stranger = new Stranger(friend, true);

        stranger.doYourThing();
    }
}
```

Only one difference exists between the main() method of the Example2c application and the main() method of the previous example: Example2b. When the Example2b application instantiated the Friend object, it passed true as the second parameter. Example2c passes false. If you look back at the code for Friend (and Stranger) shown earlier in this chapter, you will see that this parameter is used to decide whether to invoke doYourThing() directly on the Doer passed as the first parameter to the constructor. Because Example3c passes false, the Friend class will not invoke doYourThing() directly but will invoke it indirectly via an AccessController.doPrivileged() invocation.

When the Example2c program invokes doYourThing() on the Stranger object referenced from the stranger variable, the Stranger object invokes doYourThing() on the Friend object, which (because direct is false) invokes doPrivileged(), passing the anonymous inner class instance that implements PrivilegedAction. The doPrivileged() method invokes run() on the passed PrivilegedAction object, which invokes doYourThing() on the TextFileDisplayer object.

As in the previous example, TextFileDisplayer's doYourThing() method attempts to open and read a file called "answer.txt" in the current directory and print its contents to the standard output. When TextFileDisplayer's doYourThing() method creates a new FileReader object, the FileReader constructor creates a new FileInputStream whose constructor checks to see whether or not a security manager has been installed. Once again, the concrete SecurityManager has been installed, so the FileInputStream's constructor invokes checkRead() on the concrete SecurityManager. The checkRead() method instantiates a new FilePermission object

representing permission to read the file answer.txt and passes that object to the concrete SecurityManager's checkPermission() method, which passes the object on to the checkPermission() method of the AccessController. The AccessController's checkPermission() method performs the stack inspection to determine whether this thread should be permitted to open file answer.txt for reading. The stack appears as shown in Figure 3-8.

The call stack to be inspected in Example2c looks similar to the call stacks inspected in Example2a and Example2b. The difference is that Example2c's call stack has two extra frames: frame four, which represents the doPrivileged() invocation, and frame five, which represents the run() invocation on the PrivilegedAction object. As always, stack inspection starts at the top of the stack and proceeds down the stack towards frame one. But once again, the inspection process will not actually reach frame one. When the AccessController reaches frame four, it discovers a doPrivileged() invocation. As a result of this discovery, the AccessController makes one more check on the code represented by frame three. This code invoked doPrivileged() and has permission to read answer.txt. Because frame three is associated with the FRIEND protection domain, which does have permission to read question.txt, the AccessController's checkPermission() method returns normally. Because the AccessController stopped its inspection at frame three, it never considered frame two. Because frame two is associated with the STRANGER protection domain, it does not have permission

Figure 3-8
Stack inspection for Example2c, which stops at frame three

class	method	protection domain	
Example2b	main()	CDROM	1
com.artima.security.stranger.Stranger	doYourThing()	STRANGER	2
com.artima.security.friend.Friend	doYourThing()	FRIEND	3
java.security.AccessController	dPrivileged()	BOOTSTRAP	4
com.artima.security.friend.Friend$1	run()	FRIEND	5
TextFileDisplayer	doYourThing()	CDROM	6
java.io.FileReader	<init>()	BOOTSTRAP	7
java.io.FileInputStream	<init>()	BOOTSTRAP	8
java.lang.SecurityManager	checkRead()	BOOTSTRAP	9
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	10
java.security.AccessController	checkPermission()	BOOTSTRAP	11
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	12

to read answer.txt. Thus, by invoking doPrivileged(), the Friend code could read file answer.txt—although the code beneath it on the call stack did not have permission to open the file.

To get the Example2c application to work as intended, you must (as with the previous examples) start the application with an appropriate command. When using the java program from Java 2 SDK Version 1.2, the appropriate command takes the following form:

```
java -Djava.security.manager -Djava.security.policy=
policyfile.txt -Dcom.artima.ijvm.cdrom.home=d:\books\
InsideJVM\manuscript\cdrom -cp .;jars/friend.jar;jars/
stranger.jar Example2c
```

This command, which is contained in the ex2c.bat file in the security/ex2 directory of the CD-ROM, is an example of the kind of command that you will need to get the example to work. As before, to execute Example2c on your own system, you must set the com.artima.ijvm.cdrom.home property to the security/ex2 directory of your CD-ROM or to whichever directory you might have copied the security/ex2 directory from the CD-ROM. When you run this program, it should print the contents of answer.txt as follows:

```
Complexity threatens security to a significant extent. The
more
complicated a security infrastructure becomes, the more
likely
parties responsible for configuring security will either
make
mistakes that open up security holes or avoid using the
security infrastructure altogether.
```

A Futile Use of doPrivileged( )

You should understand that a method can never grant itself more privileges than it already has with a doPrivileged() invocation. By calling doPrivileged(), a method is merely enabling privileges that it already possesses. The method is telling the AccessController that it is taking responsibility for exercising its own permissions, and that the AccessController should ignore the permissions of its callers. Thus, the doPrivileged() call in the previous example, Example2c, enabled answer.txt to be read because Friend, the class that executed the doPrivileged(), already had permission to read the file—and so did all the frames above it on the stack.

For an example of a futile attempt to use doPrivileged(), consider the Example2d application from the security/ex2 directory of the CD-ROM:

```
// On CD-ROM in file security/ex2/Example2d.java
import com.artima.security.friend.Friend;
import com.artima.security.stranger.Stranger;

// This fails because even though Stranger does
// a doPrivileged() call, Stranger doesn't have
// permission to read question.txt. (Passing
// false as second arg to Stranger constructor
// causes it to do a doPrivileged().)

class Example2d {

    public static void main(String[] args) {

        TextFileDisplayer tfd = new TextFileDisplayer
        ("answer.txt");

        Stranger stranger = new Stranger(tfd, false);

        Friend friend = new Friend(stranger, true);

        friend.doYourThing();
    }
}
```

The difference between Example2d and the previous example, Example2c, is that the Stranger and Friend objects have swapped positions and roles. The Stranger object is now farther up the stack, with the Friend below it on the stack. And this time, it is Stranger that will make the call to doPrivileged(), not Friend.

When the Example2d program invokes doYourThing() on the Friend object referenced from the friend variable, the Friend object invokes doYourThing() on the Stranger object—which, because direct is false—invokes doPrivileged(), passing the anonymous inner-class instance that implements PrivilegedAction. The doPrivileged() method invokes run() on the passed PrivilegedAction object, which invokes doYourThing() on the TextFileDisplayer object.

As in the previous two examples, TextFileDisplayer's doYourThing() method attempts to open and read a file called "answer.txt" in the current directory and print its contents to the standard output. When TextFileDisplayer's doYourThing() method creates a new FileReader object, the FileReader constructor creates a new FileInputStream whose constructor checks to see whether or not

a security manager has been installed. As in all of the examples, the concrete `SecurityManager` has been installed, so the `FileInputStream`'s constructor invokes `checkRead()` on the concrete `SecurityManager`. The `checkRead()` method instantiates a new `FilePermission` object that represents the permission to read the file `answer.txt` and passes that object to the concrete `SecurityManager`'s `checkPermission()` method, which passes the object on to the `checkPermission()` method of the `AccessController`. The `AccessController`'s `checkPermission()` method performs the stack inspection to determine whether this thread should be permitted to open file `answer.txt` for reading. The stack presented to the `AccessController` by `Example2d` is shown in Figure 3-9.

The call stack to be inspected in `Example2d` looks similar to the call stack inspected in `Example2c`. The only difference is that `Friend` and `Stranger` have swapped positions. As always, stack inspection starts at the top of the stack and proceeds down the stack towards frame one. But alas, once again the inspection process will not actually reach frame one. When the `AccessController` reaches frame five, it discovers a stack frame associated with the STRANGER protection domain, which does not have permission to read `answer.txt`. As a result of this discovery, the `AccessController` throws an `AccessControlException`, indicating that the requested read of `answer.txt` should not be performed.

Had the `Stranger` class possessed the capability to enlist the assistance of an instance of some class that implemented `PrivilegedAction`, performed the desired invocation of the `TextFileDisplayer`'s

Figure 3-9
The stack inspection for `Example2d`, where frame five does not have permission

class	method	protection domain	
Example2b	main()	CDROM	1
com.artima.security.stranger.Friend	doYourThing()	FRIEND	2
com.artima.security.friend.Stranger	doYourThing()	STRANGER	3
java.security.AccessController	dPrivileged()	BOOTSTRAP	4
com.artima.security.friend.Stranger$1	run()	STRANGER	5
TextFileDisplayer	doYourThing()	CDROM	6
java.io.FileReader	<init>()	BOOTSTRAP	7
java.io.FileInputStream	<init>()	BOOTSTRAP	8
java.lang.SecurityManager	checkRead()	BOOTSTRAP	9
java.lang.SecurityManager	checkPermission()	BOOTSTRAP	10
java.security.AccessController	checkPermission()	BOOTSTRAP	11
java.security.AccessControlContext	checkPermission()	BOOTSTRAP	12

doYourThing() method, and belonged to a protection domain that had permission to read answer.txt, then Stranger's attempt to open answer.txt with the help of doPrivileged() would have still been futile. Imagine, for example, that the code of the run() method represented by frame five of Example2d's call stack had been associated with the CD-ROM protection domain. In that case, the AccessController would have determined that frame five had permission to open answer.txt and continued to frame four. At frame four, the AccessController would have discovered the doPrivileged() invocation. As a result of this discovery, the AccessController would make one more check: it would make certain that the method that invoked doPrivileged(), which in this case was Stranger's doYourThing() method represented by stack frame three, has permission to read file answer.txt. Because frame three is associated with the STRANGER protection domain that does not have permission to read answer.txt, the AccessController would still throw an AccessControlException.

To get the Example2d application to work as intended, you must start the application with yet another appropriate command. When using the java program from the Java 2 SDK Version 1.2, the appropriate command takes the following form:

```
java -Djava.security.manager -Djava.security.policy=
policyfile.txt -Dcom.artima.ijvm.cdrom.home=d:\books\
InsideJVM\manuscript\cdrom -cp .;jars/friend.jar;jars/
stranger.jar Example2d
```

This command, which is contained in the ex2d.bat file in the security/ex2 directory of the CD-ROM, is an example of the kind of command you will need to get the example to work. As before, to execute Example2d on your own system, you must set the com.artima.ijvm.cdrom.home property to the security/ex2 directory of your CD-ROM or to whichever directory you might have copied the security/ex2 directory from the CD-ROM. When you run this program, you should see the kind of output that crackers everywhere hate to see:

```
Exception in thread "main" java.
security.AccessControlException: access denied
(java.io.FilePermission answer.txt read)
      at java.security.AccessControlContext.
      checkPermission(AccessControlContext.java:195)
     at java.security.AccessController.checkPermission
     (AccessController.java:403)
     at java.lang.SecurityManager.checkPermission
     (SecurityManager.java:549)
     at java.lang.SecurityManager.checkRead
     (SecurityManager.java:873)
```

```
at java.io.FileInputStream.<init>(FileInputStream.
java:65)
at java.io.FileReader.<init>(FileReader.java:35)
at TextFileDisplayer.doYourThing(TextFileDisplayer.
java, Compiled Code)
at com.artima.security.stranger.Stranger$1.run
(Stranger.java:27)
at java.security.AccessController.doPrivileged
(Native Method)
at com.artima.security.stranger.Stranger.doYourThing
(Stranger.java:24)
at com.artima.security.friend.Friend.doYourThing
(Friend.java:21)
at Example2d.main(Example2d.java:21)
```

Missing Pieces and Future Directions

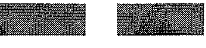
Java's security model, while far reaching, does not address all potential threats posed by mobile code. For example, two potential activities of malicious mobile code that are not currently addressed by Java's security model are as follows:

- 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 they deny the end-users from using their own computers. The Java security model does not currently offer ways to limit the usage of threads or memory by untrusted code. 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. Nevertheless, this kind of attack is a serious concern in certain situations, such as mission-critical servers that run Java servlets.

Another area not currently incorporated into the security model is the idea of awarding permissions to principals on whose behalf code is being executed. A familiar example of this kind of access control is the UNIX operating system, which controls access to files based on a user ID that can only be obtained via an correct login name and password. As this kind of access control will be important in distributed systems such as those made possible by Jini, Sun is actively working to add this kind of user-centric security functionality to Java. The aim of the *Java Authentication and Authorization Service* (JAAS) is to enable access control to be based

not just on the permissions granted to codebases and signers, but also on permissions granted to principals: the users who execute the code.



Security Beyond the Architecture

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 can't read the word processing file of your top-secret business plan if you:

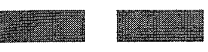
- routinely download untrusted native executables from the Internet and run them
- throw away extra printed copies of your business plan without shredding them
- leave your doors unlocked when you're gone
- hire someone to help you who is actually a spy for your arch-rival

In the context of a comprehensive security strategy, however, Java's security model can play a useful role.

The nice thing about Java's security model is that once you set it up, it does most of the work for you. You don't 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. The trouble is that, even though the designers of Java's security infrastructure did a good job of keeping things as simple as possible, the high degree functionality and flexibility offered by the security infrastructure demands a significant degree of complexity. As mentioned in the `answer.txt` file, which class `Stranger` so very much wanted to read in the `AccessController` examples given earlier in this chapter, complexity itself can represent a threat to security. The more complicated a security infrastructure becomes, the more likely parties responsible for configuring security will either make mistakes that open up security holes or avoid using the security infrastructure altogether.

End-users of Java software cannot rely solely on the security mechanisms built into Java's 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 Java's security strategy is that anyone can sign a license agreement and get a copy of the source code of Sun's Java Platform implementation. Instead of keeping the internal implementation of Java's 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 Java's internal implementation is part of Java's overall security strategy. Besides openness, there are several other aspects to Java's overall security strategy that don't directly involve its architecture. For more information about Java's overall security strategy, visit the resources page.



The Resources Page

For more information about Java and security, see the resources page: `http://www.artima.com/insidejvm/resources/`.

CHAPTER 5

The Java Virtual Machine

The previous four chapters of this book gave a broad overview of Java's architecture. They showed how the Java virtual machine fits into the overall architecture, relative to other components such as the language and the API. The remainder of this book will focus more narrowly on the Java virtual machine. This chapter gives an overview of the Java virtual machine's 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 primarily describes 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 might be talking about any of three different items. You might be speaking of any of the following:

- The abstract specification
- A concrete implementation
- A run-time 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 run-time instance hosts a single running Java application.

Each Java application runs inside a run-time 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, we added one of the following terms—specification, implementation, or instance—to the term *Java virtual machine*.

The Lifetime of a Java Virtual Machine

A run-time instance of the Java virtual machine has a clear mission in life: to run one Java application. When a Java application starts, a run-time 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—you will receive 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 or static, must return `void`, and must 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 as such:

```
// 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();
    }
}
```

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 Sun's Java 2 SDK. If you wanted to run the `Echo` application using Sun's `java` on Windows 98, for example, you would type in a command such as the following:

```
java Echo Greetings, Planet.
```

The first word in the command, "`java`," indicates that the Java virtual machine from Sun's Java 2 SDK 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 called `main()` that returns `void` and that takes a `String` array as its only parameter. The subsequent words, "`Greetings, Planet.`," are the command-line arguments for the application. These words are passed to the `main()` method in the `String` array in the order in which they appear on the command line. Therefore, for the previous example, the contents of the `String` array that are passed to main in `Echo` are as follows:

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

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

The `main()` method of an application's initial class serves as the starting point for that application's 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 that 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 class `System`.

In the `Echo` application (shown previously), the `main()` method does not invoke any other threads. After it prints out the command-line arguments, `main()` returns. This action terminates the application's 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 but 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*, which is a mechanism for loading types (classes and interfaces) when given fully qualified names. Each Java virtual machine also has an *execution engine*, which is 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 items—including bytecodes and other information that it extracts from loaded class files, objects that 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.

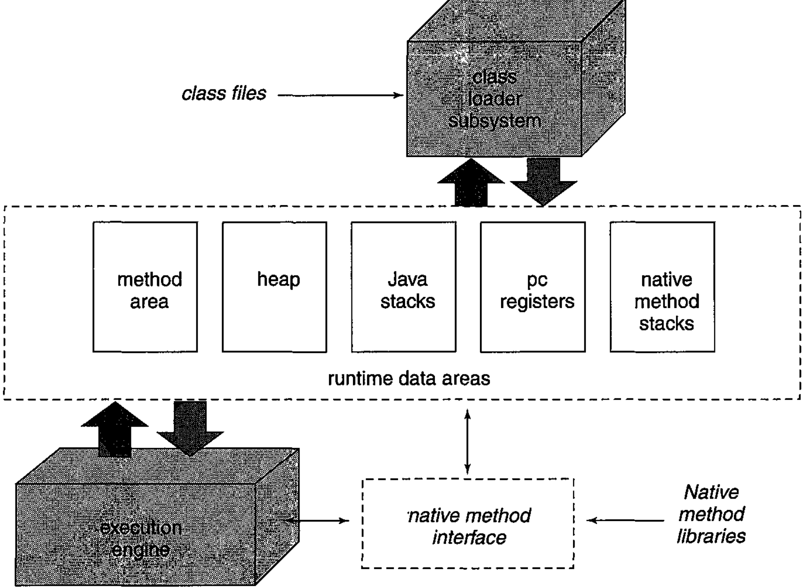
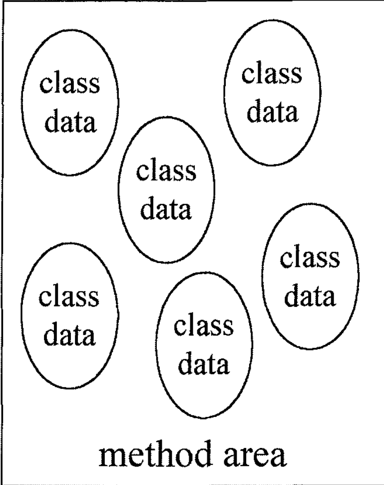


Figure 5-1
The internal architecture of the Java virtual machine

Different implementations of the virtual machine can have different memory constraints. Some implementations might have a lot of memory in which to work, while others might have little. Some implementations might have the capacity to take advantage of virtual memory, while others might 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 application's 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, then places this type information into the method area. As the program runs, the virtual machine places all objects that 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 receives 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 tells the next instruction to execute. A thread's Java stack stores the state of Java



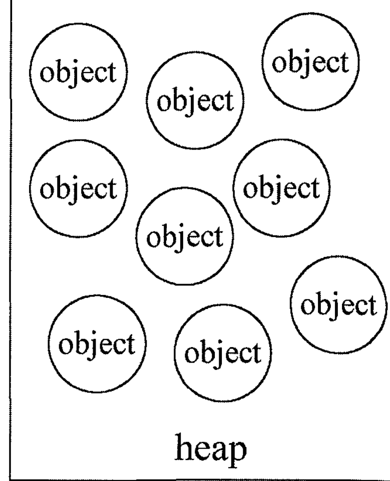


Figure 5-2 Runtime data areas that are shared among all threads

method invocations (not native 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*), which contain the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's 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 Java's designers to keep the Java virtual machine's instruction set compact and to facilitate implementation on architectures with few or irregular general-purpose registers. In addition, the stack-based architecture of the Java virtual machine's instruction set facilitates the code optimization work done by just-in-time and dynamic compilers that operate at run time in some virtual machine implementations.

See Figure 5-3 for a graphical depiction of the memory areas that the Java virtual machine creates for each thread. These areas are private to the owning thread, and no thread can access the PC register or Java stack of another thread.

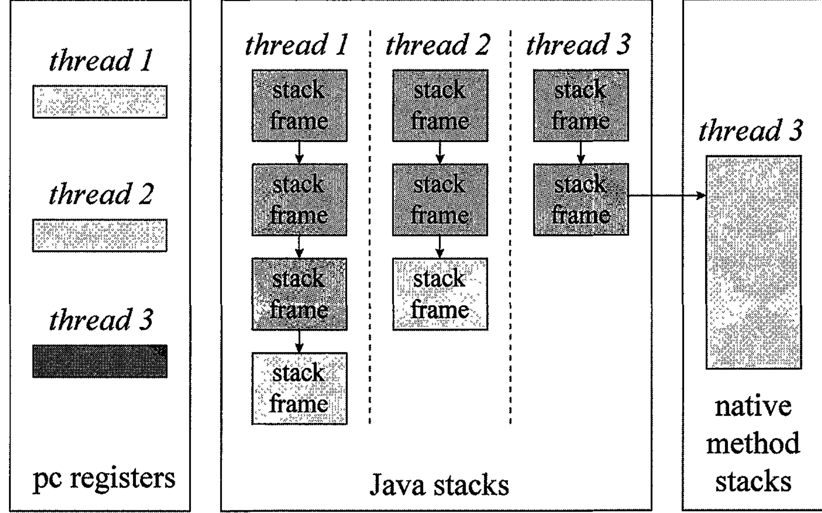


Figure 5-3 Runtime data areas that are exclusive to each thread

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 downward. 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 that the next instruction should 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—are undefined.

Data Types

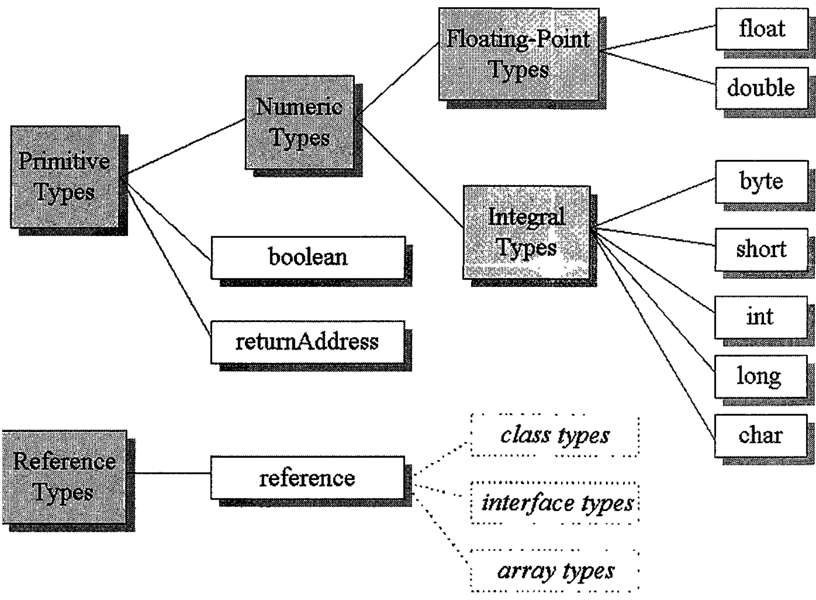
The Java virtual machine computes by performing operations on certain types of data. Both the data types and operations are strictly defined by the Java virtual machine specification. The data types can be divided into a set of *primitive types* and a *reference type*. Variables of the primitive types 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. In Figure 5-4, you can see a graphical depiction of the Java virtual machine's families of data types.

All of the primitive types of the Java programming language are primitive types of the Java virtual machine. Although `boolean` qualifies as a primitive type of the Java virtual machine, the instruction set has limited support for this type. 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, the integer zero represents `false`, and any non-zero integer represents `true`. Operations involving `boolean` values use `int`s. Arrays of `boolean` are accessed as arrays of `byte`, although they can 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 between 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 similar to a 64-bit, signed twos complement number, which is independent of the underlying host platform.

Figure 5-4 Data types of the Java virtual machine



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

The reference type of the Java virtual machine is 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 type's values are references to class instances. The array type's values are references to arrays, which are full-fledged objects in the Java virtual machine. The interface type's values are references to class instances that implement an interface. One other reference value is the `null` value, which indicates that the `reference` variable does not 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 that the designers of individual implementations have to make. The ranges of the Java virtual machine's data types are shown in Table 5-1. We give you more information about the floating-point ranges in Chapter 14, "Floating Point Arithmetic."

Table 5-1

Ranges of the Java virtual machine's 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
`returnAddress`	Address of an opcode within the same method
`reference`	Reference to an object on the heap or to `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`, `returnAddress`, 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 machine's runtime data areas are based on 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 machine's 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; rather, 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 Java's Architecture," gives an overview of this subsystem. Chapter 3, "Security," shows how the subsystem fits into Java's security model. This chapter describes the class loader subsystem in more detail and shows how it relates to the other components of the virtual machine's internal architecture.

As mentioned in Chapter 1, the Java virtual machine contains two kinds of class loaders: a *bootstrap class loader* and a *user-defined class loader*. The bootstrap class loader is part of the virtual machine implementation, and user-defined class loaders 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, user-defined class loaders are regular Java objects whose

class descends from `java.lang.ClassLoader`. The methods of class `ClassLoader` enable Java applications to access the virtual machine's class-loading machinery. Also, for every type that a Java virtual machine loads, it creates an instance of class `java.lang.Class` to represent that type. Like all objects, user-defined class loaders 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. This subsystem must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist with 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
3. *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 Type."

The Bootstrap Class Loader Java virtual machine implementations must have the capability 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.

Every Java virtual machine implementation has a bootstrap class loader that knows how to load trusted classes—including the classes of the Java API. The Java virtual machine specification does not define how the bootstrap loader should locate classes. Implementation designers must make that decision.

Given a fully qualified type name, the bootstrap class loader must *in some way* attempt to produce the data that defines the type. One common approach is demonstrated by the Java virtual machine implementation

in Sun's Version 1.1 JDK for Windows 98. This implementation searches a user-defined directory path stored in an environment variable called CLASSPATH. The bootstrap class loader looks in each directory (in the order the directories appear in the CLASSPATH) until it finds a file with the appropriate name: the type's simple name plus ".class". Unless the type is part of the unnamed package, the bootstrap 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 bootstrap class loader is searching for class java.lang.Object, it will look for Object.class in the java\lang subdirectory of each CLASSPATH directory.

In Version 1.2, the bootstrap class loader of Sun's Java 2 SDK only looks in the directory in which the system classes (the class files of the Java API) were installed. The bootstrap class loader of the implementation of the Java virtual machine from Sun's Java 2 SDK does not look on the CLASSPATH. In Sun's Java 2 SDK virtual machine, searching the class path is the job of the *system class loader*, a user-defined class loader that is created automatically when the virtual machine starts. More information about the class-loading scheme of Sun's Java 2 SDK is given in Chapter 8, "The Linking Model."

User-Defined Class Loaders Although user-defined class loaders themselves are part of the Java application, four of the methods in class ClassLoader are gateways to the Java virtual machine:

```
// Four of the methods declared in class java.lang.
ClassLoader:
protected final Class defineClass(String name, byte data[],
     int offset, int length);
protected final Class defineClass(String name, byte data[],
     int offset, int length, ProtectionDomain
protectionDomain);
protected final Class findSystemClass(String name);
protected final void resolveClass(Class c);
```

Any Java virtual machine implementation must take care to connect these methods of class ClassLoader to the internal class loader subsystem.

The two overloaded defineClass() methods accept 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—with the fully qualified name specified in

name. The type is assigned to either a default protection domain, if the first version of `defineClass()` is used, or to the protection domain object referenced by the `protectionDomain` parameter. Every Java virtual machine implementation must make sure that 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 user-defined class loader invokes this method in Versions 1.0 and 1.1, the class loader requests that the virtual machine attempts to load the named type via its bootstrap class loader. If the bootstrap class loader has already loaded or successfully loads the type, it returns a reference to the `Class` object representing the type. If it cannot locate the binary data for the type, the loader throws `ClassNotFoundException`. In Version 1.2, the `findSystemClass()` method attempts to load the requested type from the system class loader. Every Java virtual machine implementation must make sure that the `findSystemClass()` method can invoke the bootstrap class loader (if running Version 1.0 or 1.1) or system class loader (if running Version 1.2 or later) 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 (if it has not already been linked). The `defineClass()` method described previously only takes care of loading. (See the previous 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 has not necessarily been linked and initialized. Java virtual machine implementations make sure that the `resolveClass()` method of class `ClassLoader` can cause the class loader subsystem to perform linking.

The details of how a Java virtual machine performs class loading, linking, and initialization with user-defined class loaders 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 type's fully qualified name, therefore, is not always enough to uniquely identify the type 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 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 order (most significant byte first). 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 might decide to store multi-byte values in the method area in little-endian order (less significant byte first).

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 they must also think of compactness. If designing an implementation that will operate under low memory constraints, designers might decide to trade some execution speed in favor of compactness. If designing an implementation that will run on a virtual memory system, designers might decide to store redundant information in the method area to facilitate execution speed. (If the underlying host does not 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(s) that they feel optimize their implementation's performance in the context of its requirements.

All threads share the same method area, so access to the method area's data structures must be designed to be threadsafe. If two threads are attempting to find a class called `Lava`, for example, and `Lava` has not yet been loaded, only one thread should be permitted to load `Lava` while the other one waits.

The size of the method area does not need to be fixed. As the Java application runs, the virtual machine can expand and contract the method area to fit the application's needs. Also, the memory of the method area does not need to be contiguous; instead, it could be allocated on a heap—even on the virtual machine's own heap. Implementations can enable 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 user-defined class loaders, classes can become unreferenced by the application. If a class becomes unreferenced, a Java virtual machine can unload the class (garbage collect the class) 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 Type."

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 type's 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 type's modifiers (some subset of `public`, `abstract`, or `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 type's package, plus a dot, plus the type's *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 whichever form and data structures a designer chooses.

In addition to the basic type information listed previously, the virtual machine must also store the following information 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.

The Constant Pool For each type it loads, a Java virtual machine must store a *constant pool*. A constant pool is an ordered set of constants used by the type, including literals (string, integer, and floating point constants) and symbolic references to types, fields, and methods. Entries in the constant pool are referenced by index, much like the elements of an array. Because it holds symbolic references to all types, fields, and methods used by a type, the constant pool plays a central role in the dynamic linking of Java programs. The constant pool is described in more detail later in this chapter and in Chapter 6, "The Java Class File."

Field Information For each field declared in the type, the following information must be stored in the method area. In addition to the information for each field, the order in which the fields are declared by the class or interface must also be recorded. Here is the list for fields:

- The field's name
- The field's type
- The field's modifiers (some subset of `public`, `private`, `protected`, `static`, `final`, `volatile`, `transient`)

Method Information For each method declared in the type, the following information must be stored in the method area. As with fields, the order in which the methods are declared by the class or interface must be recorded, as well as the data. Here is the list:

- The method's name
- The method's return type (or `void`)

- The number and types (in order) of the method's parameters
- The method's modifiers (some subset of `public`, `private`, `protected`, `static`, `final`, `synchronized`, `native`, or `abstract`)

In addition to the items listed previously, the following information must also be stored with each method that is not abstract or native:

- The method's bytecodes
- The sizes of the operand stack and local variables sections of the method's stack frame (these are described in a later section of this chapter)
- An exception table (this concept is described in Chapter 17, "Exceptions")

Class Variables Class variables are shared among all instances of a class and can be accessed even in the absence of any instance. These variables are associated with the class—not with instances of the class—so they are logically part of the class data in the method area. Before a Java virtual machine uses a class, it must allocate memory from the method area for each non-final class variable declared in the class.

Constants (class variables declared final) are not treated in the same way as non-final class variables. Every type that uses a final class variable gets a copy of the constant value in its own constant pool. As part of the constant pool, final class variables are stored in the method area—just like non-final class variables. Whereas non-final class variables are stored as part of the data for the type that *declares* them, however, final class variables are stored as part of the data for any type that *uses* them. This special treatment of constants is explained in more detail in Chapter 6, "The Java Class File."

A Reference to Class ClassLoader For each type it loads, a Java virtual machine must keep track of whether or not the type was loaded via the bootstrap class loader or by a user-defined class loader. For those types loaded via a user-defined class loader, the virtual machine must store a reference to the user-defined class loader that loaded the type. This information is stored as part of the type's data in the method area.

The virtual machine uses this information during dynamic linking. When one type refers to another type, the virtual machine requests the

referenced type from the same class loader that loaded the referencing type. This process of dynamic linking is also central to the way the virtual machine forms separate name spaces. To properly perform dynamic linking and maintain multiple name spaces, the virtual machine needs to know which class loader loaded each type in its method area. The details of dynamic linking and name spaces are given in Chapter 8, "The Linking Model."

A Reference to Class Class An instance of class `java.lang.Class` is created by the Java virtual machine for every type it loads. The virtual machine must (in some way) associate a reference to the `Class` instance for a type with the type's data in the method area.

Your Java programs can obtain and use references to `Class` objects. One static method in class `Class` enables you to obtain a reference to the `Class` instance for any loaded class:

```
// A method declared in class java.lang.Class:
public static Class forName(String className);
```

If you invoke `forName("java.lang.Object")`, for example, you will receive a reference to the `Class` object that represents `java.lang.Object`. If you invoke `forName("java.util.Enumeration")`, you will receive a reference to the `Class` object that represents the `Enumeration` interface from the `java.util` package. You can use `forName()` to obtain a `Class` reference for any loaded type from any package, as long as the type can be (or already has been) loaded into the current name space. If the virtual machine is unable to load the requested type into the current name space, `forName()` will throw `ClassNotFoundException`.

An alternative way to obtain a `Class` reference is to invoke `getClass()` on any object reference. This method is inherited by every object from class `Object` itself:

```
// A method declared in class java.lang.Object:
public final Class getClass();
```

If you have a reference to an object of class `java.lang.Integer`, for example, you could get the `Class` object for `java.lang.Integer` simply by invoking `getClass()` on your reference to the `Integer` object.

Given a reference to a `Class` object, you can find out information about the type by invoking methods declared in `Class`. If you look at these methods, you will quickly realize that class `Class` gives the running

application access to the information stored in the method area. Here are some of the methods declared in class `Class`:

```
// Some of the methods declared in class java.lang.Class:
public String getName();
public Class getSuperClass();
public boolean isInterface();
public Class[] getInterfaces();
public ClassLoader getClassLoader();
```

These methods simply return information about a loaded type. `getName()` returns the fully qualified name of the type, and `getSuperClass()` returns the `Class` instance for the type's direct superclass. If the type is class `java.lang.Object` or is an interface, none of which have a superclass, then `getSuperClass()` returns `null`. `isInterface()` returns `true` if the `Class` object describes an interface and returns `false` if it describes a class. `getInterfaces()` returns an array of `Class` objects, one for each direct superinterface. The superinterfaces appear in the array in the order they are declared as superinterfaces by the type. If the type has no direct superinterfaces, `getInterfaces()` returns an array of length zero. `getClassLoader()` returns a reference to the `ClassLoader` object that loaded this type or returns `null` if the type was loaded by the bootstrap class loader. All of this information comes straight from the method area.

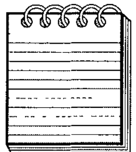
Method Tables The type information stored in the method area must be organized to be quickly accessible. In addition to the raw type information listed previously, implementations might include other data structures that hasten access to the raw data. One example of such a data structure is a *method table*. For each non-abstract class that a Java virtual machine loads, the machine could generate a method table and include it as part of the class information stored in its method area. A method table is an array of direct references to all of the instance methods that might be invoked on a class instance, including instance methods inherited from superclasses. (A method table is not helpful in the case of abstract classes or interfaces, because the program will never instantiate these items.) A method table enables a virtual machine to quickly locate an instance method invoked on an object. Method tables are described in detail in Chapter 8, "The Linking Model."

An Example of Method Area Use As an example of how the Java virtual machine uses the information it stores in the method area, consider the following classes:

```
// On CD-ROM in file jvm/ex2/Lava.java
class Lava {

    private int speed = 5; // 5 kilometers per hour

    void flow() {
    }
}

// On CD-ROM in file jvm/ex2/Volcano.java
class Volcano {

    public static void main(String[] args) {
        Lava lava = new Lava();
        lava.flow();
    }
}
```

The following paragraphs describe how an implementation might execute the first instruction in the bytecodes for the `main()` method of the `Volcano` application. Different implementations of the Java virtual machine can operate in different ways. The following description illustrates one way—but not the only way—in which a Java virtual machine could execute the first instruction of `Volcano`'s `main()` method.

To run the `Volcano` application, you give the name `Volcano` to a Java virtual machine in an implementation-dependent manner. Given the name `Volcano`, the virtual machine finds and reads file `Volcano.class`. Then, the machine extracts the definition of class `Volcano` from the binary data in the imported class file and places the information into the method area. The virtual machine then invokes the `main()` method by interpreting the bytecodes stored in the method area. As the virtual machine executes `main()`, it maintains a pointer to the constant pool (a data structure in the method area) for the current class (class `Volcano`).



NOTE: *Note that this Java virtual machine has already begun to execute the bytecodes for* `main()` *in class* `Volcano`, *although it has not yet loaded class* `Lava`. *Like many (probably most) implementations of the Java virtual machine, this implementation does not wait until all classes used by the application are loaded before it begins executing* `main()`. *This implementation loads classes only as it needs them.*

`main()`'s first instruction tells the Java virtual machine to allocate enough memory for the class listed in constant pool entry one. The virtual

machine uses its pointer to `Volcano`'s constant pool to look up entry one and finds a symbolic reference to class `Lava`. The machine checks the method area to see whether `Lava` has already been loaded.

The symbolic reference is just a string giving the class's fully qualified name: `"Lava"`. Here, you can see that the method area must be organized so that a class can be located as quickly as possible, given only the class's fully qualified name. Implementation designers can choose whichever algorithm and data structures that best fit their needs—a hash table, a search tree, anything. This same mechanism can be used by the static `forName()` method of class `Class`, which returns a `Class` reference when given a fully qualified name.

When the virtual machine discovers that it has not yet loaded a class called "`Lava`," it proceeds to find and read file `Lava.class`. The machine extracts the definition of class `Lava` from the imported binary data and places the information into the method area.

The Java virtual machine then replaces the symbolic reference in `Volcano`'s constant pool entry one, which is the string `"Lava"` with a pointer to the class data for `Lava`. If the virtual machine ever has to use `Volcano`'s constant pool entry one again, it will not have to go through the relatively slow process of searching the method area for class `Lava` when given only a symbolic reference—the string `"Lava"`. The machine can just use the pointer to access the class data for `Lava` in a quicker fashion. This process of replacing symbolic references with direct references (in this case, a native pointer) is called *constant pool resolution*. The symbolic reference is *resolved* into a direct reference by searching the method area until the referenced entity is found—loading new classes if necessary.

Finally, the virtual machine is ready to allocate memory for a new `Lava` object. Once again, the virtual machine consults the information stored in the method area and uses the pointer (which was just placed into `Volcano`'s constant pool entry one) to the `Lava` data (which was just imported into the method area) to find out how much heap space a `Lava` object requires.

A Java virtual machine can always determine the amount of memory required to represent an object by looking into the class data stored in the method area. The actual amount of heap space required by a particular object, however, is implementation dependent. The internal representation of objects inside a Java virtual machine is another decision left to implementation designers. Object representation is discussed in more detail later in this chapter.

Once the Java virtual machine has determined the amount of heap space required by a `Lava` object, it allocates that space on the heap and initializes the instance variable `speed` to zero (its default initial value). If class `Lava`'s superclass, `Object`, has any instance variables, those are also initialized to default initial values. (The details of initialization of both classes and objects are given in Chapter 7, "The Lifetime of a Type.")

The first instruction of `main()` completes by pushing a reference to the new `Lava` object onto the stack. A later instruction will use the reference to invoke Java code that initializes the `speed` variable to its proper initial value, five. Another instruction will use the reference to invoke the `flow()` method on the referenced `Lava` object.

The Heap

Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap. Because there is only one heap inside a Java virtual machine instance, all threads share the heap. Because a Java application runs inside its own exclusive Java virtual machine instance, there is a separate heap for every individual running application. Two different Java applications cannot trample on each other's heap data. Two different threads of the same application, however, could trample on each other's heap data. For this reason, you must be concerned about proper synchronization of multithreaded access to objects (heap data) in your Java programs.

The Java virtual machine has an instruction that allocates memory on the heap for a new object but has no instruction for freeing that memory. Just as you cannot explicitly free an object in Java source code, you cannot explicitly free an object in Java bytecodes. The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application. Usually, a Java virtual machine implementation uses a *garbage collector* to manage the heap.

Garbage Collection A garbage collector's primary function is to automatically reclaim the memory used by objects that are no longer referenced by the running application. The collector might also move objects as the application runs to reduce heap fragmentation.

A garbage collector is not strictly required by the Java virtual machine specification. The specification only requires that an implementation manages its own heap *in some manner*. For example, an implementation

could simply have a fixed amount of heap space available and could throw an `OutOfMemory` exception when that space fills up. While this implementation might not win many prizes, it does qualify as a Java virtual machine. The Java virtual machine specification does not say how much memory an implementation must make available to running programs. The machine specification also does not say how an implementation must manage its heap. Rather, it only says to implementation designers that the program will be allocating memory from the heap, not freeing it. Designers must figure out how they want to deal with this situation.

No garbage collection technique is dictated by the Java virtual machine specification. Designers can use whichever techniques seem most appropriate, given their goals, constraints, and talents. Because references to objects can exist in many places—Java stacks, the heap, the method area, native method stacks—the choice of garbage collection technique heavily influences the design of an implementation's run-time data areas. Various garbage collection techniques are described in Chapter 9, "Garbage Collection."

As with the method area, the memory that makes up the heap does not need to be contiguous and can be expanded and contracted as the running program progresses. An implementation's method area could, in fact, be implemented on top of its heap. In other words, when a virtual machine needs memory for a freshly loaded class, it could take that memory from the same heap on which objects reside. The same garbage collector that frees memory occupied by unreferenced objects could take care of finding and freeing (unloading) unreferenced classes. Implementations might enable users or programmers to specify an initial size for the heap, as well as a maximum and minimum size.

Object Representation The Java virtual machine specification is silent in regards to how objects should be represented on the heap. Object representation—an integral aspect of the overall design of the heap and garbage collector—is a decision left to implementation designers.

The instance variables declared in the object's class and all of its superclasses make up the primary data that must (in some way) be represented for each object. Given an object reference, the virtual machine must have the capability to quickly locate the instance data for the object. In addition, there must be some way to access an object's class data (stored in the method area) when given a reference to the object. For this reason, the memory allocated for an object usually includes some kind of pointer to the method area.

One possible heap design divides the heap into two parts: a handle pool and an object pool. An object reference is a native pointer to a handle pool entry. A handle pool entry has two components: a pointer to instance data in the object pool, and a pointer to class data in the method area. The advantage of this scheme is that the virtual machine can easily combat heap fragmentation. When the virtual machine moves an object in the object pool, it only needs to update one pointer with the object's new address: the relevant pointer in the handle pool. The disadvantage of this approach is that every point of access to an object's instance data requires dereferencing two pointers. This approach to object representation is shown graphically in Figure 5-5. This kind of heap is demonstrated interactively by the `HeapOfFish` applet described in Chapter 9, "Garbage Collection."

Another design makes an object reference a native pointer to a bundle of data that contains the object's instance data and a pointer to the object's class data. This approach requires dereferencing only one pointer to access an object's instance data but makes moving objects more complicated. When the virtual machine moves an object to combat fragmentation of this kind of heap, it must update every reference to that object anywhere in the runtime data areas. This approach to object representation is shown graphically in Figure 5-6.

Figure 5-5 Splitting an object across a handle pool and object pool

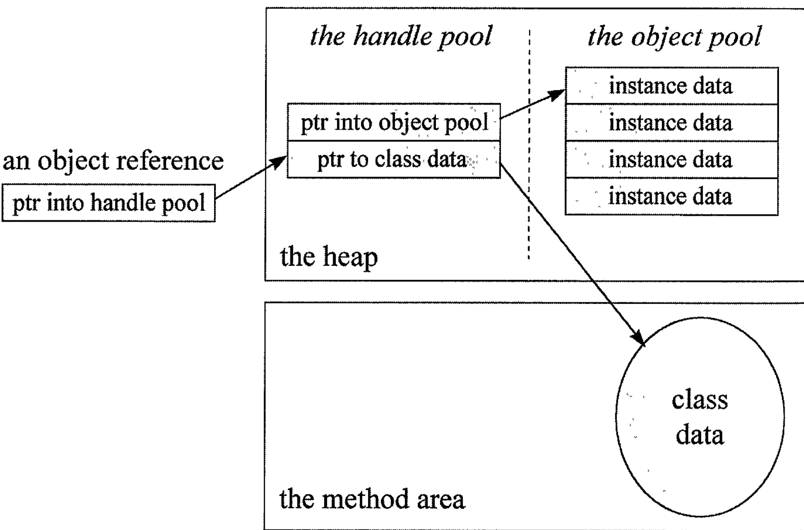
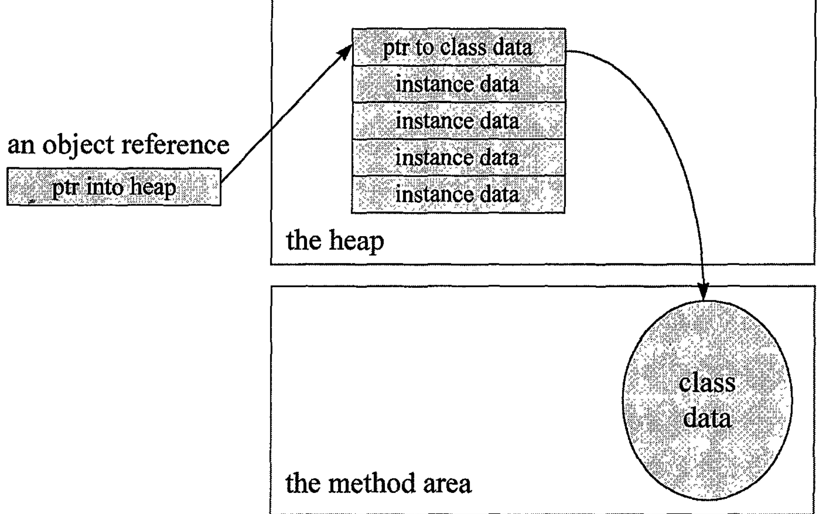


Figure 5-6 Keeping object data in one place



The virtual machine needs to get from an object reference to that object's class data for several reasons. When a running program attempts to cast an object reference to another type, the virtual machine must check to see whether the type being cast to is the actual class of the referenced object or whether it is one of its supertypes. The machine must perform the same check when a program performs an `instanceof` operation. In either case, the virtual machine must look into the class data of the referenced object. When a program invokes an instance method, the virtual machine must perform dynamic binding. In other words, the machine must choose the method to invoke based not on the type of the reference, but on the class of the object. To do this task, the machine must once again have access to the class data (given only a reference to the object).

No matter which object representation an implementation uses, a method table is probably close at hand for each object. Because method tables hasten the invocation of instance methods, they can play an important role in achieving good overall performance for a virtual machine implementation. Method tables are not required by the Java virtual machine specification and might not exist in all implementations. Implementations that have extremely low memory requirements, for instance, might not have the capacity to afford the extra memory space that method tables occupy. If an implementation does use method tables, however, an

object's method table will likely be quickly accessible when only given a reference to the object.

One way that an implementation could connect a method table to an object reference is shown graphically in Figure 5-7. This figure shows that the pointer kept with the instance data for each object points to a special structure. The special structure has two components:

- A pointer to the full class data for the object
- The method table for the object

The method table is an array of pointers to the data for each instance method that can be invoked on objects of that class. The method data pointed to by method table includes the following information:

- The sizes of the operand stack and local variables sections of the method's stack
- The method's bytecodes
- An exception table

This data gives the virtual machine enough information to invoke the method. The method table include pointers to data for methods declared explicitly in the object's class or methods inherited from superclasses. In other words, the pointers in the method table might point to methods defined in the object's class or any of its superclasses. More information about method tables is given in Chapter 8, "The Linking Model."

If you are familiar with the inner workings of C++, you might recognize the method table as being similar to the *Virtual Table* (VTBL) of C++ objects. In C++, objects are represented by their instance data plus an array of pointers to any virtual functions that can be invoked on the object. This approach could also be taken by a Java virtual machine implementation. An implementation could include a copy of the method table for a class as part of the heap image for every instance of that class. This approach would consume more heap space than the approach shown in Figure 5-7, but it might yield slightly better performance on a system that has large quantities of available memory.

One other kind of data that is not shown in Figures 5-5 and 5-6 but is logically part of an object's data on the heap is the object's *lock*. Each object in a Java virtual machine is associated with a lock (or *mutex*) that a program can use to coordinate multi-threaded access to the object. Only one thread at a time can own an object's lock. While a particular thread owns a particular object's lock, only that thread can access that object's instance variables. All other threads that attempt to access the object's

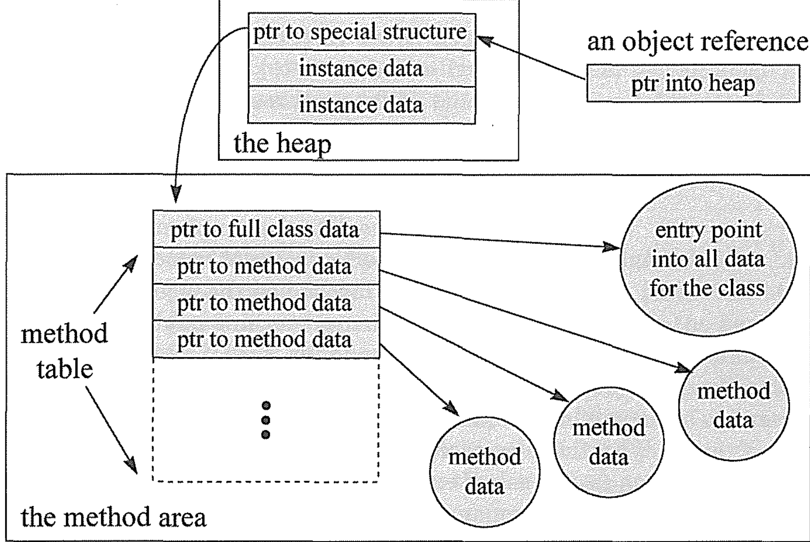


Figure 5-7
Keeping the method table close at hand

variables have to wait until the owning thread releases the object's lock. If a thread requests a lock that is already owned by another thread, the requesting thread has to wait until the owning thread releases the lock. Once a thread owns a lock, it can request the same lock multiple times, but then it has to release the lock the same number of times before the lock is made available to other threads. If a thread requests a lock three times, for example, that thread will continue to own the lock until it has released the lock three times.

Many objects will go through their entire lifetimes without ever being locked by a thread. The data required to implement an object's lock is not needed unless a thread actually requests a lock. As a result, many implementations—such as the ones shown in Figure 5-5 and 5-6—might not include a pointer to lock data within the object itself. Such implementations must create the necessary data to represent a lock when the lock is requested for the first time. In this scheme, the virtual machine must associate the lock with the object in some indirect way, such as by placing the lock data into a search tree based on the object's address.

Along with data that implements a lock, every Java object is logically associated with data that implements a *wait set*. Whereas locks help threads to work independently on shared data without interfering with one another, wait sets help threads to cooperate—to work towards a common goal.

Wait sets are used in conjunction with wait and notify methods. Every class inherits from `Object` three wait methods (overloaded forms of a method called `wait()`) and two notify methods (`notify()` and `notifyAll()`). When a thread invokes a wait method on an object, the Java virtual machine suspends that thread and adds it to that object's wait set. When a thread invokes a notify method on an object, the virtual machine will at some time wake up one or more threads from that object's wait set. As with the data that implements an object's lock, the data that implements an object's wait set is not needed unless a wait or notify method is actually invoked on the object. As a result, many implementations of the Java virtual machine might keep the wait set data separate from the actual object data. Such implementations could allocate the data needed to represent an object's wait set when a wait or notify method is first invoked on that object by the running application. For more information about locks and wait sets, see Chapter 20, "Thread Synchronization."

One last example of a type of data that can be included as part of the image of an object on the heap is any data needed by the garbage collector. The garbage collector must (in some way) keep track of which objects are referenced by the program. This task invariably requires data to be kept for each object on the heap. The kind of data required depends upon the garbage-collection technique being used. For example, if an implementation uses a *mark and sweep* algorithm, it must have the capability to mark an object as referenced or unreferenced. For each unreferenced object, it might also need to indicate whether or not the object's finalizer has been run. As with thread locks, this data can be kept separate from the object image. Some garbage-collection techniques only require this extra data while the garbage collector is actually running. A mark and sweep algorithm, for instance, could potentially use a separate bitmap for marking referenced and unreferenced objects. More detail about various garbage-collection techniques and the data that each of these techniques requires is given in Chapter 9, "Garbage Collection."

In addition to data that a garbage collector uses to distinguish between referenced and unreferenced objects, a garbage collector needs data to keep track of the objects on which it has already executed a finalizer. Garbage collectors must run the finalizer on any object whose class declares a finalizer before it reclaims the memory occupied by that object. The Java language specification states that a garbage collector will only execute an object's finalizer once, but the specification permits finalizer to resurrect the object (to make the object referenced again). When the object becomes unreferenced for a second time, the garbage collector must not finalize the object again. Because most objects will probably not have a finalizer—and

few of those will resurrect their objects—this scenario of garbage collecting the same object twice will probably be extremely rare. As a result, the data used to keep track of objects that have already been finalized, although logically part of the data associated with an object, will probably not be part of the object representation on the heap. In most cases, garbage collectors will keep this information in a separate place. Chapter 9, "Garbage Collection," gives more information about finalization.

Array Representation In Java, arrays are full-fledged objects. Like objects, arrays are always stored on the heap, and implementation designers can decide how they want to represent arrays on the heap.

Arrays have a `Class` instance associated with their class, similar to any other object. All arrays of the same dimension and type have the same class. The length of an array (or the lengths of each dimension of a multidimensional array) does not play any role in establishing the array's class. For example, an array of three `int`s has the same class as an array of 300 `int`s. The length of an array is considered part of its instance data.

The name of an array's class has one open square bracket for each dimension, plus a letter or string representing the array's type. For example, the class name for an array of `int`s is `"[I"`. The class name for a three-dimensional array of `byte`s is `"[[[B"`. The class name for a two-dimensional array of `Object`s is `"[[Ljava.lang.Object"`. The full details of this naming convention for array classes is given in Chapter 6, "The Java Class File."

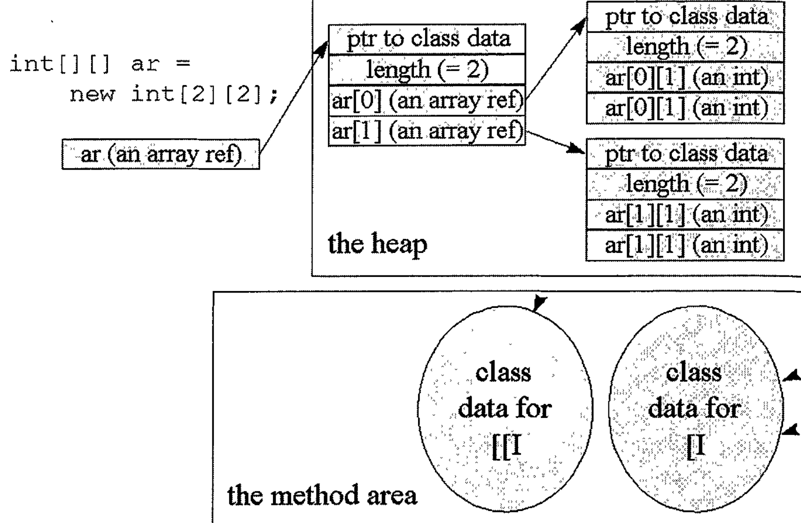
Multi-dimensional arrays are represented as arrays of arrays. A two-dimensional array of `int`s, for example, would be represented by a one-dimensional array of references to several one-dimensional arrays of `int`s. This scenario is shown graphically in Figure 5-8.

The data that must be kept on the heap for each array is the array's length, the array data, and some kind of reference to the array's class data. Given a reference to an array, the virtual machine must have the capacity to determine the array's length, to obtain and set its elements by index (checking to make sure that the array bounds are not exceeded), and to invoke any methods declared by `Object`, the direct superclass of all arrays.

The Program Counter

Each thread of a running program has its own *Program Counter* (PC) register, which is created when the thread is started. The PC register is one word in size, so it can hold both a native pointer and a `returnAddress`.

Figure 5-8
One possible heap representation for arrays



As a thread executes a Java method, the PC register contains the address of the current instruction being executed by the thread. An address can be a native pointer or an offset from the beginning of a method's bytecodes. If a thread is executing a native method, the value of the PC register is undefined.

The Java Stack

When a new thread is launched, the Java virtual machine creates a new Java stack for the thread. As mentioned earlier, a Java stack stores a thread's state in discrete frames. The Java virtual machine only performs two operations directly on Java stacks: pushing and popping frames.

The method that is currently being executed by a thread is the thread's *current method*. The stack frame for the current method is the *current frame*. The class in which the current method is defined is called the *current class*, and the current class's constant pool is the *current constant pool*. As it executes a method, the Java virtual machine keeps track of the current class and current constant pool. When the virtual machine encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame.

When a thread invokes a Java method, the virtual machine creates and pushes a new frame onto the thread's Java stack. This new frame then becomes the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data.

A method can complete in either of two ways. If a method completes by returning, it is said to have *normal completion*. If it completes by throwing an exception, it is said to have *abrupt completion*. When a method completes, whether normally or abruptly, the Java virtual machine pops and discards the method's stack frame. The frame for the previous method then becomes the current frame.

All of the data on a thread's Java stack is private to that thread. A thread has no way to access or alter the Java stack of another thread. For this reason, you never need to worry about synchronizing multi-threaded access to local variables in your Java programs. When a thread invokes a method, the method's local variables are stored in a frame on the invoking thread's Java stack. Only one thread can ever access those local variables: the thread that invoked the method.

Similar to the method area and heap, the Java stack and stack frames do not need to be contiguous in memory. Frames could be allocated on a contiguous stack, allocated on a heap, or allocated based on some combination of the two. The actual data structures used to represent the Java stack and stack frames is a decision left to implementation designers. Implementations might enable users or programmers to specify an initial size for Java stacks, as well as a maximum or minimum size.

The Stack Frame

The stack frame has three parts: local variables, operand stack, and frame data. The sizes of the local variables and operand stack, which are measured in words, depend on the needs of each individual method. These sizes are determined at compile time and are included in the class-file data for each method. The size of the frame data is implementation dependent.

When the Java virtual machine invokes a Java method, it checks the class data to determine the number of words required by the method in the local variables and operand stack. Then, the machine creates a stack frame of the proper size for the method and pushes it onto the Java stack.

Local Variables The local variables section of the Java stack frame is organized as a zero-based array of words. Instructions that use a value from the local variables section provide an index to the zero-based array.

Values of type `int`, `float`, `reference`, and `returnAddress` occupy one entry in the local variables array. Values of type `byte`, `short`, and `char` are converted to `int` before being stored in the local variables. Values of type `long` and `double` occupy two consecutive entries in the array.

To refer to a long or double value in the local variables, you will use instructions to provide the index of the first of the two consecutive entries occupied by the value. For example, if a `long` occupies array entries three and four, instructions would refer to that `long` value by index three. All values in the local variables are word aligned. Dual-entry `long`s and `double`s can start at any index.

The local variables section contains a method's parameters and local variables. Compilers place the parameters into the local variable array first in the order in which they are declared. Figure 5-9 shows the local variables section for the following two methods:

```
// On CD-ROM in file jvm/ex3/Example3a.java
class Example3a {

    public static int runClassMethod(int i, long l, float f,
        double d, Object o, byte b) {

        return 0;
    }

    public int runInstanceMethod(char c, double d, short s,
        boolean b) {

        return 0;
    }
}
```

runClassMethod()

index	type	parameter
0	int	int i
1	long	long l
3	float	float f
4	double	double d
6	reference	Object o
7	int	byte b

runInstanceMethod()

index	type	parameter
0	reference	hidden this
1	int	char c
2	double	double d
4	int	short s
5	int	boolean b

Figure 5-9 Method parameters on the local variables section of a Java stack

Figure 5-9 shows that the first parameter in the local variables for `runInstanceMethod()` is of type `reference`, although no such parameter appears in the source code. This reference is the hidden `this` reference that is passed to every instance method. Instance methods use this reference to access the instance object data upon which they were invoked. As you can see by looking at the local variables for `runClassMethod()` in Figure 5-9, class methods do not receive a hidden `this`. Class methods are not invoked on objects, so you cannot directly access a class's instance variables from a class method because there is no instance associated with the method invocation.

Also note that types `byte`, `short`, `char`, and `boolean` in the source code become `ints` in the local variables. This characteristic is also true of the operand stack. As mentioned earlier, the `boolean` type is not supported directly by the Java virtual machine. The Java compiler always uses `ints` to represent `boolean` values in the local variables or in the operand stack. Data types `byte`, `short`, and `char`, however, are supported directly by the Java virtual machine. These types can be stored on the heap as instance variables or as array elements, or in the method area as class variables. When placed into local variables or the operand stack, however, values of type `byte`, `short`, and `char` are converted into `ints`. They are manipulated as `ints` while on the stack frame, then are converted back into `byte`, `short`, or `char` types when stored back into the heap or method area.

Note that `Object o` is passed as a reference to `runClassMethod()`. In Java, all objects are passed by reference. Because all objects are stored on the heap, you will never find an image of an object in the local variables or operand stack—only object references.

Aside from a method's parameters, which compilers must place into the local variables array first and in order of declaration, Java compilers can arrange the local variables array as they wish. Compilers can place the method's local variables into the array in any order, and they can use the same array entry for more than one local variable. For example, if two local variables have limited scopes that do not overlap, such as the `i` and `j` local variables in `Example3b`, compilers are free to use the same array entry for both variables. During the first half of the method, before `j` comes into scope, entry zero could be used for `i`. During the second half of the method, after `i` has gone out of scope, entry zero could be used for `j`.

```
// On CD-ROM in file jvm/ex3/Example3b.java
class Example3b {
```

```
        public static void runtwoLoops() {

            for (int i = 0; i < 10; ++i) {
                System.out.println(i);
            }

            for (int j = 9; j >= 0; --j) {
                System.out.println(j);
            }
        }
    }
```

As with all of the other run-time memory areas, implementation designers can use whichever data structures they deem most appropriate to represent the local variables. The Java virtual machine specification does not indicate how `longs` and `doubles` should be split across the two array entries they occupy. Implementations that use a word size of 64 bits could, for example, store the entire long or double in the lower part of the two consecutive entries, leaving the higher entry unused.

Operand Stack Similar to the local variables, the operand stack is organized as an array of words. Unlike the local variables, however, which are accessed via array indices, the operand stack is accessed by pushing and popping values. If an instruction pushes a value onto the operand stack, a later instruction can pop and use that value.

The virtual machine stores the same data types in the operand stack that it stores in the local variables: `int`, `long`, `float`, `double`, `reference`, and `returnType`. The machine converts values of type `byte`, `short`, and `char` to `int` before pushing them onto the operand stack.

Other than the program counter, which cannot be directly accessed by instructions, the Java virtual machine has no registers. The Java virtual machine is stack based, rather than register based, because its instructions take their operands from the operand stack rather than from registers. Instructions can also take operands from other places, such as immediately following the opcode (the byte representing the instruction) in the bytecode stream or from the constant pool. The Java virtual machine instruction set's main focus of attention, however, is the operand stack.

The Java virtual machine uses the operand stack as a work space. Many instructions pop values from the operand stack, operate on them, and then push the result. For example, the `iadd` instruction adds two integers by popping two `ints` off the top of the operand stack, adding them, and pushing the `int` result. Here is how a Java virtual machine

would add two local variables that contain ints and would store the int result in a third local variable:

```
iload_0     // push the int in local variable 0
iload_1     // push the int in local variable 1
iadd        // pop two ints, add them, push result
istore_2    // pop int, store into local variable 2
```

In this sequence of bytecodes, the first two instructions, iload_0 and iload_1, push the ints stored in local variable positions zero and one onto the operand stack. The iadd instruction pops those two int values, adds them, and pushes the int result back onto the operand stack. The fourth instruction, istore_2, pops the result of the add off the top of the operand stack and stores it into local variable position two. In Figure 5-10, you can see a graphical depiction of the state of the local variables and operand stack while executing these instructions. In this figure, unused slots of the local variables and operand stack are left blank.

Frame Data In addition to the local variables and operand stack, the Java stack frame includes data to support constant pool resolution, normal method return, and exception dispatch. This data is stored in the *frame data* portion of the Java stack frame.

Many instructions in the Java virtual machine's instruction set refer to entries in the constant pool. Some instructions merely push constant values of type int, long, float, double, or String from the constant pool onto the operand stack. Some instructions use constant pool entries to refer to classes or arrays to instantiate, fields to access, or methods to invoke. Other instructions determine whether a particular object is a descendant of a particular class or interface specified by a constant pool entry.

Figure 5-10
Adding two local variables

		before starting	after iload_0	after iload_1	after iadd	after istore_2
local variables	0	100	100	100	100	100
	1	98	98	98	98	98
	2					198
operand stack			100	100	198	
				98		

Whenever the Java virtual machine encounters any of the instructions that refer to an entry in the constant pool, it uses the frame data's pointer to the constant pool to access that information. As mentioned earlier, references to types, fields, and methods in the constant pool are initially symbolic. When the virtual machine looks up a constant pool entry that refers to a class, interface, field, or method, that reference might still be symbolic. If so, the virtual machine must resolve the reference at that time.

Aside from constant pool resolution, the frame data must assist the virtual machine with processing a normal or abrupt method completion. If a method completes normally (by returning), the virtual machine must restore the stack frame of the invoking method. The machine must also set the PC register to point to the instruction in the invoking method that follows the instruction that invoked the completing method. If the completing method returns a value, the virtual machine must push that value onto the operand stack of the invoking method.

The frame data must also contain some kind of reference to the method's exception table, which the virtual machine uses to process any exceptions thrown during the course of execution of the method. An exception table, which is described in detail in Chapter 17, "Exceptions," defines ranges within the bytecodes of a method that are protected by catch clauses. Each entry in an exception table gives a starting and ending position of the range protected by a catch clause, an index into the constant pool that gives the exception class being caught, and a starting position of the catch clause's code.

When a method throws an exception, the Java virtual machine uses the exception table referred to by the frame data to determine how to handle the exception. If the virtual machine finds a matching catch clause in the method's exception table, it transfers control to the beginning of that catch clause. If the virtual machine does not find a matching catch clause, then the method completes abruptly. The virtual machine uses the information in the frame data to restore the invoking method's frame and then rethrows the same exception in the context of the invoking method.

In addition to data that supports constant pool resolution, normal method return, and exception dispatch, the stack frame might also include other information that is implementation dependent, such as data to support debugging.

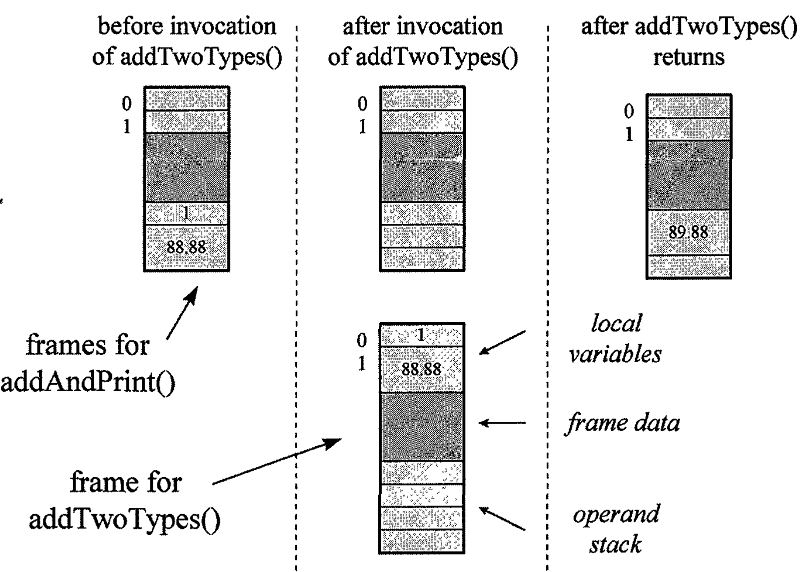
Possible Implementations of the Java Stack Implementation designers can represent the Java stack in whichever way they wish. As mentioned earlier, one potential way to implement the stack is by allocating

each frame separately from a heap. As an example, consider the following class:

```
// On CD-ROM in file jvm/ex3/Example3c.java
class Example3c {

    public static void addAndPrint() {
        double result = addTwoTypes(1, 88.88);
        System.out.println(result);
    }

    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
```

Figure 5-11 shows three snapshots of the Java stack for a thread that invokes the `addAndPrint()` method. In the implementation of the Java virtual machine represented in this figure, each frame is allocated separately from a heap. To invoke the `addTwoTypes()` method, use the `addAndPrint()` method which first pushes an `int` one and `double` 88.88 onto its operand stack, then invokes the `addTwoTypes()` method.

Figure 5-11
Allocating frames from a heap



The instruction to invoke addTwoTypes() refers to a constant pool entry. The Java virtual machine looks up the entry and resolves the entry if necessary.

Note that the addAndPrint() method uses the constant pool to identify the addTwoTypes() method, although it is part of the same class. Similar to references to fields and methods of other classes, references to the fields and methods of the *same* class are initially symbolic and must be resolved before they are used.

The resolved constant pool entry points to information in the method area about the addTwoTypes() method. The virtual machine uses this information to determine the sizes required by addTwoTypes() for the local variables and operand stack. In the class file generated by Sun's javac compiler from JDK Version 1.1, addTwoTypes() requires three words in the local variables and four words in the operand stack. (As mentioned earlier, the size of the frame data portion is implementation dependent.) The virtual machine allocates just enough memory for the addTwoTypes() frame from a heap, then pops the double and int parameters (88.88 and one) from addAndPrint()'s operand stack and places them into addTwoType()'s local variable slots one and zero.

When addTwoTypes() returns, it first pushes the double return value (in this case, 89.88) onto its operand stack. The virtual machine uses the information in the frame data to locate the stack frame of the invoking method, addAndPrint(). The machine then pushes the double return value onto addAndPrint()'s operand stack and then frees the memory occupied by addTwoType()'s frame. The virtual machine makes addAndPrint()'s frame current and continues executing the addAndPrint() method at the first instruction past the addTwoType() method invocation.

Figure 5-12 shows snapshots of the Java stack of a different virtual machine implementation executing the same methods. Instead of allocating each frame separately from a heap, this implementation allocates frames from a contiguous stack. This approach enables the implementation to overlap the frames of adjacent methods. The portion of the invoking method's operand stack that contains the parameters to the invoked method becomes the base of the invoked method's local variables. In this example, addAndPrint()'s entire operand stack becomes addTwoType()'s entire local variables section.

This approach saves memory space, because the same memory is used by the calling method to store the parameters as is used by the invoked method to access the parameters. This approach also saves time, because the Java virtual machine does not have to spend time copying the parameter values from one frame to another.

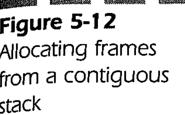
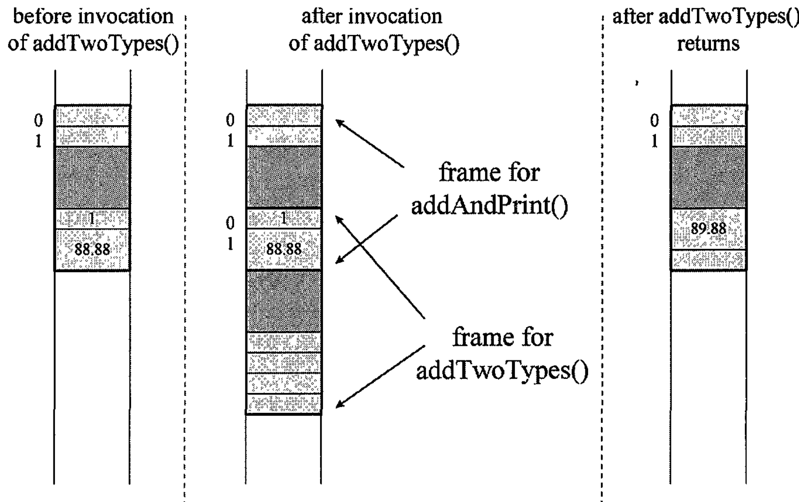


Figure 5-12 Allocating frames from a contiguous stack



Note that the operand stack of the current frame is always at the top of the Java stack. Although this situation might be easier to visualize in the contiguous memory implementation of Figure 5-12, it is true no matter how the Java stack is implemented. (As mentioned earlier, in all of the graphical images of the stack shown in this book, the stack grows downward. The top of the stack is always shown at the bottom of the picture.) Instructions that push values onto (or pop values off) the operand stack always operate on the current frame. Thus, pushing a value onto the operand stack can be seen as pushing a value onto the top of the entire Java stack. In the remainder of this book, pushing a value onto the stack refers to pushing a value onto the operand stack of the current frame.

One other possible approach to implementing the Java stack is a hybrid of the two approaches shown in Figure 5-11 and Figure 5-12. A Java virtual machine implementation can allocate a chunk of contiguous memory from a heap when a thread starts. In this memory, the virtual machine can use the overlapping-frames approach shown in Figure 5-12. If the stack outgrows the contiguous memory, the virtual machine can allocate another chunk of contiguous memory from the heap. Then, the machine can use the separate-frames approach shown in Figure 5-11 to connect the invoking method's frame sitting in the old chunk with the invoked method's frame sitting in the new chunk. Within the new chunk, it can once again use the contiguous memory approach.

Native Method Stacks

In addition to all of the runtime data areas (described previously) that are defined by the Java virtual machine specification, a running Java application can use other data areas created by or for native methods. When a thread invokes a native method, it enters a new world in which the structures and security restrictions of the Java virtual machine no longer hamper its freedom. A native method can likely access the runtime data areas of the virtual machine (depending on the native method interface), but it can also do anything else it wants. The method can use registers inside the native processor, allocate memory on any number of native heaps, or use any kind of stack.

Native methods are inherently implementation dependent. Implementation designers are free to decide which mechanisms they will use to enable a Java application running on their implementation to invoke native methods.

Any native method interface will use some kind of native method stack. When a thread invokes a Java method, the virtual machine creates a new frame and pushes it onto the Java stack. When a thread invokes a native method, however, that thread leaves the Java stack behind. Instead of pushing a new frame onto the thread's Java stack, the Java virtual machine will simply dynamically link to and directly invoke the native method. One way to think of this process is that the Java virtual machine is dynamically extending itself with native code—as if the Java virtual machine implementation is just calling another (dynamically linked) method within itself at the bequest of the running Java program.

If an implementation's native method interface uses a C-linkage model, then the native method stacks are C stacks. When a C program invokes a C function, the stack operates in a certain way. The arguments to the function are pushed onto the stack in a certain order. The return value is passed back to the invoking function in a certain way. This behavior is true for the native method stacks in that implementation.

A native method interface will more than likely have the capacity to call back into the Java virtual machine and invoke a Java method (once again, this decision is up to the designers). In this case, the thread leaves the native method stack and enters another Java stack.

Figure 5-13 shows a graphical depiction of a thread that invokes a native method that calls back into the virtual machine to invoke another Java method. This figure shows the full picture of what a thread can expect inside the Java virtual machine. A thread might spend its entire lifetime executing Java methods and working with frames on its Java

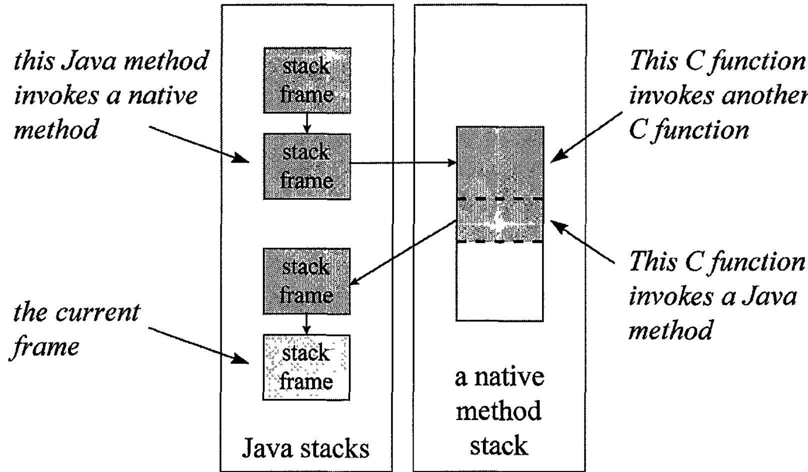


Figure 5-13
The stack for a thread that invokes Java and native methods

stack, or it might jump back and forth between the Java stack and native method stacks.

As depicted in Figure 5-13, a thread first invoked two Java methods—the second of which invoked a native method. This act caused the virtual machine to use a native method stack. In this figure, the native method stack is shown as a finite amount of contiguous memory space. Assume this item is a C stack. The stack area used by each C-linkage function is shown in gray and is bounded by a dashed line. The first C-linkage function, which was invoked as a native method, invoked another C-linkage function. The second C-linkage function invoked a Java method through the native method interface. This Java method invoked another Java method, which is the current method shown in the figure.

As with the other runtime memory areas, the memory occupied by native method stacks does not need to be a fixed size. This memory can expand and contract as needed by the running application. Implementations can enable users or programmers to specify an initial size for the method area, as well as a maximum or minimum size.

Execution Engine

At the core of any Java virtual machine implementation is its execution engine. In the Java virtual machine specification, the behavior of the

execution engine is defined in terms of an instruction set. For each instruction, the specification describes in detail *what* an implementation should do when it encounters the instruction as it executes bytecodes, but the specification says little about *how*. As mentioned in previous chapters, implementation designers are free to decide how their implementations will execute bytecodes. Their implementations can interpret, just-in-time compile, execute natively in silicon, use a combination of these techniques, or dream up some brand-new technique.

Similar to the three senses of the term *Java virtual machine* described at the beginning of this chapter, the term *execution engine* can also be used in any of three senses: an abstract specification, a concrete implementation, or a run-time instance. The abstract specification defines the behavior of an execution engine in terms of the instruction set. Concrete implementations, which can use a variety of techniques, are either software, hardware, or a combination of both. A run-time instance of an execution engine is a thread.

Each thread of a running Java application is a distinct instance of the virtual machine's execution engine. From the beginning of its lifetime to the end, a thread is either executing bytecodes or native methods. A thread can execute bytecodes directly (by interpreting or executing natively in silicon) or indirectly (by just-in-time compiling and executing the resulting native code). A Java virtual machine implementation might use other threads that are invisible to the running application, such as a thread that performs garbage collection. Such threads do not need to be instances of the implementation's execution engine. All threads that belong to the running application, however, are execution engines in action.

The Instruction Set A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte *opcode* followed by zero or more *operands*. The opcode indicates the operation to be performed. Operands supply extra information needed by the Java virtual machine to perform the operation specified by the opcode. The opcode itself indicates whether or not it is followed by operands and which form the operands take (if any). Many Java virtual machine instructions take no operands and therefore consist only of an opcode. Depending upon the opcode, the virtual machine might refer to data stored in other areas in addition to (or instead of) operands that trail the opcode. When the virtual machine executes an instruction, it might use entries in the current constant pool, entries in the current frame's local variables, or values sitting on top of the current frame's operand stack.

The abstract execution engine runs by executing bytecodes one instruction at a time. This process takes place for each thread (execution engine instance) of the application running in the Java virtual machine. An execution engine fetches an opcode, and if that opcode has operands, it fetches the operands. The engine executes the action requested by the opcode and its operands, then fetches another opcode. Execution of bytecodes continues until a thread completes, either by returning from its starting method or by not catching a thrown exception.

From time to time, the execution engine might encounter an instruction that requests a native method invocation. On such occasions, the execution engine will dutifully attempt to invoke that native method. When the native method returns (if it completes normally, not by throwing an exception), the execution engine will continue executing the next instruction in the bytecode stream.

One way you can think of native methods, therefore, is as programmer-customized extensions to the Java virtual machine's instruction set. If an instruction requests an invocation of a native method, the execution engine invokes the native method. Running the native method is how the Java virtual machine executes the instruction. When the native method returns, the virtual machine moves on to the next instruction. If the native method completes abruptly (by throwing an exception), the virtual machine follows the same steps to handle the exception as it does when any instruction throws an exception.

Part of the job of executing an instruction is determining the next instruction to execute. An execution engine determines the next opcode to fetch in one of three ways. For many instructions, the next opcode to execute directly follows the current opcode and its operands, if any, in the bytecode stream. For some instructions, such as `goto` or `return`, the execution engine determines the next opcode as part of its execution of the current instruction. If an instruction throws an exception, the execution engine determines the next opcode to fetch by searching for an appropriate catch clause.

Several instructions can throw exceptions. The `athrow` instruction, for example, throws an exception explicitly. This instruction is the compiled form of the `throw` statement in Java source code. Every time the `athrow` instruction is executed, it will throw an exception. Other instructions throw exceptions only when certain conditions are encountered. For example, if the Java virtual machine discovers (to its chagrin) that the program is attempting to divide an integer by zero, it will throw an `ArithmeticException`. This situation can occur while executing any of four instructions—`idiv`, `ldiv`, `irem`, and `lrem`—which perform divisions or calculate remainders on `int`s or `long`s.

Each type of opcode in the Java virtual machine's instruction set has a mnemonic. In the typical assembly language style, streams of Java bytecodes can be represented by their mnemonics followed by (optional) operand values.

For an example of method's bytecode stream and mnemonics, consider the doMathForever() method of this class:

```
// On CD-ROM in file jvm/ex4/Act.java
class Act {

    public static void doMathForever() {
        int i = 0;
        for (;;) {
            i += 1;
            i *= 2;
        }
    }
}
```

The stream of bytecodes for doMathForever() can be disassembled into mnemonics as follows. The Java virtual machine specification does not define any official syntax for representing the mnemonics of a method's bytecodes. The code shown as follows illustrates the manner in which streams of bytecode mnemonics will be represented in this book. The left-hand column shows the offset in bytes from the beginning of the method's bytecodes to the start of each instruction. The center column shows the instruction and any operands. The right-hand column contains comments, which are preceded with a double slash (just as in Java source code).

```
// Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
// Method void doMathForever()
// Left column: offset of instruction from beginning of method
// |    Center column: instruction mnemonic and any operands
// |    |                   Right column: comment
   0    iconst_0            // 03
   1    istore_0            // 3b
   2    iinc 0, 1           // 84 00 01
   5    iload_0             // 1a
   6    iconst_2            // 05
   7    imul                // 68
   8    istore_0            // 3b
   9    goto 2              // a7 ff f9
```

This way of representing mnemonics is similar to the output of the javap program of Sun's Java 2 SDK. javap enables you to look at the bytecode mnemonics of the methods of any class file. Note that jump

addresses are given as offsets from the beginning of the method. The `goto` instruction causes the virtual machine to jump to the instruction at offset two (an `iinc`). The actual operand in the stream is minus seven. To execute this instruction, the virtual machine adds the operand to the current contents of the PC register. The result is the address of the `iinc` instruction at offset two. To make the mnemonics easier to read, the operands for jump instructions are shown as if the addition has already taken place. Instead of saying "`goto -7`," the mnemonics say, "`goto 2`."

The central focus of the Java virtual machine's instruction set is the operand stack. Values are generally pushed onto the operand stack before they are used. Although the Java virtual machine has no registers for storing arbitrary values, each method has a set of local variables. The instruction set treats the local variables as a set of registers that are referred to by indexes. Nevertheless, other than the `iinc` instruction, which increments a local variable directly, values stored in the local variables must be moved to the operand stack before being used.

For example, to divide one local variable by another, the virtual machine must push both onto the stack, perform the division, and then store the result back in the local variables. To move the value of an array element or object field into a local variable, the virtual machine must first push the value onto the stack, then store it into the local variable. To set an array element or object field to a value stored in a local variable, the virtual machine must follow the reverse procedure. First, it must push the value of the local variable onto the stack, then pop it off the stack and into the array element or object field on the heap.

Several goals—some of which are conflicting—guided the design of the Java virtual machine's instruction set. These goals are basically the same as those described in Part I of this book as the motivation behind Java's entire architecture: platform independence, network mobility, and security.

The platform independence goal was a major influence in the design of the instruction set. The instruction set's stack-centered approach, described previously, was chosen instead of a register-centered approach to facilitate efficient implementation on architectures with few or irregular registers, such as the Intel 80X86. This feature of the instruction set—the stack-centered design—makes it easier to implement the Java virtual machine on a wide variety of host architectures.

Another motivation for Java's stack-centered instruction set is that compilers usually use a stack-based architecture to pass an intermediate compiled form or the compiled program to a linker/optimizer. The Java class file, which is in many ways similar to the Unix `.o` or Windows `.obj` file emitted by a C compiler, actually represents an intermediate, compiled

form of a Java program. In the case of Java, the virtual machine serves as a (dynamic) linker and might serve as an optimizer. The stack-centered architecture of the Java virtual machine's instruction set facilitates the optimization that can be performed at run time in conjunction with execution engines that perform just-in-time compiling or adaptive optimization.

As mentioned in Chapter 4, "Network Mobility," one major design consideration was class-file compactness. Compactness is important because it facilitates speedy transmission of class files across networks. In the bytecodes stored in class files, all instructions—except two that deal with table jumping—are aligned on byte boundaries. The total number of opcodes is small enough so that opcodes occupy only one byte. This design strategy favors class-file compactness, possibly at the cost of some performance when the program runs. In some Java virtual machine implementations, especially those executing bytecodes in silicon, the single-byte opcode might preclude certain optimizations that could improve performance. Also, better performance might have been possible on some implementations if the bytecode streams were word aligned instead of byte aligned. (An implementation could always realign bytecode streams or translate opcodes into a more efficient form as classes are loaded. Bytecodes are byte aligned in the class file and in the specification of the abstract method area and execution engine. Concrete implementations can store the loaded bytecode streams any way they wish.)

Another goal that guided the design of the instruction set was the capability for bytecode verification, especially all at once by a data-flow analyzer. The verification capability is needed as part of Java's security framework. The capability to use a data-flow analyzer on the bytecodes when they are loaded, rather than verifying each instruction as it is executed, facilitates the execution speed. One way this design goal manifests itself in the instruction set is that most opcodes indicate the type on which they operate.

For example, instead of simply having one instruction that pops a word from the operand stack and stores it in a local variable, the Java virtual machine's instruction set has two instructions. One instruction, `istore`, pops and stores an `int`. The other instruction, `fstore`, pops and stores a `float`. Both of these instructions perform the same function when executed: they pop a word and store it. Distinguishing between popping and storing an `int` versus a `float` is important only to the verification process.

For many instructions, the virtual machine needs to know the types being operated on to know how to perform the operation. For example, the Java virtual machine supports two ways of adding two words together, yielding a one-word result. One addition treats the words as `int`s, while

the other treats the words as `floats`. The difference between these two instructions facilitates verification but also tells the virtual machine whether it should perform integer or floating-point arithmetic.

A few instructions operate on any type. The `dup` instruction, for example, duplicates the top word of a stack regardless of its type. Some instructions, such as `goto`, do not operate on typed values. The majority of the instructions, however, operate on a specific type. The mnemonics for most of these typed instructions indicate their type by a single character prefix that starts their mnemonic. Table 5-2 shows the prefixes for the various types. A few instructions, such as `arraylength` or `instanceof`, do not include a prefix because their type is obvious. The `arraylength` opcode requires an array reference, and the `instanceof` opcode requires an object reference.

Values on the operand stack must be used in a manner appropriate to their type. It is illegal, for example, to push four `ints`, then add them as if they were two `longs`. Also, it is illegal to push a `float` value onto the operand stack from the local variables, then store it as an `int` in an array on the heap. Furthermore, it is illegal to push a `double` value from an object field on the heap, then store the topmost of its two words into the local variables as a value of type `reference`. The strict type rules that are enforced by Java compilers must also be enforced by Java virtual machine implementations.

Implementations must also observe rules when executing instructions that perform generic stack operations that are type independent. As mentioned previously, the `dup` instruction pushes a copy of the top word of the

Table 5-2

Type prefixes of bytecode mnemonics

Type	Code	Example	Description
`byte`	`b`	`baload`	load `byte` from array
`short`	`s`	`saload`	load `short` from array
`int`	`i`	`iaload`	load `int` from array
`long`	`l`	`laload`	load `long` from array
`char`	`c`	`caload`	load `char` from array
`float`	`f`	`faload`	load `float` from array
`double`	`d`	`daload`	load `double` from array
`reference`	`a`	`aaload`	load `reference` from array

stack (irrespective of type). This instruction can be used on any value that occupies one word—an `int`, `float`, `reference`, or `returnAddress`. Y ou cannot use `dup` when the top of the stack contains either a `long` or `double`, which are the data types that occupy two consecutive operand stack locations. A `long` or `double` sitting on the top of the operand stack can be duplicated in its entirety by the `dup2` instruction, which pushes a copy of the top two words onto the operand stack. The generic instructions cannot be used to split dual-word values.

To keep the instruction set small enough to enable each opcode to be represented by a single byte, not all operations are supported on all types. Most operations are not supported for types `byte`, `short`, and `char`. These types are converted to `int` when they are moved from the heap or method area to the stack frame. They are operated on as `int`s, then are converted back to `byte`, `short`, or `char` before being stored back into the heap or method area.

Table 5-3 shows the computation types that correspond to each storage type in the Java virtual machine. As used here, a *storage type* is the manner in which values of the type are represented on the heap. The storage type corresponds to the type of the variable in Java source code. A *computation type* is the manner in which the type is represented on the Java stack frame.

Implementations of the Java virtual machine must in some way ensure that values are operated on by instructions that are appropriate to their type. They can verify bytecodes up front as part of the class verification

Table 5-3

Storage and computation types inside the Java virtual machine

Storage Type	Minimum Bits in Heap or Method Area		Computation Type	Words in the Java Stack Frame
`byte`	8	`int`	load `byte` from array	1
`short`	16	`int`	load `short` from array	1
`int`	32	`int`	load `int` from array	1
`long`	64	`long`	load `long` from array	2
`char`	16	`int`	load `char` from array	1
`float`	32	`float`	load `float` from array	1
`double`	64	`double`	load `double` from array	2
`reference`	32	`reference`	load `reference` from array	1

process or on the fly as the program executes, or they can use some combination of both. Bytecode verification is described in more detail in Chapter 7, "The Lifetime of a Type." The entire instruction set is covered in detail in Chapters 10 through 20.

Execution Techniques Various execution techniques that can be used by an implementation—interpreting, just-in-time compiling, adaptive optimization, and native execution in silicon—were described in Chapter 1, "Introduction to Java's Architecture." The main point to remember about execution techniques is that an implementation can use any technique to execute bytecodes, as long as it adheres to the semantics of the Java virtual machine instruction set.

One of the most interesting and speedy execution techniques is adaptive optimization. The adaptive optimization technique, which is used by several existing Java virtual machine implementations (including Sun's Hotspot virtual machine) borrows from techniques used by earlier virtual machine implementations. The original JVMs interpreted bytecodes one at a time. Second-generation JVMs added a JIT compiler, which compiles each method to native code upon first execution, then executes the native code. Thereafter, whenever the method is called, the native code is executed. Adaptive optimizers, taking advantage of information available only at run time, attempt to combine bytecode interpretation and compilation to native in the way that will yield optimum performance.

An adaptive optimizing virtual machine begins by interpreting all code, but it monitors the execution of that code. Most programs spend 80 to 90 percent of their time executing 10 to 20 percent of the code. By monitoring the program execution, the virtual machine can figure out which methods represent the program's hot spot—the 10 to 20 percent of the code that is executed 80 to 90 percent of the time.

When the adaptive optimizing virtual machine decides that a particular method is in the hot spot, it fires a background thread that compiles those bytecodes to native and heavily optimizes the native code. Meanwhile, the program can still execute that method by interpreting its bytecodes. Because the program is not held up and because the virtual machine is only compiling and optimizing the hot spot (perhaps 10 to 20 percent of the code), the virtual machine has more time than a traditional JIT to perform optimizations.

The adaptive optimization approach yields a program in which the code that is executed 80 to 90 percent of the time is native code (as heavily optimized as statically compiled C++, with a memory footprint not much

bigger than a fully interpreted Java program). In other words, this program is fast. An adaptive optimizing virtual machine can keep the old bytecodes around in case a method moves out of the hot spot. (The hot spot might move somewhat as the program executes.) If a method moves out of the hot spot, the virtual machine can discard the compiled code and revert to interpreting that method's bytecodes.

As you might have noticed, an adaptive optimizer's approach to making Java programs run fast is similar to the approach that programmers should take to improve a program's performance. An adaptive optimizing virtual machine, unlike a regular JIT-compiling virtual machine, does not carry out premature optimization. The adaptive optimizing virtual machine begins by interpreting bytecodes. As the program runs, the virtual machine profiles the program to find the program's hot spot, which means the 10 to 20 percent of the code that is executed 80 to 90 percent of the time. Like a good programmer, the adaptive optimizing virtual machine just focuses its optimization efforts on that time-critical code.

There is a bit more to the adaptive optimization story, however. Adaptive optimizers can be tuned for the run-time characteristics of Java programs—in particular, of well-designed Java programs. According to David Griswold, Hotspot manager at JavaSoft, "Java is a lot more object-oriented than C++. You can measure that; you can look at the rates of method invocations, dynamic dispatches, and such things. And the rates [for Java] are much higher than they are in C++." Now, this high rate of method invocations and dynamic dispatches is especially prominent in a well-designed Java program, because one aspect of a well-designed Java program is highly factored, fine-grained design—in other words, lots of compact, cohesive methods and objects.

This run-time characteristic of Java programs—the high frequency of method invocations and dynamic dispatches—affects performance in two ways. First, there is an overhead associated with each dynamic dispatch. Second (and more significantly), method invocations reduce the effectiveness of compiler optimization.

Method invocations reduce the effectiveness of optimizers, because optimizers do not perform well across method-invocation boundaries. As a result, optimizers end up focusing on the code between method invocations. The greater the method invocation frequency, the fewer amount of code the optimizer has to work with between method invocations, and the less effective the optimization becomes.

The standard solution to this problem is inlining—the copying of an invoked method's body directly into the body of the invoking method. Inlining eliminates method calls and gives the optimizer more code with

which to work, making possible more effective optimization at the cost of increasing the run-time memory footprint of the program.

The trouble is that inlining is harder with object-oriented languages such as Java and C++ than with non-object-oriented languages such as C, because object-oriented languages use dynamic dispatching. The problem is worse in Java than in C++, because Java has a greater call frequency and a greater percentage of dynamic dispatches than C++.

A regular optimizing static compiler for a C program can inline in a straight-forward manner, because there is one function implementation for each function call. The trouble with inlining in object-oriented languages is that dynamic method dispatch means that there might be multiple function (or method) implementation for any given function call. In other words, the Java virtual machine might have many different implementations of a method to choose from at run time, based on the class of the object on which the method is being invoked.

One solution to the problem of inlining a dynamically dispatched method call is to just inline all of the method implementations that might be selected at run time. The trouble with this solution is that in cases where there are many method implementations, the size of the optimized code can grow large.

One advantage that adaptive optimization has over static compilation is that because it happens at run time, it can use information that is not available to a static compiler. For example, although there might be 30 possible implementations that are called for a particular method invocation, perhaps only two of them are ever called at run time. The adaptive optimization approach enables only those two to be inlined, thereby minimizing the size of the optimized code.

Threads The Java virtual machine specification defines a threading model that aims to facilitate implementation on a wide variety of architectures. One goal of the Java threading model is to enable implementation designers, where possible and appropriate, to use native threads. Alternatively, designers can implement a thread mechanism as part of their virtual machine implementation. One advantage to using native threads on a multi-processor host is that different threads of a Java application can run simultaneously on different processors.

One tradeoff of Java's threading model is that the specification of priorities is the lowest common denominator. A Java thread can run at any one of 10 priorities. Priority one is the lowest, and priority 10 is the highest. If designers use native threads, they can map the 10 Java priorities onto the native priorities in whatever manner seems most appropriate.

The Java virtual machine specification defines the behavior of threads at different priorities only by indicating that all threads at the highest priority will receive some CPU time. Threads at lower priorities are guaranteed to receive CPU time only when all higher-priority threads are blocked. Lower-priority threads *might* receive some CPU time when higher-priority threads are not blocked, but there are no guarantees.

The specification does not assume time-slicing between threads of different priorities, because not all architectures time-slice. (As used here, *time-slicing* means that all threads at all priorities will be guaranteed some CPU time, even when no threads are blocked.) Even among those architectures that time-slice, the algorithms used to allot time slots to threads at various priorities can differ greatly.

As mentioned in Chapter 2, "Platform Independence," you must not rely on time-slicing for program correctness. You should use thread priorities only to give the Java virtual machine hints at the tasks on which it should spend more time. To coordinate the activities of multiple threads, you should use synchronization.

The thread implementation of any Java virtual machine must support two aspects of *synchronization*: object locking and thread wait and notify. Object locking helps keep threads from interfering with one another while working independently on shared data. Thread wait and notify helps threads cooperate with one another while working together toward some common goal. Running applications access the Java virtual machine's locking capabilities via the instruction set and access its wait and notify capabilities via the `wait()`, `notify()`, and `notifyAll()` methods of class `Object`. For more details, see Chapter 20, "Thread Synchronization."

In the Java virtual machine specification, the behavior of Java threads is defined in terms of *variables*, a *main memory*, and *working memories*. Each Java virtual machine instance has a main memory, which contains all of the program's variables (instance variables of objects, components of arrays, and class variables). Each thread has a working memory in which the thread stores working copies of variables that it uses or assigns. Local variables and parameters, because they are private to individual threads, can be logically seen as part of either the working memory or the main memory.

The Java virtual machine specification defines many rules that govern the low-level interactions of threads with main memory. For example, one rule states that all operations on primitive types, except in some cases `longs` and `doubles`, are *atomic*. For example, if two threads compete to write two different values to an `int` variable, even in the absence

of synchronization, the variable will end up with one value or the other. The variable will not contain a corrupted value. In other words, one thread will win the competition and will write its value to the variable first. The losing thread does not need to sulk, however, because it will write its value to the variable second, overwriting the winning thread's value.

The exception to this rule is any `long` or `double` variable that is not declared `volatile`. Rather than being treated as a single, atomic, 64-bit value, such variables can be treated by some implementations as two atomic, 32-bit values. Storing a non-volatile `long` to memory, for example, could involve two 32-bit write operations. This non-atomic treatment of `long`s and `double`s means that two threads competing to write two different values to a `long` or `double` variable can legally yield a corrupted result.

Although implementation designers are not required to treat operations involving non-volatile `long`s and `double`s atomically, the Java virtual machine specification encourages them to do so anyway. This non-atomic treatment of `long`s and `double`s is an exception to the general rule that operations on primitive types are atomic. This exception was created with the intention of facilitating efficient implementation of the threading model on processors that do not provide efficient ways to transfer 64-bit values to and from memory. In the future, this exception might be eliminated. For the time being, however, Java programmers must be sure to synchronize access to shared `long`s and `double`s.

Fundamentally, the rules governing low-level thread behavior specify when a thread can and must complete the following actions:

1. Copying values of variables from the main memory to its working memory
2. Writing values from its working memory back into the main memory

For certain conditions, the rules specify a precise and predictable order of memory reads and writes. For other conditions, however, the rules do not specify any order. The rules are designed to enable Java programmers to build multi-threaded programs that exhibit predictable behavior while giving implementation designers some flexibility. This flexibility enables designers of Java virtual machine implementations to take advantage of standard hardware and software techniques that can improve the performance of multi-threaded applications.

The fundamental, high-level implication of all of the low-level rules that govern the behavior of threads is as follows: If access to certain variables is not synchronized, threads are enabled to update those variables

in main memory in any order. Without synchronization, your multi-threaded applications might exhibit surprising behavior on some Java virtual machine implementations. With proper use of synchronization, however, you can create multi-threaded Java applications that behave in a predictable way on any implementation of the Java virtual machine.

Native Method Interface

Java virtual machine implementations are not required to support any particular native method interface. Some implementations might support no native method interfaces at all. Others might support several interfaces, each geared towards a different purpose.

Sun's JNI is geared towards portability. JNI is designed so that it can be supported by any implementation of the Java virtual machine, no matter which garbage-collection technique or object representation the implementation uses. In turn, this feature enables developers to link the same (JNI-compatible) native method binaries to any JNI-supporting virtual machine implementation on a particular host platform.

Implementation designers can choose to create proprietary native method interfaces in addition to (or instead of) JNI. To achieve its portability, JNI uses indirection through pointers to pointers and pointers to functions. To obtain the ultimate in performance, designers of an implementation might decide to offer their own low-level native method interface that is tied closely to the structure of their particular implementation. Designers could also decide to offer a higher-level native method interface than JNI, such as an interface that brings Java objects into a component software model.

To do useful work, a native method must have the capacity to interact (to some degree) with the internal state of the Java virtual machine instance. For example, a native method interface might enable native methods to do some or all of the following actions:

- Passing and returning data
- Accessing instance variables or invoking methods in objects on the garbage-collected heap
- Accessing class variables or invoking class methods
- Accessing arrays
- Locking an object on the heap for exclusive use by the current thread

- Creating new objects on the garbage-collected heap
- Loading new classes
- Throwing new exceptions
- Catching exceptions thrown by Java methods that the native method invoked
- Catching asynchronous exceptions thrown by the virtual machine
- Indicating to the garbage collector that it no longer needs to use a particular object

Designing a native method interface that offers these services can be complicated. The design needs to ensure that the garbage collector does not free any objects that are being used by native methods. If an implementation's garbage collector moves objects to keep heap fragmentation at a minimum, the native method interface design must make sure that the following situations can occur:

1. An object can be moved after its reference has been passed to a native method
2. Any objects whose references have been passed to a native method are pinned until the native method returns or otherwise indicates that it is finished with the objects

As you can see, native method interfaces are intertwined with the inner workings of a Java virtual machine.

The Real Machine

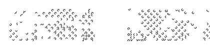
As mentioned at the beginning of this chapter, all of the subsystems, runtime data areas, and internal behaviors defined by the Java virtual machine specification are abstract. Designers are not required to organize their implementations around real components that map closely to the abstract components of the specification. The abstract internal components and behaviors are merely a vocabulary with which the specification defines the required external behavior of any Java virtual machine implementation.

In other words, an implementation can be anything on the inside as long as it behaves like a Java virtual machine on the outside. Implementations must have the capability to recognize Java class files and must adhere to the semantics of the Java code that the class files contain. Otherwise,

anything goes. How bytecodes are executed, how the runtime data areas are organized, how garbage collection is accomplished, how threads are implemented, how the bootstrap class loader finds classes, which native method interfaces are supported—these are some of the many decisions left to implementation designers.

The flexibility of the specification gives designers the freedom to tailor their implementations to fit their circumstances. In some implementations, minimizing usage of resources can be critical. In other implementations where resources are plentiful, maximizing performance might be the one and only goal.

By clearly marking the line between the external behavior and the internal implementation of a Java virtual machine, the specification preserves compatibility among all implementations while promoting innovation. Designers are encouraged to apply their talents and creativity towards building even better Java virtual machines.



Eternal Math: A Simulation

The CD-ROM contains several simulation applets that serve as interactive illustrations for the material presented in this book. The applet shown in Figure 5-14 simulates a Java virtual machine executing a few bytecodes. You can run this applet by loading `applets/EternalMath.html` from the CD-ROM into any Java-enabled Web browser or applet viewer that supports JDK Version 1.0.

The simulation instructions represent the body of the `doMathForever()` method of class `Act`, shown previously in the Instruction Set section of this chapter. This simulation shows the local variables and operand stack of the current frame, the PC register, and the bytecodes in the method area. This simulation also shows an optop register, which you can think of as part of the frame data of this particular implementation of the Java virtual machine. The optop register always points to one word beyond the top of the operand stack.

The applet has four buttons: Step, Reset, Run, and Stop. Each time you press the Step button, the Java virtual machine simulator will execute the instruction pointed to by the PC register. Initially, the PC register points to an `iconst_0` instruction. The first time you press the Step button, therefore, the virtual machine will execute `iconst_0` and will push a zero onto the stack and set the PC register to point to the next instruction to execute. Subsequent presses of the Step button will execute subsequent

instructions, and the PC register will lead the way. If you press the Run button, the simulation will continue with no further coaxing on your part until you press the Stop button. To start the simulation again, press the Reset button.

The value of each register (PC and optop) is shown two ways. The contents of each register, an integer offset from the beginning of either the method's bytecodes or the operand stack, is shown in an edit box. Also, a small arrow (either `pc>` or `optop>`) indicates the location contained in the register.

In the simulation, the operand stack is shown growing down the panel (up in memory offsets) as words are pushed onto the stack. The top of the stack recedes up the panel as words are popped from the stack.

The `doMathForever()` method only has one local variable, `i`, which sits at array position zero. The first two instructions, `iconst_0` and `istore_0`, initialize the local variable to zero. The next instruction, `iinc`, increments `i` by one. This instruction implements the `i += 1` statement from `doMathForever()`. The next instruction, `iload_0`, pushes the value of the local variable onto the operand stack. `iconst_2` pushes an `int` 2 onto the operand stack. `imul` pops the top two `ints` from the operand stack, multiplies them, and pushes the result. The `istore_0` instruction pops the result of the multiply and puts it into the local variable. The previous four instructions implement the `i *= 2` statement from `doMathForever()`. The last instruction, `goto`, sends the program counter back to the `iinc` instruction. The `goto` implements the `for (;;)` loop of `doMathForever()`.

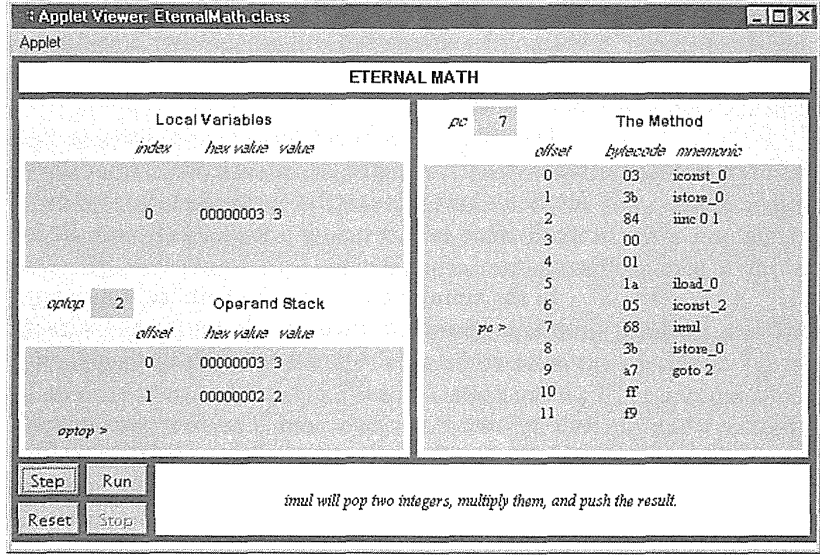
With enough patience and clicks of the Step button (or a long enough run of the Run button), you can receive an arithmetic overflow. When the Java virtual machine encounters such a condition, it simply truncates (as shown by this simulation). The machine does not throw any exceptions.

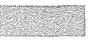
For each step of the simulation, a panel at the bottom of the applet contains an explanation of what the next instruction will do (see Figure 5-14).

On the CD-ROM

The CD-ROM contains the source code examples from this chapter in the `jvm` directory. The *Eternal Math* applet is contained on a Web page on the CD-ROM in file `applets/EternalMath.html`. The source code for this applet is found alongside its class files in the `applets/JVMSimulators` and `applets/JVMSimulators/COM/artima/jvmsim` directories.

Figure 5-14
The Eternal Math applet





The Resources Page

For links to more information about the Java virtual machine, visit the resources page at `http://www.artima.com/insidejvm/resources/`.

CHAPTER 6

The Java Class File

The previous chapter (the first of Part II), "Java Internals," gave an overview of the Java virtual machine. The next four chapters will focus on different aspects of the Java virtual machine. This chapter takes a look at the Java class file and describes the contents of the class file, including the structure and format of the constant pool. This chapter serves as a complete reference for the Java class file format.

Accompanying this chapter on the CD-ROM is an applet that interactively illustrates the material presented in the chapter. The applet, called `Getting Loaded`, simulates the Java virtual machine loading a Java class file. At the end of this chapter, you will find a description of this applet and instructions on how to use the application.

What Is a Java Class File?

The Java class file is a precisely defined binary file format for Java programs. Each Java class file represents a complete description of one Java class or interface. There is no way to put more than one class or interface into a single class file. The precise definition of the class file format ensures that any Java class file can be loaded and correctly interpreted by any Java virtual machine, no matter which system produced the class file or which system hosts the virtual machine.

Although the class file is related to the Java language architecturally, it is not inextricably linked to the Java language. As shown in Figure 6-1, you could write programs in other languages and compile them to class files, or you could compile your Java programs to a different binary file format. You can, in fact, express valid programs in Java class file form that are impossible to express in Java source code. Nevertheless, most

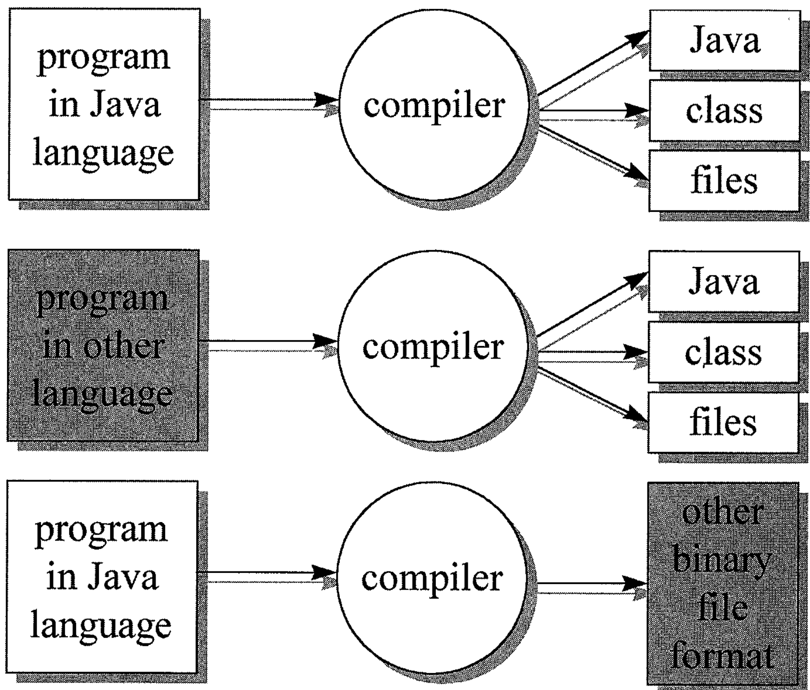


Figure 6-1 The non-exclusive relationship of the Java language and class file

Java programmers will likely use the class file as the primary vehicle for delivering their programs to Java virtual machines.

As mentioned in earlier chapters, the Java class file is a binary stream of 8-bit bytes. Data items are stored sequentially in the class file, with no padding between adjacent items. The lack of padding helps keep class files compact. Items that occupy more than one byte are split into several consecutive bytes that appear in *big-endian* (higher bytes first) order.

Just as your Java classes can contain varying numbers of fields, methods, method parameters, local variables, and so on, the Java class file can contain many items that vary in size or number from one class file to another. In the class file, the size or length of a variable-length item precedes the actual data for the item. This feature enables class file streams to be parsed from beginning to end, reading the size of an item first, followed by the item data.

What Is in a Class File?

The Java class file contains everything a Java virtual machine needs to know about one Java class or interface. The remainder of this chapter describes the class file format using tables. Each table has a name and shows an ordered list of items that can appear in a class file. Items appear in the table in the order in which they appear in the class file. Each item has a type, a name, and a count. The type is either a table name or one of the "primitive types" shown in Table 6-1. All values stored in items of type `u2`, `u4`, and `u8` appear in the class file in big-endian order.

The major components of the class file, in their order of appearance in the class file, are shown in Table 6-2 as items in the variable-length `ClassFile` table.

Table 6-1

Class file "primitive types"

u1	a single, unsigned byte
u2	two unsigned bytes
u4	four unsigned bytes
u8	eight unsigned bytes

Table 6-2

Format of a `ClassFile` Table

Type	Name	Count
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count 1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

The items of the `ClassFile` table are as follows:

`magic`

The first four bytes of every Java class file are its *magic number*, 0xCAFEBABE. The magic number makes non-Java class files easier to identify. If a file does not start with 0xCAFEBABE, it definitely is not a Java class file. A magic number can be chosen by a file format's designers to be any arbitrary number that is not already in widespread use. The magic number for the Java class file was chosen back in the days when "Java" was called "Oak." According to Patrick Naughton, a key member of the original Java team, the magic number was chosen "long before the name Java was ever uttered in reference to this language. We were looking for something fun, unique, and easy to remember. It is only a coincidence that 0xCAFEBABE, an oblique reference to the cute baristas at Peet's Coffee, was foreshadowing for the name Java."

`minor_version` and `major_version`

The second four bytes of the class file contain the minor and major version numbers. As Java technology evolves, new features may occasionally be added to the Java class file format. Each time the class file format changes, the version numbers will change, as well. To the Java virtual machine, the version numbers identify the format to which a particular class file adheres. Java virtual machines will generally be able to load class files with a given major version number and a range of minor version numbers. Java virtual machines must reject class files with version numbers outside their valid range.

The Java virtual machine implementation in Sun's JDK release 1.0.2 supports class file format versions 45.0 (the major version number is 45, while the minor version number is 0) through 45.3. The virtual machines in all 1.1 releases of the JDK can support class file format versions 45.0 through 45.65535. In the 1.2 SDK from Sun, the virtual machine can support versions 45.0 through 46.0.

1.0 or 1.1 compilers should generate class files with version number 45.3. The `javac` compiler in Sun's 1.2 SDK, by default, also generates class files with version 45.3. But if `-target 1.2` is specified on the `javac` command line, the 1.2 compiler will generate class files with version 46.0. Class files created with the `-target 1.2` flag will not run on 1.0 or 1.1 virtual machines.

The second edition of the Java virtual machine specification altered the interpretation of the major and minor version numbers of the class file. According to the second edition, the major version number of a class file is intended to correspond to a major release of the Java platform. (For example, with the release of the Java 2 Platform, the major version number was increased from 45 to 46.) The minor version numbers are intended to correspond to individual releases of a particular major platform release. Thus, although a difference in class file format will definitely be identifiable via a difference in version number, a difference in version number does not necessarily indicate a difference in class file format. Rather, a difference in version number may indicate only that the class file was generated by or is intended for a different release of the Java Platform—although the class file format has not changed.

`constant_pool_count` and `constant_pool`

Following the magic and version numbers in the class file is the *constant pool*. As mentioned in Chapter 5, "The Java Virtual Machine," the constant pool contains the constants associated with the class or interface defined by the file. Constants such as literal strings, final variable values, class names, and method names are stored in the constant pool. The constant pool is organized as a list of entries. A count of the number of entries

in the list, constant_pool_count, precedes the actual list, constant_pool.

Many entries in the constant pool refer to other entries in the constant pool, and many items that follow the constant pool in the class file refer back to entries in the constant pool. Throughout the class file, constant pool entries are referred to by the integer index that indicates their position in the constant_pool list. The first entry in the list has an index of one, the second has an index of two, and so on. Although there is no entry in the constant_pool list that has an index of zero, the missing 0th entry is included in the constant_pool_count. For example, if a constant_pool list includes 14 entries (with indexes one through 14), the constant_pool_count would be 15.

Each constant pool entry starts with a one-byte tag that indicates the type of constant making its home at that position in the list. Once a Java virtual machine grabs and interprets this tag, it knows what to expect after the tag. Table 6-3 shows the names and values of the constant pool tags.

For each tag shown in Table 6-3, there is a corresponding table. The name of the table is formed by appending "_info" to the tag name. For

Table 6-3

Constant pool tags

Entry Type	Tag Value	Description
CONSTANT_Utf8	1	A UTF-8 encoded Unicode string
CONSTANT_Integer	3	An int literal value
CONSTANT_Float	4	A float literal value
CONSTANT_Long	5	A long literal value
CONSTANT_Double	6	A double literal value
CONSTANT_Class	7	A symbolic reference to a class or interface
CONSTANT_String	8	A String literal value
CONSTANT_Fieldref	9	A symbolic reference to a field
CONSTANT_Methodref	10	A symbolic reference to a method declared in a class
CONSTANT_InterfaceMethodref	11	A symbolic reference to a method declared in an interface
CONSTANT_NameAndType	12	Part of a symbolic reference to a field or method

example, the table that corresponds to the `CONSTANT_Class` tag is called `CONSTANT_Class_info`. The `CONSTANT_Utf8_info` table stores a compressed form of unicode strings. The tables for the various kinds of constant pool entries are described in detail later in this chapter.

The constant pool plays an important role in the dynamic linking of Java programs. In addition to literal constant values, the constant pool contains the following kinds of symbolic references:

- Fully qualified names of classes and interfaces
- Field names and descriptors
- Method names and descriptors

A *field* is an instance or class variable of the class or interface. A *field descriptor* is a string that indicates the field's type. A *method descriptor* is a string that indicates the method's return type and the number, order, and types of its parameters. The constant pool's fully qualified names and method and field descriptors are used at run time to link code in this class or interface with code and data in other classes and interfaces. The class file contains no information about the eventual memory layout of its components, so classes, fields, and methods cannot be referenced directly by the bytecodes in the class file. The Java virtual machine resolves the actual address of any referenced item at run time, given a symbolic reference from the constant pool. For example, bytecode instructions that invoke a method give constant pool index of a symbolic reference to the method to invoke. This process of using the symbolic references in the constant pool is described in more detail in Chapter 8, "The Linking Model."

`access_flags`

The first two bytes after the constant pool, the *access flags*, reveal several pieces of information about the class or interface defined in the file. To start with, the access flags indicate whether the file defines a class or an interface. The access flags also indicate which modifiers were used in the declaration of the class or interface. Classes and interfaces can be public or abstract. Classes can be final, but final classes cannot be abstract. Interfaces cannot be final. The bits used for the various flags are shown in Table 6-4.

The `ACC_SUPER` flag exists for backwards compatibility with Sun's older Java compilers. In Sun's older Java virtual machines, the `invokespecial` instruction had more relaxed semantics. All new compilers should set the `ACC_SUPER` flag. All new implementations of the Java virtual machine should implement the newer, stricter `invokespecial` semantics. (See the `invokespecial` instruction in Appendix A for a description of these

Table 6-4

Flag bits in the access_flags item of ClassFile tables

Flag Name	Value	Meaning If Set	Set By
ACC_PUBLIC	0x0001	Type is public	Classes and interfaces
ACC_FINAL	0x0010	Class is final	Classes only
ACC_SUPER	0x0020	Use new invokespecial semantics	Classes and interfaces
ACC_INTERFACE	0x0200	Type is an interface, not a class	All interfaces, no classes
ACC_ABSTRACT	0x0400	Type is abstract	All interfaces, some classes

semantics.) Sun's older compilers generate class files with the ACC_SUPER flag set to zero. Sun's older Java virtual machines ignore the flag if it is set.

All unused bits in access_flags must be set to zero by compilers and ignored by Java virtual machine implementations.

this_class

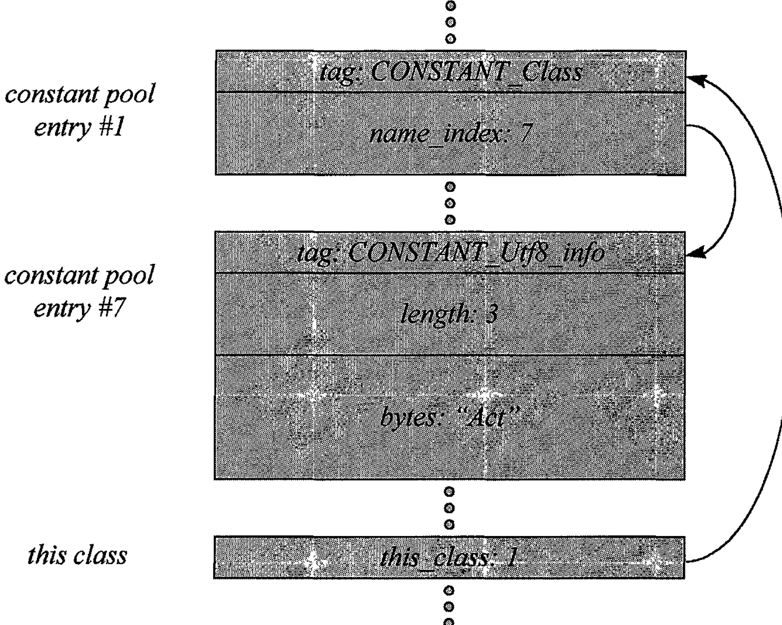
The next two bytes are the this_class item, an index into the constant pool. The constant pool entry at position this_class must be a CONSTANT_Class_info table, which has two parts: a tag and a name_index. The tag will have the value CONSTANT_Class. The constant pool entry at position name_index will be a CONSTANT_Utf8_info table containing the fully qualified name of the class or interface.

The this_class item provides a glimpse of how the constant pool is used. By itself, the this_class item is just an index into the constant pool. When a Java virtual machine looks up the constant pool entry at the position this_class, it will find an entry that identifies itself via its tag as a CONSTANT_Class_info. The Java virtual machine knows that CONSTANT_Class_info entries always have an index into the constant pool, called name_index, following their tag. So the virtual machine looks up the constant pool entry at position name_index, where it should find a CONSTANT_Utf8_info entry that contains the fully qualified name of the class or interface. See Figure 6-2 for a graphical depiction of this process.

super_class

Following this_class in the class file is the super_class item, another two-byte index into the constant pool. The constant pool entry at position super_class will be a CONSTANT_Class_info entry that

Figure 6-2
Example of constant pool usage



refers to the fully qualified name of this class's superclass. Because the base class of every object in Java programs is the `java.lang.Object` class, the `super_class` constant pool index will be valid for every class except `Object`. For `Object`, `super_class` is a zero. For interfaces, the constant pool entry at position `super_class` is `java.lang.Object`.

`interfaces_count` and `interfaces`

The component that follows `super_class` starts with `interfaces_count`, a count of the number of superinterfaces directly implemented by the class or extended by the interface defined in this file. Immediately following the count is `interfaces`, an array that contains one index into the constant pool for each superinterface directly implemented by this class or interface. Each superinterface is represented by a `CONSTANT_Class_info` entry in the constant pool that refers to the fully qualified name of the interface. Only direct superinterfaces, those that appear in the `implements` clause of the class or the `extends` clause of the interface declaration, appear in this array. The superinterfaces appear in the array in the order in which they appear (left to right) in the `implements` or `extends` clause.

fields_count and fields

Following the interfaces component in the class file is a description of the fields declared by this class or interface. This component starts with fields_count, a count of the number of fields that includes both class and instance variables. Following the count is a list of variable-length field_info tables, one for each field. (The fields_count indicates the number of field_info tables in the list.) The only fields that appear in the fields list are those that are declared by the class or interface defined in the file. No fields inherited from superclasses or superinterfaces appear in the fields list. On the other hand, the fields list could include fields not mentioned in a corresponding Java source file, because Java compilers may add fields to classes or interfaces during compilation. For example, to the fields list of an inner class, the Java compiler adds instance variables to hold references to each enclosing class instance. Any fields in the fields list that were not mentioned in the source, but were instead added by the compiler, should be marked with a Synthetic attribute.

Each field_info table reveals information about one field. The table contains the field's name, descriptor, and modifiers. If the field is declared as final, the field_info table also reveals the field's constant value. Some of this information is contained in the field_info table itself, and some is contained in constant pool locations referred to by the table. The field_info table is described in more detail later in this chapter.

methods_count and methods

Following the fields in the class file is a description of the methods which are declared by the class or interface. This component starts with methods_count, a two-byte count of the number of methods in the class or interface. The count includes only those methods that are explicitly defined by this class or interface. (The count does not include any methods inherited from superclasses or superinterfaces.) Following the method count are the methods themselves, described in a list of method_info tables. (The methods_count indicates the number of method_info tables in the list.)

The method_info table contains several pieces of information about the method, including the method's name and descriptor (its return type and argument types). If the method is not abstract and not native, the method_info table includes the number of stack words required for the method's local variables, the maximum number of stack words required for the method's operand stack, a table of exceptions caught by the method, the bytecode sequence, and optional line number and local variable tables. If the method can throw any checked exceptions,

the method_info table includes a list of those checked exceptions. The method_info table is described in detail later in this chapter.

attributes_count and attributes

The last component in the class file are the attributes, which give general information about the particular class or interface defined by the file. The attributes component starts with attributes_count, a count of the number of attribute_info tables appearing in the subsequent attributes list. The first item in each attribute_info table is an index into the constant pool of a CONSTANT_Utf8_info table that gives the attribute's name.

Attributes come in many varieties. Several varieties are defined by the Java virtual machine specification, but anyone can create their own varieties of attributes (following certain rules) and place them into class files. Java virtual machine implementations must silently ignore any attributes they do not recognize. The rules surrounding the creation of new varieties of attributes are described later in this chapter.

Attributes appear in several places in the class file, not just in the attributes item of the top-level ClassFile table. The attributes that appear in the ClassFile table give more information about the class or interface define by the file. Attributes that give more information about a field may be included as part of field_info table. Attributes that give more information about a method may be included as part of a method_info table.

The Java virtual machine specification defines two kinds of attributes that may appear in the attributes list of the ClassFile table: SourceCode and InnerClasses. These two attributes are described in detail later in this chapter.

Special Strings

The symbolic references contained in the constant pool involve three special kinds of strings: fully qualified names, simple names, and descriptors. All symbolic references include the fully qualified name of a class or interface. Symbolic references to fields include a simple field name and field descriptor, in addition to a fully qualified type name. Symbolic references to methods include a simple method name and method descriptor, in addition to a fully qualified type name.

The same special strings that are used in symbolic references are also used simply to describe the class or interface being defined by the class

file. The name of the class or interface being defined, for example, is given as a fully qualified name. For each field declared by the class or interface, the constant pool contains a simple name and field descriptor. For each method declared by the class or interface, the constant pool contains a simple name and method descriptor.

Fully Qualified Names

Whenever constant pool entries refer to classes and interfaces, they give the fully qualified name of the class or interface. In the class file, fully qualified names have their dots replaced with slashes. For example, the representation of the fully qualified name of `java.lang.Object` in the class file is `java/lang/Object`. The fully qualified name of `java.util.Hashtable` in the class file is `java/util/Hashtable`.

Simple Names

The names of fields and methods appear in constant pool entries as simple (not fully qualified) names. For example, a constant pool entry that refers to the `String toString()` method of class `java.lang.Object` would give its method name as `"toString"`. A constant pool entry that refers to the `java.io.PrintStream out` field of class `java.lang.System` would specify the field name simply as `"out"`.

Descriptors

Symbolic references to fields and methods include a descriptor string, in addition to a fully qualified class or interface name and a simple field or method name. A field descriptor gives the field's type. A method descriptor gives the method's return type and the number, types, and order of the method's parameters.

Field and method descriptors are defined by the context-free grammar shown as follows. Nonterminals of this grammar, such as *FieldType*, are shown in italic font. Terminals, such as `B` or `V`, are shown in fixed-width font. The asterisk character (*) stands for zero or more occurrences of the item that precedes it placed side by side (with no intervening white space).

```
FieldDescriptor:
       FieldType
ComponentType:
       FieldType
FieldType:
       BaseType
       ObjectType
       ArrayType
BaseType:
       B
       C
       D
       F
       I
       J
       S
       Z
ObjectType:
       L<classname>;
ArrayType:
       [ ComponentType
MethodDescriptor:
       ( ParameterDescriptor* ) ReturnDescriptor
ParameterDescriptor:
       FieldType
ReturnDescriptor:
       FieldType
       V
```

The meaning of each of the *BaseType* terminals is shown in Table 6-5. The `V` terminal represents methods that return `void`. Each of the eight *BaseType* characters, the *ReturnDescriptor* `V`, the `L` and `;` of *ObjectType*, the `[` of *ArrayType*, and the `(` and `)` characters of *MethodDescriptor* are all ASCII characters. (Except for the null character, each Unicode character that corresponds to an ASCII character is represented in UTF-8 form by that ASCII character.) The `<classname>` portion of an *ObjectType* is a fully qualified name. This fully qualified name, like all fully qualified names in the class file, appears with dots replaced by slashes.

Table 6-6 shows some examples of field descriptors, and Table 6-7 shows some examples of method descriptors. Note that the method descriptors for instance methods do not include the hidden `this` parameter passed as the first argument to all instance methods. Rather, this parameter is implicitly passed by all Java virtual machine instructions that invoke instance methods.

A method descriptor can contain only as many parameters as will fit into 255 words. The hidden `this` reference passed to instance methods occupies one word, and any parameters of the primitive types `long` or `double` occupy two words. Any other parameter occupies one word.

Table 6-5

BaseType terminals

Terminal	Type
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
Z	boolean

Table 6-6

Examples of field descriptors. (The this reference is never passed to class methods, because class methods are not invoked on an object.)

Descriptor	Field Declaration
`I`	`int i;`
`[[J`	`long[][] windingRoad;`
`[Ljava/lang/Object;`	`java.lang.Object[] stuff;`
`Ljava/util/Hashtable;`	`java.util.Hashtable ht;`
`[[[Z`	`boolean[][][] isReady;`

Table 6-7

Examples of method descriptors

Descriptor	Method Declaration
`()I`	`int getSize();`
`()Ljava/lang/String;`	`String toString();`
`([Ljava/lang/String;)V`	`void main(String[] args);`
`()V`	`void wait()`
`(JI)V`	`void wait(long timeout, int nanos)`
`(ZILjava/lang/String;II)Z`	`boolean regionMatches(boolean ignoreCase, int toOffset, String other, int ooffset, int len);`
`([BII)I`	`int read(byte[] b, int off, int len);`

The Constant Pool

The constant pool is an ordered list of variable-length `cp_info` tables, each of which follows the general form shown in Table 6-8. The `tag` item of a `cp_info` table, an unsigned byte, indicates the table's variety and format. `cp_info` tables come in 11 varieties, each of which is described in detail in the following sections.

The CONSTANT_Utf8_info Table

The variable-length `CONSTANT_Utf8_info` table stores one constant string value in a modified UTF-8 format. This table is used to store many different kinds of strings, including the following:

- string literals that get instantiated as `String` objects
- the fully qualified name of the class or interface being defined
- the fully qualified name of the superclass (if any) of the class being defined
- the fully qualified names of any superinterfaces of the class or interface being defined
- the simple names and descriptors of any fields declared by the class or interface
- the simple names and descriptors of any methods declared by the class or interface
- fully qualified names of any referenced classes and interfaces
- simple names and descriptors of any referenced fields
- simple names and descriptors of any referenced methods
- strings associated with attributes

As you can see from this list, four basic kinds of information are stored in `CONSTANT_Utf8_info` tables: string literals, descriptions of the class or interface being defined, symbolic references to other classes and interfaces, and strings associated with attributes. Some examples of strings

Table 6-8
General form of a `cp_info` table

Type	Name	Count
u1	tag	1
u1	info	depends on tag value

associated with attributes are as follows: the name of the attribute, the name of the source file from which the class file was generated, and the names and descriptors of local variables.

The UTF-8 encoding scheme permits all two-byte unicode characters to be represented in a string but enables ASCII characters (except the null character) to be represented by just one byte. Table 6-9 shows the format of a `CONSTANT_Utf8_info` table.

The items in the `CONSTANT_Utf8_info` table are as follows:

`tag`

The `tag` item has the value `CONSTANT_Utf8` (1).

`length`

The `length` item gives the length in bytes of the subsequent `bytes` item.

`bytes`

The `bytes` item contains the characters of the string which are stored in a modified UTF-8 format. Characters in the range `'\u0001'` through `'\u007f'` (all the ASCII characters except the null character) are represented by one byte:

byte 0

7	6	5	4	3	2	1	0
0	*6*	*5*	*4*	*3*	*2*	*1*	*0*

The `null` character, `'\u0000'`, and the characters in the range `'\u0080'` through `'\u07ff'` are represented by two bytes:

byte 0

7	6	5	4	3	2	1	0
1	1	0	*10*	*9*	*8*	*7*	*6*

byte 1

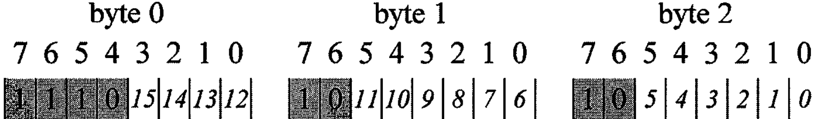
7	6	5	4	3	2	1	0
1	0	*5*	*4*	*3*	*2*	*1*	*0*

Table 6-9

Format of a `CONSTANT_Utf8_info` table

Type	Name	Count
u1	tag	1
u2	length	1
u1	bytes	length

Characters in the range '\u0800' through '\uffff' are represented by three bytes:



The encoding of UTF-8 strings in the bytes item of CONSTANT_Utf8_info tables differs from the standard UTF-8 format in two ways. First, in the standard UTF-8 encoding scheme, the null character is represented by one byte. In a CONSTANT_Utf8_info table, null characters are represented by two bytes. This two-byte encoding of nulls means that the bytes item never contains any byte equal to zero. The second way the bytes item of a CONSTANT_Utf8_info departs from the standard UTF-8 encoding is that only one-, two-, and three-byte encodings are used in the bytes item. The standard UTF-8 includes longer formats that are not used in CONSTANT_Utf8_info tables.

The CONSTANT_Integer_info Table

The fixed-length CONSTANT_Integer_info table stores a constant int value. This table is used only to store int literals and is not used in symbolic references. Table 6-10 shows the format of a CONSTANT_Integer_info table.

The items in the CONSTANT_Integer_info table are as follows:

tag

The tag item has the value CONSTANT_Integer (3).

bytes

The bytes item contains the int value stored in big-endian order.

Table 6-10

Format of a CONSTANT_Integer_info table

Type	Name	Count
u1	tag	1
u4	bytes	1

The CONSTANT_Float_info Table

The fixed-length CONSTANT_Float_info table stores a constant float value. This table is used only to store float literals and is not used in symbolic references. Table 6-11 shows the format of a CONSTANT_Float_info table.

The items in the CONSTANT_Float_info table are as follows:

tag

The tag item has the value CONSTANT_Float (4).

bytes

The bytes item contains the float value stored in big-endian order. For the details of the representation of float in the Java class file, see Chapter 14, "Floating Point Arithmetic."

The CONSTANT_Long_info Table

The fixed-length CONSTANT_Long_info table stores a constant long value. This table is only used to store long literals and is not used in symbolic references. Table 6-12 shows the format of a CONSTANT_Long_info table.

As noted previously, a long occupies two slots in the constant pool table. In the class file, a long entry is just followed by the next entry, but the index of the next entry is two more than that of the long entry.

The items of the CONSTANT_Long_info table are as follows:

tag

The tag item has the value CONSTANT_Long (5).

Table 6-11

Format of a CONSTANT_Float_info table

Type	Name	Count
u1	tag	1
u4	bytes	1

Table 6-12

Format of a CONSTANT_Long_info table

Type	Name	Count
u1	tag	1
u8	bytes	1

bytes

The bytes item contains the long value stored in big-endian order.

The CONSTANT_Double_info Table

The fixed-length CONSTANT_Double_info table stores a constant double value. This table is used only to store double literals and is not used in symbolic references. Table 6-13 shows the format of a CONSTANT_Double_info table.

As noted previously, a double occupies two slots in the constant pool table. In the class file, a double entry is just followed by the next entry, but the index of the next entry is two more than that of the double entry.

The items of the CONSTANT_Double_info table are as follows:

tag

The tag item has the value CONSTANT_Double (6).

bytes

The bytes item contains the double value stored in big-endian order. For the details of the representation of double in the Java class file, see Chapter 14, "Floating Point Arithmetic."

The CONSTANT_Class_info Table

The fixed-length CONSTANT_Class_info table represents a class or interface in symbolic references. All symbolic references, whether they refer to a class, interface, field, or method, include a CONSTANT_Class_info table. Table 6-14 shows the format of a CONSTANT_Class_info table.

The items in the CONSTANT_Class_info table are as follows:

tag

The tag item has the value CONSTANT_Class (7).

name_index

The name_index item gives the index of a CONSTANT_Utf8_info table that contains a fully qualified name of a class or interface.

Table 6-13

Format of a CONSTANT_Double_info table

Type	Name	Count
u1	tag	1
u8	bytes	1

Table 6-14

Format of a CONSTANT_Class_info table

Type	Name	Count
u1	tag	1
u2	name_index	1

Table 6-15

Format of a CONSTANT_String_info table

Type	Name	Count
u1	tag	1
u2	string_index	1

Because arrays are full-fledged objects in Java, CONSTANT_Class_info tables can also represent array classes. The name_index item of such a CONSTANT_Class_info table refers to a CONSTANT_Utf8_info table that contains the array's descriptor, which serves as the name of the array class. For example, the class name for the double[][] array type is its descriptor, [[D. The class name for the net.jini.core.lookup.ServiceItem[][][] array type is its descriptor, [[[Lnet/jini/core/lookup/ServiceItem;. Because a Java array can have no more than 255 dimensions, an array descriptor can have no more than 255 leading [characters.

The CONSTANT_String_info Table

The fixed-length CONSTANT_String_info represents a literal string value, which will be represented as an instance of class java.lang.String. This table is only used to represent literal strings and is not used in symbolic references. Table 6-15 shows the format of a CONSTANT_String_info table.

The items of the CONSTANT_String_info table are as follows:

tag

The tag item has the value CONSTANT_String (8).

string_index

The string_index item gives the index of a CONSTANT_Utf8_info entry that contains the value of the literal string.

Table 6-16

Format of a CONSTANT_Fieldref_info table

Type	Name	Count
u1	tag	1
u2	class_index	1
u2	name_and_type_index	1

The CONSTANT_Fieldref_info Table

The fixed-length CONSTANT_Fieldref_info table represents a symbolic reference to a field. Table 6-16 shows the format of a CONSTANT_Fieldref_info table.

The items of the CONSTANT_Fieldref_info table are as follows:

tag

The tag item has the value CONSTANT_Fieldref (9).

class_index

The class_index gives the index of the CONSTANT_Class_info entry for the class or interface that declares the referenced field.

Note that the CONSTANT_Class_info specified by class_index may represent an interface, not just a class. Although interfaces can declare fields, those fields are by definition public, static, and final. As mentioned in earlier chapters, class files do not contain symbolic references to static final fields of other classes if those fields are initialized with compile-time constants. Instead, class files contain a copy of the constant value of any such static final fields it uses. For example, if a class uses a static final field of type float that is declared in an interface and is initialized to a compile-time constant, the class would have a CONSTANT_Float_info table in its own constant pool that stores the float value. But if the interface initialized its static final field with an expression that can only be evaluated at run time, the class that uses the field would have a CONSTANT_Fieldref_info table in its constant pool that symbolically refers to the field in the interface. For more information about this special treatment of static final fields, see Chapter 8, "The Linking Model."

name_and_type_index

The name_and_type_index provides the index of a CONSTANT_NameAndType_info entry that gives the field's simple name and descriptor.

The CONSTANT_Methodref_info Table

The fixed-length CONSTANT_Methodref_info table represents a symbolic reference to a method declared in a class (not in an interface). Table 6-17 shows the format of a CONSTANT_Methodref_info table.

The items of the CONSTANT_Methodref_info table are as follows:

tag

The tag item has the value CONSTANT_Methodref (10).

class_index

The class_index gives the index of a CONSTANT_Class_info entry for the class that declares the referenced method. The CONSTANT_Class_ info table specified by class_index must be a class and not an interface. Symbolic references to methods declared in interfaces use CONSTANT_ InterfaceMethodref.

name_and_type_index

The name_and_type_index gives the index of a CONSTANT_ NameAndType_info entry that gives the method's simple name and descriptor. If the method's simple name begins with a < character ('\u003c'), the method must be an instance initialization method. Its simple name must be <init>, and its return type must be void. Otherwise, the method name must be a valid Java programming language identifier.

The CONSTANT_InterfaceMethodref_info Table

The fixed-length CONSTANT_InterfaceMethodref_info table is a symbolic reference to a method declared in an interface (not in a class). The format of a CONSTANT_InterfaceMethodref_info table is shown in Table 6-18.

The items in the CONSTANT_InterfaceMethodref_info table are as follows:

Table 6-17

Format of a CONSTANT_ Methodref_info table

Type	Name	Count
u1	tag	1
u2	class_index	1
u2	name_and_type_index	1

Table 6-18

Format of a CONSTANT_InterfaceMethodref_info table

Type	Name	Count
u1	tag	1
u2	class_index	1
u2	name_and_type_index	1

Table 6-19

Format of a CONSTANT_NameAndType_info table

Type	Name	Count
u1	tag	1
u2	name_index	1
u2	descriptor_index	1

tag

The tag item has the value CONSTANT_InterfaceMethodref (11).

class_index

The class_index gives the index of a CONSTANT_Class_info entry for the interface that declares the referenced method. The CONSTANT_Class_info table specified by class_index must be an interface and not a class. Symbolic references to methods declared in classes use CONSTANT_Methodref.

name_and_type_index

The name_and_type_index provides the index of a CONSTANT_NameAndType_info entry that gives the method's simple name and descriptor.

The CONSTANT_NameAndType_info Table

The fixed-length CONSTANT_NameAndType_info table forms part of a symbolic reference to a field or method. This table gives constant pool entries of the simple name and the descriptor of the referenced field or method. Table 6-19 shows the format of a CONSTANT_NameAndType_ info table.

The items of the CONSTANT_NameAndType_info table are as follows:

tag

The tag item has the value CONSTANT_NameAndType (12).

name_index

The name_index gives the index of a CONSTANT_Utf8_info entry that gives the name of the field or method. The name must be either a valid Java programming language identifier or <init>.

descriptor_index

The descriptor_index gives the index of a CONSTANT_Utf8_info entry that holds the descriptor of the field or method. The descriptor must be a valid field or method descriptor.

Fields

Each field (class variable and instance variable) declared in a class or interface is described by a variable-length field_info table in the class file. No two fields in the same class file can have the same name and descriptor. (Note that although no two fields declared in the same class or interface can have the same name in the Java programming language, two fields can have the same name in the class file—as long as the descriptor is different. In other words, although you cannot declare two fields with the same name but different types in the same class or interface in the Java language, two such fields can legally appear in the same Java class file.) The format of the field_info table is shown in Table 6-20.

The items in the field_info table are as follows:

access_flags

The modifiers used in declaring the field are placed into the field's access_flags item. Table 6-21 shows the bits used by each flag.

Table 6-20

Format of a field_info table

Type	Name	Count
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

Table 6-21

Flags in the access_flags item of field_info tables

Flag Name	Value	Meaning if Set	Set By
ACC_PUBLIC	0x0001	Field is public	Classes and interfaces
ACC_PRIVATE	0x0002	Field is private	Classes only
ACC_PROTECTED	0x0004	Field is protected	Classes only
ACC_STATIC	0x0008	Field is static	Classes and interfaces
ACC_FINAL	0x0010	Field is final	Classes and interfaces
ACC_VOLATILE	0x0040	Field is volatile	Classes only
ACC_TRANSIENT	0x0080	Field is transient	Classes only

For fields declared in a class (not an interface), one of ACC_PUBLIC, ACC_PRIVATE, and ACC_PROTECTED may be set (at most). ACC_FINAL and ACC_VOLATILE must not both be set. All fields declared in interfaces must have (and can only have) the ACC_PUBLIC, ACC_STATIC, and ACC_FINAL flags set.

All unused bits in access_flags must be set to zero and ignored by Java virtual machine implementations.

name_index

The name_index gives the index of a CONSTANT_Utf8_info entry that gives the simple (not fully qualified) name of the field. Each field name in a class file must be a valid field name in the Java programming language.

descriptor_index

The descriptor_index gives the index of a CONSTANT_Utf8_info entry that gives the descriptor of the field.

attributes_count and attributes

The attributes item is a list of attribute_info tables. The attributes_count indicates the number of attribute_info tables in the list. A field can have any number of attributes in its list. Three kinds of attributes defined by the Java virtual machine specification that may appear in this item are ConstantValue, Deprecated, and Synthetic. These three attributes are described in detail later in this chapter. The only field attribute that Java virtual machine implementations are required to recognize is the ConstantValue attribute. Implementations must ignore any attributes they do not recognize.

Methods

Each method declared in a class or interface or generated by the compiler is described in the class file by a variable-length method_info table. No two methods in the same class file can have the same name and descriptor. Note that although no two methods declared in the same class or interface in the Java programming language can have the same signature (the descriptor minus the return type), two methods can have the same signature in the class file so long as the descriptor is different. In other words, when in the same class in a Java source file, if you try to declare two methods with the same name and number and types of parameters but different return types, the program will not compile. In the Java programming language, you cannot overload methods by varying only the return type. Two such methods can coexist happily in a Java class file, however.

The two types of compiler-generated methods that may appear in class files are instance initialization methods (named <init>) and class and interface initialization methods (named <clinit>). For more information on the compiler-generated methods, see Chapter 7, "The Lifetime of a Class." The format of the method_info table is shown in Table 6-22.

The items in the method_info table are as follows:

access_flags

The modifiers used in declaring the method are placed into the method's access_flags item. Table 6-23 shows the bits used by each flag. The ACC_STRICT flag was added in 1.2 and indicates that all expressions in the method should be evaluated in FP-strict mode. FP-strict mode is described in detail in Chapter 14, "Floating-Point Arithmetic."

For methods declared in a class (not an interface), one of ACC_PUBLIC, ACC_PRIVATE, and ACC_PROTECTED may be set (at most). If a method's ACC_ABSTRACT flag is set, then its ACC_PRIVATE, ACC_STATIC, ACC_FINAL, ACC_SYNCHRONIZED, ACC_NATIVE, and ACC_STRICT flags must not be set. All methods declared in interfaces must have their

Table 6-22

Format of a method_info table

Type	Name	Count
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

Table 6-23

Flags in the access_flags item of method_info tables

Flag Name	Value	Meaning if Set	Set By
ACC_PUBLIC	0x0001	Method is public	Classes and all methods of interfaces
ACC_PRIVATE	0x0002	Method is private	Classes only
ACC_PROTECTED	0x0004	Method is protected	Classes only
ACC_STATIC	0x0008	Method is static	Classes only
ACC_FINAL	0x0010	Method is final	Classes only
ACC_SYNCHRONIZED	0x0020	Method is synchronized	Classes only
ACC_NATIVE	0x0100	Method is native	Classes only
ACC_ABSTRACT	0x0400	Method is abstract	Classes and all methods of interfaces
ACC_STRICT	0x0800	Method is strict FP	Classes and the <clinit> method of interfaces

ACC_PUBLIC and ACC_ABSTRACT flags set. Interface methods may have no other flags set, except for the interface initialization (<clinit>) method, which may have its ACC_STRICT flag set.

Instance initialization (<init>) methods may only use flags ACC_PUBLIC, ACC_PRIVATE, and ACC_PROTECTED. Because class and interface initialization (<clinit>) methods are invoked by the Java virtual machine and never directly by Java bytecodes, the bits of the access_flags for <clinit> methods—except for ACC_STRICT—are ignored.

All unused bits in access_flags must be set to zero and ignored by Java virtual machine implementations.

name_index

The name_index gives the index of a CONSTANT_Utf8_info entry that gives the simple (not fully qualified) name of the method. The name must be either <init>, <clinit>, or a valid method name (simple, not fully qualified) in the Java programming language.

descriptor_index

The descriptor_index gives the index of a CONSTANT_Utf8_info entry that gives the descriptor of the method.

attributes_count and attributes

The attributes item is a list of attribute_info tables. The attributes_count indicates the number of attribute_info tables in

the list. A field can have any number of attributes in its list. Four kinds of attributes defined by the Java virtual machine specification that may appear in this item are `Code`, `Deprecated`, `Exceptions`, and `Synthetic`. These four attributes are described in detail later in this chapter. The only method attributes that Java virtual machine implementations are required to recognize are the `Code` and `Exceptions` attributes. Implementations must ignore any attributes they do not recognize.

Attributes

As mentioned previously, attributes appear in several places inside a Java class file. They can appear in the `ClassFile`, `field_info`, `method_info`, and `Code_attribute` tables. The `Code_attribute` table, an attribute itself, is described later in this section.

The Java virtual machine specification defines nine types of attributes, which are shown in Table 6-24. To correctly interpret Java class files, all Java virtual machine implementations must recognize three of these attributes: `Code`, `ConstantValue`, and `Exceptions`. To properly implement the Java and Java 2 platform class libraries, implementations must recognize `InnerClasses` and `Synthetic` attributes. Implementations can choose whether to recognize or ignore the other predefined attributes. (The

Table 6-24

Types of `attribute_info` tables defined by the specification

Name	Used By	Description
`Code`	`method_info`	The bytecodes and other data for one method
`ConstantValue`	`field_info`	The value of a final variable
`Deprecated`	`field_info`, `method_info`	An indicator that a field or method has been deprecated
`Exceptions`	`method_info`	The checked exceptions that a method may throw
`InnerClasses`	`ClassFile`	A list of inner and outer classes
`LineNumberTable`	`Code_attribute`	A mapping of line numbers to bytecodes for one method
`LocalVariableTable`	`Code_attribute`	A description of the local variables for one method
`SourceFile`	`ClassFile`	The name of the source file
`Synthetic`	`field_info`, `method_info`	An indicator that a field or method was generated by the compiler

Table 6-25

Format of an `attribute_info` table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

Deprecated, InnerClasses, and Synthetic attributes were added in Java 1.1.) All of these predefined attributes are described in detail later in this chapter.

Anyone (besides Sun) who wishes to add a new attribute to a Java class file must follow these two rules:

1. Any attribute that is not predefined by the specification must not affect the semantics of class or interface types. New attributes can only add more information to the class file, such as information used during debugging.
2. The attribute must be named using the reverse Internet domain name scheme that is defined for package naming in the Java Language Specification. For example, if you had the Internet domain name artima.com and you wished to create a new attribute named CompilerVersion, you would name the attribute com.artima.CompilerVersion.

Attribute Format

Every attribute follows the same general format of the variable-length attribute_info table, shown in Table 6-25. The first two bytes of an attribute, the attribute_name_index, form an index into the constant pool of a CONSTANT_Utf8_info table that contains the string name of the attribute. Each attribute_info, therefore, identifies its "type" by the first item in its table, much like the way cp_info tables identify their type by the initial tag byte. The difference is that whereas the type of a cp_info table is indicated by an unsigned byte value, such as 3 (CONSTANT_Integer_info), the type of an attribute_info table is indicated by a string.

Following the attribute_name_index is a four-byte attribute_length item, which gives the length of the entire attribute_info table minus the initial six bytes. (The attribute_length item can be zero.) This length is necessary, because anyone following certain rules (outlined below) is allowed to add attributes to a Java class file. Java virtual machine implementations are allowed to recognize new attributes. Implementations must ignore any attributes they do not recognize. The

attribute_length enables virtual machines to skip unrecognized attributes as they parse the class file.

The items of the attribute_info table are as follows:

attribute_name_index

The attribute_name_index gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the name of the attribute.

attribute_length

The attribute_length item indicates the length (in bytes) of the attribute data, excluding the initial six bytes that contain the attribute_name_index and attribute_length.

info

The info item contains the attribute data.

The Code Attribute

The variable-length Code_attribute table defines the bytecode sequence and other information for a method. One Code_attribute table appears in the method_info table of every method that is not abstract or native. The format of a Code_attribute table is shown in Table 6-26.

The items of the Code_attribute table are as follows:

Table 6-26

Format of a Code_attribute table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count

attribute_name_index

The attribute_name_index item gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the string "Code".

attribute_length

The attribute_length item gives the length in bytes of the Code attribute excluding the initial six bytes that contain the attribute_name_index and attribute_length items.

max_stack

The max_stack item gives the maximum number of words that will be on the operand stack of this method at any point during its execution.

max_locals

The max_locals item gives the number of words in the local variables that are required by this method. The virtual machine must allocate an array of local variables of length max_locals whenever it invokes the method being described by this Code attribute. This array will be used to store parameters passed to the method and local variables used by the method. The maximum valid local variable index for a value of type long or double is max_locals-2. The maximum valid local variable index for a value of any other type is max_locals-1.

code_length and code

The code_length item gives the length (in bytes) of the bytecode stream for this method. The bytecodes themselves appear in the code item. The value of code_length must be greater than zero.

exception_table_length and exception_table

The exception_table item is a list of exception_info tables. Each exception_info table describes one exception table entry. The exception_table_length item gives the number of exception_info tables that appear in the exception_table list. The order in which the exception_info tables appear in the list is the order in which the Java virtual machine will check for a matching exception handler (catch clause) if an exception is thrown while this method executes. The format of an exception_info table is shown in Table 6-27 and is described in the next section, "The exception_info Table." For more information about exception tables, see Chapter 17, "Exceptions."

attributes_count and attributes

The attributes item is a list of attribute_info tables. The attributes_count indicates the number of attribute_info tables in

Table 6-27

Format of an exception_info table

Type	Name	Count
u2	start_pc	1
u2	end_pc	1
u2	handler_pc	1
u2	catch_type	1

the list. The two kinds of attributes defined by the Java virtual machine specification that may appear in this item are LineNumberTable and LocalVariableTable. These two attributes are described in detail later in this chapter. Java virtual machine implementations are permitted to ignore any attributes in the attributes item of the Code attribute and are required to ignore any they do not recognize.

The exception_info Table The fixed-length exception_info table describes one exception table entry. This table appears in the Code attribute's exception_info item, which is composed of a list of exception_info tables. The format of the exception_info table is shown in Table 6-27. For more information about exception tables, see Chapter 17, "Exceptions."

The items in the exception_info table are as follows:

start_pc

The start_pc item gives the offset from the beginning of the code array for the beginning of the range covered by this exception handler.

end_pc

The end_pc item gives the offset from the beginning of the code array for one byte past the end of the range covered by this exception handler.

handler_pc

The handler_pc item gives the offset from the beginning of the code array for the instruction to jump to the first instruction of the exception handler—if a thrown exception is caught by this entry.

catch_type

The catch_type item gives the constant pool index of a CONSTANT_Class_info entry for the type of exception caught by this exception handler. The CONSTANT_Class_info entry must represent class java.lang.Throwable or one of its subclasses.

If the value of catch_type is zero (which is not a valid index into the constant pool, because the constant pool starts at index one), the exception handler handles all exceptions. A catch_type of zero is used to implement finally clauses. See Chapter 18, "Finally Clauses," for more information about how finally clauses are implemented.

The ConstantValue Attribute

The fixed-length ConstantValue attribute appears in field_info tables for fields that have a constant value. At most, one ConstantValue attribute may appear in the attributes item of a given field_info table. In the access_flags of a field_info table which includes a ConstantValue attribute, the ACC_STATIC flag must be set. The ACC_FINAL flag may also be set, although this action is not required. When the virtual machine initializes a field that has a ConstantValue attribute, it assigns the constant value to the field. This assignment occurs immediately before the virtual machine invokes the class or interface initialization method for the class or interface in which the field is declared. The format of a ConstantValue_attribute table is shown in Table 6-28.

The items of the ConstantValue_attribute table are as follows:

attribute_name_index

The attribute_name_index gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the string "ConstantValue".

attribute_length

The attribute_length item of a ConstantValue_attribute is always 2.

constantvalue_index

The constantvalue_index item gives the index in the constant pool of an entry that gives a constant value. Table 6-29 shows the type of entry for each type of field.

Table 6-28

Format of a ConstantValue_attribute table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

Table 6-29

Constant pool entry types for constant value attributes

Type of Constant Value	Type of Constant Pool Entry
byte, short, char, int, boolean	CONSTANT_Integer_info
long	CONSTANT_Long_info
float	CONSTANT_Float_info
double	CONSTANT_Double_info
java.lang.String	CONSTANT_String_info

Table 6-30

Format of a Deprecated_ attribute table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1

The Deprecated Attribute

The fixed-length Deprecated attribute, which may optionally appear in the attributes items of field_info, method_info, and ClassFile tables, indicates that a field, method, or type has been deprecated. (*Deprecated* means that although the field, method, or type still exists and functions as expected, programmers are encouraged not to use that approach. Rather, programmers are encouraged to use some other preferred field, method, type, or approach, instead of using the deprecated item.) A compiler, virtual machine, or any other tool that reads class files can use the Deprecated attribute to notify the programmer that the program is using a deprecated field, method, or type. The Deprecated attribute was added in Java 1.1 to support the @deprecated tag in documentation comments used by the javadoc tool. The format of a Deprecated_attribute is shown in Table 6-30.

The items of the Deprecated_attribute table are as follows:

attribute_name_index

The attribute_name_index gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the string "Deprecated".

attribute_length

The attribute_length must be zero.

Table 6-31

Format of an `Exceptions_attribute` table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_exceptions	1
u2	exception_index_table	number_of_exceptions

The Exceptions Attribute

The variable-length `Exceptions` attribute lists the checked exceptions that a method may throw. One `Exceptions_attribute` table appears in the `method_info` table of every method that may throw checked exceptions. The format of an `Exceptions_attribute` table is shown in Table 6-31.

A method should only throw an exception if it is an instance or subclass of either `RuntimeException`, `Error`, or one of the exceptions listed in the method's `Exceptions` attribute. Although this rule should be enforced by Java compilers, it is not enforced by Java virtual machines. Thus, the `Exceptions` attribute exists in the Java class file for the benefit of Java compilers.

The items of the `Exceptions_attribute` table are as follows:

`attribute_name_index`

The `attribute_name_index` gives the index in the constant pool of a `CONSTANT_Utf8_info` entry that contains the string, `"Exceptions"`.

`attribute_length`

The `attribute_length` item provides the length (in bytes) of the `Exceptions_attribute`, excluding the initial six bytes that contain the `attribute_name_index` and `attribute_length` items.

`number_of_exceptions` and `exception_index_table`

The `exception_index_table` is an array of indexes into the constant pool of `CONSTANT_Class_info` entries for the exceptions declared in this method's `throws` clause. In other words, the `exception_index_table` lists all the checked exceptions in which this method may throw. The `number_of_exceptions` item indicates the number of indexes in the array.

The InnerClasses Attribute

The variable-length `InnerClasses` attribute describes the names, access flags, and enclosing types of any nested types that are declared as members

of, or are otherwise mentioned by, a class or interface. (A *nested type* is a type that is not a member of a package, but rather is a member of a class or interface.) If the code of a class or interface refers to a nested type, the constant pool of that class or interface will contain a CONSTANT_Class_info entry for that nested type. The constant pool must also contain a CONSTANT_Class_info entry for each nested type (if any) that is declared as an immediate member of a class or interface—even if the class or interface would not otherwise mention the nested type. If the constant pool of a class or interface contains any CONSTANT_Class_info entries for nested types, the class file for that class or interface must contain one InnerClasses_attribute table in the attributes item of its ClassFile table. The format of an InnerClasses_ attribute table is shown in Table 6-32.

The Java virtual machine does not currently verify that class files representing types mentioned by an InnerClasses_attribute table are consistent with the InnerClasses_attribute.

The items of an InnerClasses_attribute table are as follows:

attribute_name_index

The attribute_name_index gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the string "InnerClasses".

attribute_length

The attribute_length item gives the length of the InnerClasses_attribute in bytes, excluding the initial six bytes that contain the attribute_name_index and attribute_length items.

number_of_classes and classes

The classes item is an array of inner_class_info tables. The number_of_classes gives the number of inner_class_info tables that appear in the classes array. The format of the inner_class_info table is shown in Table 6-33 and is described in the next section, "The inner_class_info Table."

The classes item of the InnerClasses attribute contains one inner_class_info table for each nested class mentioned in a CONSTANT_

Table 6-32

Format of an InnerClasses_attribute table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_classes	1
inner_classes_info	classes	number_of_classes

Table 6-33

Format of an `inner_class_info` table

Type	Name	Count
u2	inner_class_info_index	1
u2	outer_class_info_index	1
u2	inner_name_index	1
u2	inner_class_access_flags	1

`Class_info` entry of the constant pool. Because a `CONSTANT_ Class_info` entry must appear in an enclosing type's constant pool for each nested type declared as an immediate member of that enclosing type, the `classes` item of the enclosing type's `InnerClasses` attribute will definitely contain an `inner_class_info` table for each nested type declared as an immediate member of the enclosing type.

For example, if class `Rain`, class `Snow`, and interface `Wet` are declared as members of class `Weather`, the `InnerClasses` attribute for `Weather` will definitely contain an `inner_class_info` table for `Rain`, `Snow`, and `Wet`. Likewise, if class `Thunder` is declared as a member of class `Rain` (which is declared inside `Weather`), the `InnerClasses` attribute for `Rain` will definitely contain an `inner_class_info` table for `Thunder`. An `inner_ class_info` table for class `Thunder` may also appear in `Weather`'s `InnerClasses` attribute, but not necessarily. Because `Thunder` is not declared as a member of `Weather`, `Thunder` will appear in `Weather`'s `InnerClasses` attribute only if `Weather`'s code explicitly refers to `Thunder`.

In addition to mentioning all nested types declared as members, the `InnerClasses` attribute will mention all enclosing classes of a nested type. All types always mention themselves in their own constant pool, in the `CONSTANT_Class_info` entry referred to by the `this_class` item of their `ClassFile` table. Thus, if the type being defined by a class file is a nested type (not a member of a package, but a member of some other class or interface), the type being defined will appear in its own `InnerClasses` attribute. Because the `outer_class_info_index` item of the `inner_class_info` table for a given nested type refers to the enclosing type of that nested type, the `InnerClasses` attribute in the class file that defines a nested type will include an `inner_class_info` table for all of its enclosing types.

For example, if class `Thunder` is declared as a member of class `Rain`, class `Rain` as a member of class `Weather`, and class `Weather` as a member of a package, the `InnerClasses` attribute for class `Thunder` will definitely include `inner_class_info` tables for both of its enclosing types, `Rain` and `Weather`. Similarly, the `InnerClasses` attribute for class `Rain` will definitely include an `inner_class_info` table for its enclosing type, `Weather`.

The inner_class_info Table The fixed-length inner_class_info table, which is contained in the classes item of an InnerClasses attribute, provides information about a type that is either a nested type itself or is a type in which at least one other type is declared as a member. (In other words, each inner_class_ info table describes a type that is either a nested type, an enclosing type, or both.) The format of the inner_class_info table is shown in Table 6-33.

The items of the inner_class_info table are as follows:

inner_class_info_index

The inner_class_info_index gives an index into the constant pool for the CONSTANT_Class_info entry that represents the nested class described by this inner_class_info table.

outer_class_info_index

The outer_class_info_index gives an index into the constant pool for the CONSTANT_Class_info entry (of the type in which a nested type described by this inner_class_info table is declared as a member). If this inner_class_info table does not describe a nested type, the outer_class_info_index must be zero. inner_class_info tables may describe types that are not nested (in other words, types that are declared as members of a package), because enclosing types are also mentioned in the InnerClasses attribute. The outermost enclosing type of any nested type will always be a member of a package.

For example, if class Rain is declared as a member of class Weather, and class Weather is declared as a member of a package, class Rain's InnerClasses attribute will include an inner_class_info table for Weather. Because Weather is declared as a member of a package, its outer_class_info_index will be zero.

inner_name_index

Unless this inner_class_info table describes an anonymous inner class, the inner_name_index gives an index in the constant pool for a CONSTANT_Utf8_info entry that gives the simple name of the type described by this inner_class_info table. If this inner_class_info table describes an anonymous inner class, the inner_name_index will be zero.

Note that for any type described by an inner_class_info table, you can always obtain the name of the type by conducting a two-step lookup process: First, follow the inner_class_info_index to a CONSTANT_ Class_info entry for the type. Then, follow CONSTANT_Class_info entry's name_index item to a CONSTANT_Utf8_info entry that gives the

simple name of the type. For anonymous inner classes, this two-step lookup process will yield the name given to the anonymous inner class by the compiler. For any other (non-anonymous) type described by an inner_class_info table, the two-step lookup process will yield the same name referred to by the inner_name_index. The inner_name_index, therefore, is not strictly needed for getting at the name of a type described by a inner_class_info table. Rather, the inner_name_index serves primarily to differentiate those types that started out as anonymous inner classes in the source code (whose names were generated by a compiler) from non-anonymous types (whose names were typed into the source code by a programmer).

inner_class_access_flags

The inner_class_access_flags item gives the access flags for the inner class. Compilers use these flags to recover information about the declaration of nested classes when the original source code is not available. The flags used in this item are shown in Table 6-34. All unused bits in inner_class_access_flags must be set to zero by compilers and ignored by Java virtual machine implementations.

The LineNumberTable Attribute

The variable-length LineNumberTable attribute maps offsets in a method's bytecode stream to line numbers in the source file. One LineNumberTable_attribute table may appear (it is optional) in the attributes component of Code_attribute tables. The format of a LineNumberTable_attribute table is shown in Table 6-35.

Table 6-34

Flag bits in the inner_class_access_flags item of inner_class_info tables

Flag Name	Value	Meaning if Set
ACC_PUBLIC	0x0001	Marked or implicitly public in the source
ACC_PRIVATE	0x0002	Marked private in the source
ACC_PROTECTED	0x0004	Marked protected in the source
ACC_STATIC	0x0008	Marked or implicitly static in the source
ACC_FINAL	0x0010	Marked final in the source
ACC_INTERFACE	0x0200	Was an interface in the source
ACC_ABSTRACT	0x0400	Marked or implicitly abstract in the source

Table 6-35

Format of a LineNumber-Table_attribute table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_number_table_length

The items of the LineNumberTable_attribute table are as follows:

attribute_name_index

The attribute_name_index gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the string "Line_Number_Table".

attribute_length

The attribute_length item provides the length (in bytes) of the LineNumberTable_attribute, excluding the initial six bytes that contain the attribute_name_index and attribute_length items.

line_number_table_length and line_number_table

The line_number_table item is an array of line_number_info tables. The line_number_table_length gives the number of line_number_info tables that appear in the line_number_table array. The tables in this array may appear in any order, and there may be more than one table for the same line number. The format of a line_number_info is shown in Table 6-36 and is described in the next section.

The line_number_info Table The fixed-length line_number_info table, which is contained in the line_number_table item of a LineNumberTable_attribute table, relates one source code line number to an instruction in the bytecode array which corresponds to the beginning of the compiled form of that line of source code. The format of a line_number_info is shown in Table 6-36.

The items of the line_number_info table are as follows:

start_pc

The start_pc item gives an offset from the beginning of the code array where a new line begins. The value of start_pc must be less than the value of the code_length item found in the Code attribute to which this LineNumberTable attribute belongs.

Table 6-36

Format of a line_number_info table

Type	Name	Count
u2	start_pc	1
u2	line_number	1

Table 6-37

Format of a LocalVariable-Table_attribute table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

line_number

The line_number item gives the line number of the line that begins at start_pc.

The LocalVariableTable Attribute

The variable-length LocalVariableTable attribute maps words in the local variables portion of the method's stack frame to names and descriptors of local variables in the source code. One LocalVariableTable_attribute table may appear (but is optional) in the attributes component of Code_attribute tables. The format of a LocalVariableTable_attribute table is shown in Table 6-37.

The items in the LocalVariableTable_attribute table are as follows:

attribute_name_index

The attribute_name_index gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the string "Localattribute_length".

The attribute_length item gives the length (in bytes) of the LocalVariableTable_attribute, excluding the initial six bytes that contain the attribute_name_index and attribute_length items.

local_variable_table_length and local_variable_ table

The local_variable_table item is an array of local_variable_info tables. The local_variable_table_length gives the number of local_variable_info tables that appear in the local_variable_table array. The format of a local_variable_info table is shown in Table 6-38 and is described in the next section.

The local_variable_info Table The fixed-length local_variable_info table, which is contained in the local_variable_table item of a LocalVariableTable_attribute table, relates one source code local variable name and type to its scope in the bytecode array and index in the local variables of the stack frame. The format of a local_variable_info is shown in Table 6-38.

The items in the local_variable_info table are as follows:

start_pc and length

The start_pc item gives an offset in the code array of the start of an instruction. The length item gives the length of the range of code that starts with start_pc for which a local variable is valid. The byte at offset start_pc + length from the beginning of the code array must either be the first byte of an instruction or the first byte past the end of the code array.

name_index

The name_index item gives an index in the constant pool of a CONSTANT_Utf8_info entry for the name of the local variable.

descriptor_index

The descriptor_index item gives an index in the constant pool of a CONSTANT_Utf8_info entry that contains the descriptor for this local variable. (A local variable descriptor adheres to the same grammar as a field descriptor.)

Table 6-38

Format of a local_variable_info table

Type	Name	Count
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

index

The index item gives the index in the local variable portion of this method's stack frame, where the data for this local variable is kept as the method executes. If the local variable is of type long or double, the data occupies two words at positions index and index + 1. Otherwise, the data occupies one word at position index.

The SourceFile Attribute

The fixed-length SourceFile attribute, which may optionally appear in the attributes component of a ClassFile table, gives the name of the source file from which the class file was generated. No more than one SourceFile_attribute table can appear in the attributes table of a ClassFile table. The format of a SourceFile_attribute table is shown in Table 6-39.

The items of the SourceFile_attribute table are as follows:

attribute_name_index

The attribute_name_index gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the string "SourceFile".

attribute_length

The attribute_length item of a SourceFile_attribute is always two.

sourcefile_index

The sourcefile_index item gives the index in the constant pool of a CONSTANT_Utf8_info entry that contains the name of the source file. The source file name never includes a directory path.

Table 6-39

Format of a SourceFile_attribute table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

The Synthetic Attribute

The fixed-length `Synthetic` attribute, which may optionally appear in the `attributes` items of `field_info`, `method_info`, and `ClassFile` tables, indicates that a field, method, or type was generated by the compiler. A class member that does not appear in the source code must be marked with a `Synthetic` attribute. The `Synthetic` attribute was added in Java 1.1 to support nested classes. The format of a `Synthetic_attribute` is shown in Table 6-40.

The items in the `Synthetic_attribute` table are as follows:

`attribute_name_index`

The `attribute_name_index` gives the index in the constant pool of a `CONSTANT_Utf8_info` entry that contains the string `"Synthetic"`.

`attribute_length`

The `attribute_length` must be zero.

Getting Loaded: A Simulation

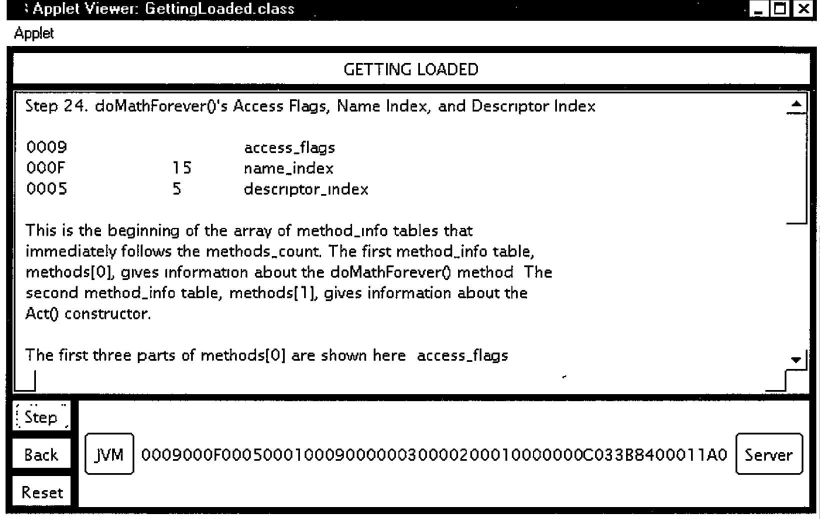
The *Getting Loaded* applet, shown in Figure 6-3, simulates a Java virtual machine loading a class file. The class file being loaded in the simulation was generated by the 1.1 `javac` compiler from the following Java source code. Although the snippet of code used in the simulation may not be useful in the real world, it does compile to a real class file and provides a reasonably simple example of the class-file format. This class is the same one used in the *Eternal Math* simulation applet described in Chapter 5, "The Java Virtual Machine."

```
// On CD-ROM in file classfile/ex1/Act.java
class Act {

    public static void doMathForever() {
        int I = 0;
        for (;;) {
            I += 1;
            I *= 2;
        }
    }
}
```

Table 6-40
Format of a `Synthetic_attribute` table

Type	Name	Count
u2	attribute_name_index	1
u4	attribute_length	1

Figure 6-3
The Getting Loaded applet



The *Getting Loaded* applet enables you to drive the class load simulation one step at a time. For each step along the way, you can read about the next chunk of bytes that is about to be consumed and interpreted by the Java virtual machine. Just press the "Step" button to cause the Java virtual machine to consume the next chunk. Pressing "Back" will undo the previous step, and pressing "Reset" will return the simulation to its original state, enabling you to start over.

The Java virtual machine is shown at the bottom left-hand side as it consumes the stream of bytes that makes up the class file `Act.class`. The bytes are shown in hex streaming out of a server on the bottom right-hand side. The bytes travel right to left, between the server and the Java virtual machine, one chunk at a time. The chunk of bytes to be consumed by the Java virtual machine on the next "Step" button press are shown in red. These highlighted bytes are described in the large text area above the Java virtual machine. Any remaining bytes beyond the next chunk are shown in black.

As mentioned in previous sections, many items in the class file refer to constant pool entries. To make it easier for you to look up constant pool entries as you step through the simulation, a list of the contents of `Act`'s constant pool is shown in Table 6-41.

Each chunk of bytes is fully explained in the text area. Because there is a lot of detail in the text area, you may wish to skim through all the steps first to get the general idea, then look back for more details.

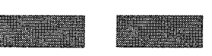
Table 6-41

Class Act's constant pool

Index	Type	Value
1	CONSTANT_Class_info	7
2	CONSTANT_Class_info	16
3	CONSTANT_Methodref_info	2, 4
4	CONSTANT_NameAndType_info	6, 5
5	CONSTANT_Utf8_info	"()V"
6	CONSTANT_Utf8_info	"<init>"
7	CONSTANT_Utf8_info	"Act"
8	CONSTANT_Utf8_info	"Act.java"
9	CONSTANT_Utf8_info	"Code"
10	CONSTANT_Utf8_info	"ConstantValue"
11	CONSTANT_Utf8_info	"Exceptions"
12	CONSTANT_Utf8_info	"LineNumberTable"
13	CONSTANT_Utf8_info	"LocalVariables"
14	CONSTANT_Utf8_info	"SourceFile"
15	CONSTANT_Utf8_info	"doMathForever"
16	CONSTANT_Utf8_info	"java/lang/Object"

On the CD-ROM

The CD-ROM contains the source code examples from this chapter in the `classfile` directory. The *Getting Loaded* applet is contained in a Web page on the CD-ROM in file `applets/GettingLoaded.html`. The source code for this applet is found alongside its class files in the `applets/GettingLoaded` directory.



The Resources Page

For more information about class files, visit the resources page: `http://www.artima.com/insidejvm/resources/`.

INDEX

Note: **boldface** numbers indicate illustrations.

100 percent Pure Java certification, platform independence, 36

A

aaload opcode, 443
aastore opcode, 444
abrupt completion of methods in, 163
Abstract Windows Toolkit (AWT), 12–13
  layout managers for, 13
  platform independence of, 12–13, 20
  security and, 66
access control (*see also* security), 44–45, 50–51, 62–68, 80, 86–109
  AccessController for, 86–109
  class loaders for, 50–51
  Java Authentication and Authorization Service (JAAS), 109–110
  memory and, 60
  security and, 80
  security manager and, 62–68
Access Controller, 14, 86–109
  checkPermission( ) method in, 87
  doPrivileged( ) method in, 100–109
  implies( ) method in, 87–89, 87
  stack inspection by, examples of "yes" and "no," 90–99
access flags, class files 196–197
aconst_null opcode, 387
acquiring the monitor for synchronization, 498
active use of Java types, 239, 251–253
ActiveX, security and, 79
adaptive garbage collection, 362–363
adaptive optimization, 6–7, 181–183
administration across distributed systems, mobility concepts and, 115–116
ahead-of-time compilers, 18
aload, aload_*xxx* opcodes, 392
AND (*see* logic)
anewarray opcode, 442
applet class loaders, 10
applets, 122–125, **124**
  browser use of, 123–125
  class loaders and, 125
  dynamic linking and, 125
  HTML tags for, 124
  mobility concepts and, 122–125, **124**
  platform independence and, 123
  security, 123
  symbol reference and, 125
application programming interface (API), 1, 4, 12–14, **13**, 19–20
applications for Java, 2
architecture of Java (*see also* class files; platform independence; virtual machine), 1–21
  ahead-of-time compilers and, 18
  application programming interface (API) for, 1, 4, 12–14, **13**, 19–20
  class files and, 20
  class files in, 1, 4, 8–11, **9**, 11–12
  compilers and, 18
  distributed applications and, 17–19
  Java API for, 12–14, **13**
  Java/Java 2 platform in, 4–5, **5**
  mobility concepts and, 11–12, 120–122
  namespaces and, 10–11

platform independence and, 11–12, 19–20, 23–40
programming language for, 4, 14–16
resources Web site for, 21
speed of execution, 17
tradeoffs of 16–21
virtual machine defined for, 1, 4, 5–7, **6, 8,** 136–187
areturn opcode, 495
arithmetic
floating-point arithmetic, 423–436
integer arithmetic, 407–416
array data type, 141
arraylength opcode, 442
arrays, 161, **162**, 437–447
arrays of arrays, 161
bounds checking for, 16, 60
class files and, 210
linking and, 278–279
local variables versus, 438
multi-dimension or arrays-of-arrays, 161
opcodes for, 440–443
primitive types and, 438
resolution and, 278–279
storing, 444
Three-Dimensional Array applet for, 443–447, **444**
arrays of arrays, 161
ArrayType descriptors in, 203
astore, astore_*xxx* opcodes, 393
asymmetrical invocation of finally clauses, 471–474
atomic types, 184–185
attributes, 215, 218–229
class files, 201
code as, 220–223
ConstantValue, 223, 224
deprecated, 224, 225
exception_info table for, 222
exceptions, 225
format of, 219–220, 219
inner_class_info table for, 228–229
InnerClasses, 225–229
line_number_info table for, 230–231
LineNumberTable, 229–231
local_variable_info table for, 232–233
LocalVariableTable, 231–233
service objects, Jini technology and, 128
SourceFile, 233
Synthetic, 234
types of, 218–219, **218**
authentication (*see also* security), 43, 68–75
JAR files and, 69–73, **71, 72**, 77–79
Java Authentication and Authorization Service (JAAS), 109–110

B

baload opcode, 443
BaseType descriptors in, 203, **204**
bastore opcode, 444
big-endian byte order in class files, 11, 146, 193
binary compatibility
class files and, 58–56, 121
mobility concepts and, 121
binary data, binary files, 3, 11, 20, 240–241
Java types and, 240–241
binding, 480
bipush opcode, 389
bootstrap class loaders (*see also* class loaders), 8–11, 9, 47–49, 132, 142, 143–144, 188, 241, 264–268
bounds checking, arrays, 16, 60

branching, 449–458
conditional branching in, 450–452
conditional branching with tables in, 453–455
unconditional branching in, 453
browsers and applets, 123–125
bugs, security and, 42, 44, 79–80
platform independence and, 34
built-in class loaders, 48–50
bytecodes, bytecode processing, 5–6, 174–181, 188, 243
class file verifier actions on, 53
data flow analyzer for, 244
finally clauses and, 469–478
frames in, 55
Halting Problem and, 55–56
Java stack and, 55
opcodes in, 55
operand stack in, 55
_quick instructions, 305–306
threads in, 55
verification of, through class file verifier, 54, 244

C

C++ versus Java programming language, 15
platform independence and, 26
caches for garbage collection, 376–378
caload opcode, 443
canonicializing mappings for garbage collection, 376–378
castore opcode, 444
catching exceptions, 460–464
central processing unit (CPU), mobility concepts and, 114
certificate authorities (CA) and, 74–75, 80–81
check methods for security manager, 65–68
checkcast opcode, 440
checkPermission( ) method, 87
Circle of Squares applet, 423, 434–436, **435**
class class reference in, 150–151
class data type, 141
class file verifier, 52–59
binary compatibility and, 58–56
bytecode analysis through, 53
bytecode verification with, 54–56
Halting Problem and, 55–56
robustness and, 52–53
security and, 44
semantic checks on type data with, 54
structural checks on class file with, 53–54
symbolic reference verification with, 56–58
class files, 1, 4, 6, 11–12, 20, 187, 191–236
access flags in, 196–197
arrays in, 210
ArrayType descriptors in, 203
attributes in, 201, 215, 218–229
BaseType descriptors in, 203, **204**
big-endian byte order in, 11, 146, 193
binary compatibility and, 58–56, 121
binary files and, 11
bytecode verification with, 54–56
class reference in, 150–151, 209–210
code attribute in, 220–223
compactness of, 121, 122, 178
constant pool in, 195–201, 196, 205–214, 270
CONSTANT_Class_info table for, 209–210, 278–287
CONSTANT_Double_info table for, 209
CONSTANT_Fieldref_info table for, 211, 287–288

CONSTANT_Float_info table, 208
CONSTANT_Integer_info table for, 207
CONSTANT_InterfaceMethodref_info table for, 212–213, 289–290
CONSTANT_Long_info table for, 208–209
CONSTANT_Methodref_info table for, 212, 288–289
CONSTANT_NameAndType_info table for, 213–214
CONSTANT_String_info table for, 210, 211
CONSTANT_Utf8_info table for, 205–207, **206**
ConstantValue attribute, 223, 224
contents and components of, 193–201
deprecated attributed, 224, 225
descriptors in, 20, 201, 202–204
double integers in, 209
dynamic linking and, 196
exception_info table for, 222
exceptions attribute, 225
field descriptors in, 196, 200, 203, **204**
fields in, 200, 211, 214–215
FieldType descriptors, 202–203
floating-point numbers in, 208
fully qualified names in, 201, 202
Getting Loaded applet for, 191, 234–236, **235, 236**
Halting Problem and, 55–56
inner_class_info table for, 228–229
InnerClasses attribute, 225–229
integer types in, 207
interfaces in, 199–200, 212–213
language of Java and, 192, **192**
line_number_info table for, 230–231
LineNumberTable attribute, 229–231, 229
little-endian byte order in, 11, 146, 193
local_variable_info table for, 232–233
LocalVariableTable attribute, 231–233
long integers in, 208–209
magic number in, 194
major version numbers in, 195
method descriptors in, 196, 200–201, 203, **204**
MethodDescriptors in, 203
methods in, 212, 216–218
minor version numbers in, 195
mobility concepts and, 11–12, 120, 121, 132
names in, 213–214
obfuscation of, 21
ObjectType descriptors in, 203
platform independence and, 26
primitive types in, 193–194
referencing, 20
relationships within, 192, **192**
resolution of other CONSTANT_info entries, 292
ReturnDescriptors in, 203
semantic checks, class file verifier and, 54
simple names in, 201, 202
SourceFile attribute, 233
special strings in, 201–204
strings in, 210, 211
structural checks on, class file verifier and, 53–54
superclasses in, 151 198–199, 242, 243
superinterfaces in, 199–200
symbolic reference verification in, 56–58
symbolic references in, 20, 56–58, 153, 196, 201, 202–204, 209–210, 211

Synthetic attribute, 234
version numbers in, 195
class initialization method, Java types, 246–251
class instantiation, 254–264
class loaders, 6, **6**, 8–11, **9**, 136, **137**, 142–146, 149–150, 188
access control in, 50–51
applet class loaders, 10
applets and, 125
bootstrap, 47–49, 142, 143–144, 188, 241, 265–268
built-in, 48–50
current, 278
defineClass( ) method for, 144–145
defining, 277
dynamic extension in, 10
findSystemClass( ) method for, 145
forbidden packages/types and, 51–52
fully qualified names in, 145
initialization operations in, 143, 277
Java types, 241
linking and, 143, 269, 276–277
loading constraints and, 292–294, 343–353
loading of Java types with, 280–285
loading operations in, 143
malicious versus trusted classes and, 50–51
mobility concepts and, 132
name spaces in, 10–11, 45, 46, **46**, 142, 145–146
parent-delegation model for, 47–49, 276–277
primordial, 47–49
protection domains in, 145
referenced versus referencing class, 10
resolveClass( ) method for, 145
runtime packages and, 51
security and, 44, 45–52, 66, 343–353
system, 48–50, 144
unloading of types and, 265–268, 336–343
user-defined, 47, 48–50, 142–143, 144–145, 241, 265–268, 323–333
user-defined, version 1.1, 323–329
user-defined, version 1.2, 329–333
classes, 187
clone( ) method for, 254
getObject ( ) method for, 254
inner_class_info table for, 228–229
InnerClasses attribute of, 225–229
instantiation of, 254–264
linking and, 269, 279–287
newInstance( ) method for, 254–255
resolution and, 279–287
subclasses, 243
superclasses, 242, 243
client/server applications
mobility concepts and, 114–115
service objects and, 129–131, **130**
<clinit> ( ) method for initialization of Java types, 247–251
clipboard, security and, 66
clone( ) method, 254
code attribute as, 220–223
code signing, 43, 68–75
example of, 75–79
hash algorithms and, 69
JAR files for, 69–73, **71**, **72**, 77–79
code source
policies and, 80
security and, 64–65, 80
compacting garbage collection, 359–360, 383–384
compacting heap, 383–384
compactness of class files, 121, 122, 178

compilers, 6, 18
 ahead-of-time compilers, 18
compile-time resolution of constants, 294–296
computation types, 180
conditional branching in, 450–452
conditional branching with tables in, 453–455
conservative style garbage collection, 358
constant pool in, 148, 152–153, 168, 195–201, **196**, 205–214, 243, 270
 compile-time resolution of constants, 294–296
 CONSTANT_Class_info table in, 209–210, 278–287
 CONSTANT_Double_info table in, 209
 CONSTANT_Fieldref_info table in, 211, 287–288
 CONSTANT_Float_info table, 208
 CONSTANT_Integer_info table for, 207
 CONSTANT_InterfaceMethodref_info table in, 212–213, 289–290
 CONSTANT_Long_info table in, 208–209
 CONSTANT_Methodref_info table in, 212, 288–289
 CONSTANT_NameAndType_info table in, 213–214
 CONSTANT_String_info table in, 210, 211, 290–292
 CONSTANT_Utf8_info table in, 205–207, **206**
 ConstantValue attribute, 223, 224
 current, 162
 direct references, 296–305, **300, 301, 302, 303**
 method tables, 300–303, **300, 301, 302, 303**
 resolution of, 270, 277–278
 resolution of CONSTANT_Class_info entries, 278–287
 resolution of CONSTANT_Fieldref_info entries, 287–288
 resolution of CONSTANT_InterfaceMethodref_info entries, 289–290
 resolution of CONSTANT_Methodref_info entries, 288–289
 resolution of CONSTANT_String_info entries, 290–292
 resolution of other CONSTANT_info entries, 292
 runtime constant pool, 270
constant variables, 149
CONSTANT_Class_info table, 209–210
 resolution of, 278–287
CONSTANT_Double_info table for, 209
CONSTANT_Fieldref_info, 211
 resolution and, 287–288
CONSTANT_Float_info table, 208
CONSTANT_Integer_info table for, 207
CONSTANT_InterfaceMethodref_info, 212–213
 resolution and, 289–290
CONSTANT_Long_info table for, 208–209
CONSTANT_Methodref_info, 212
 resolution and, 288–289
CONSTANT_Methodref_info table for, 212
CONSTANT_NameAndType_info table for, 213–214
CONSTANT_String_info, 210, 211
 resolution and, 290–292
CONSTANT_Utf8_info table for, 205–207, **206**
ConstantValue attribute, 223, 224
constraints, loading constraints, resolution and, 292–294, 343–353
content services, mobility concepts and, 117–119

control flow, 449–458
branching in, 449
conditional branching in, 450–452
conditional branching with tables in, 453–455
jumps in, 449, 453–455
resource Web page for, 458
Saying Tomato applet for, 449, 455–457, **455**
Three-Dimensional Arrays applet for, 437
unconditional branching in, 453
Conversion Diversion applet, 399, 402–405, **403**
conversion of types, 399–405
cooperation in synchronization, 498, 499
coordination in synchronization, 511
copying garbage collection, 360–361, **361**
current class loader, 278
current constant pool, 162
current frame, 162
current method, 162
current namespace, 278

D

dadd opcode, 432
daemon threads, virtual machine and, 135–136
daload opcode, 443
dastore opcode, 444
data flow analyzer, 244
data types (*see* Java types)
dcmpg/dcmpl opcode, 451
dconst_*xxx* opcodes, 387
ddiv opcode, 433
defineClass( ) method, 144–145
defining class loader, 277
Delete operations, security and, 66
denial-of-service attacks, 109
deployment of Java Platform, platform independence and, 29
deprecated attributed, 224, 225
descriptors, 20, 201, 202–204
design of Java, 2
digital certificates, 74–75
digital signature, 43, 68–75
direct references, 296–305, **300, 301, 302, 303**
discovery protocol for Jini technology and, 127
disruptive algorithms for garbage collection, 363
distributed applications, distributed processing, 3, 17–19
mobility concepts and, 115, 116–118
platform independence and, 24, 29, 35–36
service objects and, 129–131, **130**
dload, dload_*xxx* opcodes, 391
dmul opcode, 432
domains, protection domains, 65, 84–86, **86**
doPrivileged( ) method, 100–109
double integers in, 209
double value set for floating-point arithmetic, 429–430
double-extended-exponent value set for floating-point arithmetic, 430
download time and processing, mobility concepts and, 121–122
downloading files, security and, 42
drem opcode, 433
dreturn opcode, 495
dstore, dstore_*xxx* opcodes, 393
dsub opcode, 432

dup, dup_*xxx* opcodes, 390
dynamic (late) binding, 157, 480
dynamic extension, 10, 14, 272–275, 318–323
forName ( ) and, 333–336
Greet Application of, 318–323
mobility concepts and, 121, 122
resolution and, 272–275, 318–323
dynamic linking, 14, 57, 242
applets and, 125
class files, 196
method area and, 150
mobility concepts and, 120–121, 122
name spaces, 150
resolution and, 270–272

E

early resolution and, 271
embedded devices
Java Embedded Platform, 27
platform independence and, 24, 26, 27, 28, 39–40
encryption, 69–75
keystore files, 78, 82
entering the monitor for synchronization, 498
entry sets in synchronization, 498, 503
error handling, 14
security and, 62
eteral math simulation, 188–189, **190**
event queue, security and, 66
exception table, 168, 464–465
exception_info table, 222
exceptions, 459–468
catching, 460–464
exception table for, 464–465
Play Ball simulation applet for, 459, 465–468, **466**
resource Web page for, 468
throwing, 460–464
exceptions attribute, 225
execution engine, 6, 136, **137**
adaptive optimization in, 181–183
computation types, 180
data types in, 178–180
execution techniques for, 181–183
inlining in, 182–183
instruction set in, 174–181
opcodes in, 174–181
operands in, 174–181
storage types in, 180
synchronization and, 184–186
threading in, 183–186
exponent of floating-point arithmetic, 424–427

F

fadd opcode, 432
faload opcode, 443
fastore opcode, 444
fcmpg/fcmpl opcode, 451
fdiv opcode, 433
Fibonacci Forever applet, 386, 394–397, **395**
field descriptors, class files, 196, 200, 203, **204**
field information for data types, 148
fields, 200, 214–215, 243
linking and, 287–288
resolution and, 287–288
FieldType descriptors, 202–203

finalization of objects, 160–161, 264–265, 368–370
finally clauses, 469–478
  asymmetrical invocation of, 471–474
  Hop Around applet for, 469, 474–477, **474**
  miniature subroutine action of, 470–471
  opcodes for, 470
  resource Web page for, 478
  return in, 471–474
findSystemClass( ) method, 145
first-in first-out (FIFO) entry set for synchronization, 503
fload, fload_*xxx* opcodes, 391
float value set for floating-point arithmetic, 429–430
float-extended-exponent value set for floating-point arithmetic, 430
floating-point arithmetic, 140, 208, 423–436
  Circle of Squares applet for, 423, 434–436, **435**
  conversion of value sets in, 430–431
  defining floating-point numbers for, 424
  double value set for, 429–430
  double-extended-exponent value set for, 430
  exponent of, 424–427
  float value set for, 429–430
  float-extended-exponent value set for, 430
  FP-default mode for, 429
  FP-strict mode for, 429
  Inner Float applet for, 423, 427, **428**
  mantissa of, 424–427
  modes for, 428–431
  normalization in, 424–427
  not a number (NaN) value in, 425, 426
  opcodes for, 431–434
  radix of, 424–427
  relaxing rules for, 431
  resources Web page for, 436
  sign of, 424–427
  value sets for, 429–431
floating-point numbers in, 140, 208
fmul opcode, 432
forbidden packages/types and, 51–52
forName ( ) dynamic extension and, 333–336
FP-default mode for floating-point arithmetic, 429
FP-strict mode for floating-point arithmetic, 429
fragmentation, 356
frames (*see also* Java stack; stack frame), 55, 138, 142
  current, 162
  frame data in, 167–168
frem opcode, 433
freturn opcode, 495
fsub opcode, 432
fully qualified names in, 145, 147, 201, 202

G

garbage collection, 14, 15–16, 19, 154–155, 160–161, 188, 264, 355–384, 438
  adaptive, 362–363
  algorithms for, 357–358
  caches for, 376–378
  canonicializing mappings for, 376–378
  changes of reachability states and, 373–376
  compacting, 359–360, 383–384
  conservative style, 358
  copying, 360–361, **361**
  disruptive algorithms for, 363
  finalization of objects and, 160–161, 264–265, 368–370

fragmentation and, 356
freeing memory using, 356
generational, 362, 364
heap area in memory for, 60, 438
Heap of Fish applet for, 355, 378–384, **380, 381, 383, 384**
incremental, 363
integrity of programs through, 356–357
mark and sweep algorithm, 160
phantom reachable states and, 371–373, **372**, 375–376, 377–378
platform independence and, 33, 35
popular objects and, 367, 368
pre-mortem cleanup and, 376–378
reachability for, 357, 370–378
reachable state and, 370
reference counting and, 358–359
reference objects and, 371–373, **372**
reference queues and, 373, **374**
remembered sets and, 367–368
resource Web page for, 384
resurrectable state and, 370, 375
roots and root sets, 357–358
softly reachable states and, 371–373, **372**, 375, 376–378
stop-and-copy, 360–361, **361**
strongly reachable states and, 374
tracing, 358, 359
train algorithm for, 363–368, **365**
unloading of types and, 265–268
unreachable state and, 370, 375
weakly reachable states and, 371–373, **372**, 375, 376–378
generational garbage collection, 362, 364
getfield opcode, 439, 440
getObject ( ) method, 254
getstatic opcode, 439, 440
Getting Loaded applet, 191, 234–236, **235, 236**
goto opcodes, 453
grant clauses, 82–84, 82
graphical user interface (GUI), platform independence and, 12, 34, 36
Greet Application
dynamic extension of, 318–323
dynamic extension of, version 1.1 user-defined class loader and, 323–329
dynamic extension of, version 1.2 user-defined class loader and, 329–333
unloading Java types, 336–343

H

Halting Problem, 55–56
handle pool in, 156, **156**
hash algorithms and, 69
heap, 137, 153–161
allocating objects to, 379–380, **380**
arrays in, 161, **162**
assigning references to objects in, 381–382, **381**
compacting of, 383–384
dynamic binding and, 157
fragmentation and, 356
garbage collection in, 154–155, 160–161, 355–384, 382–383, **383**, 438
handle pool in, 156, **156**
Heap of Fish applet for, 355, 378–384, **380, 381, 383, 384**
Java stack and, 168–171, **169, 171**
locks in, 497, 503–505
mark and sweep algorithm, 160
method tables for, 157–158
mutex (*see* locks)

object pool in, 156, **156**
object representation in, 155–161
pointers in, 155, 156, **157**
referencing in, 155–161
wait sets for, 159–160
Heap of Fish applet, 355, 378–384, **380, 381, 383, 384**
Hop Around applet, 469, 474–477, **474**
HTML, applet tags, 124
hypertext transport protocol (HTTP), 10

I

i2*xxx* conversion opcodes, 400
iadd opcode, 410
iaload opcode, 443
iand opcode, 418
iastore opcode, 444
iconst_*xxx* opcodes, 387
idiv opcode, 411
if_acmpxx opcodes, 452
if_icmp*xxx* conditional branching opcodes, 451
ifnonnull opcode, 452
ifnull opcode, 452
ifxx conditional branching opcodes, 450
iinc opcode, 410
iload, iload_*xxx* opcodes, 391
implies( ) method, 87–89
imul opcode, 411
incremental garbage collection, 363
ineg opcode, 413
<init> ( ) method, 258–264
invokespecial instruction and, 483–486
initialization of Java types, 238–240, **238**, 245–253
initialization of classes, 154
initialization operations in class loaders, 143, 277
inlining in, 182–183
Inner Float applet, 423, 427, **428**
Inner Int applet, 407, 409, **409**
inner_class_info table for, 228–229
InnerClasses attribute, 225–229
instance initialization method for, 258–264
instanceof opcode, 440
instruction set, 174–181, 385
mnemonic for, 176, 179
_quick instructions, 305–306
integer arithmetic, 407–416
Inner Int applet for, 407, 409, **409**
opcodes for, 409–412
Prime Time simulation applet for, 412–416, **412**
resources Web page for, 416
two's complement numbers in, 407, 408–409
integers, 207
integral data types, 140
interfaces, 141, 199–200, 212–213
initialization method, Java types, 246–251
invokeinterface instruction for, 490
linking and, 269, 289–290
resolution and, 289–290
interpreter, 7
invocation of methods, 479–496
dynamic (late) binding in, 480
examples of, 491–495
<init>( ) and invokespecial instruction in, 483–486
instance versus class methods, 480
invokeinterface instruction for, 490
Java method invocation, 481–482

native method invocation, 482
opcodes for, 480
private methods and invokespecial instruction in, 486–487
resource Web page for, 496
returning from methods in, 495
special invocation (invokespecial instruction), 482–489
speed of processing and, 490
static (early) binding in, 480
super keyword and invokespecial instruction in, 487–489
invokeinterface instruction for, 490
invokespecial instruction, 482–489
<init>( ) method and, 483–486
private methods and, 486–487
super keyword and invokespecial instruction in, 487–489
invokestatic opcode, 480
invokevirtual opcode, 480
ior opcode, 418
IP address and Jini technology and, 127
irem opcode, 413
ireturn opcode, 495
ishl/ishr opcodes, 418
istore, istore_*xxx* opcodes, 392
isub opcode, 411
iushr opcode, 418
ixor opcode, 418

J

JAR files
code signing and, 69–73, **71, 72**, 77–79
keystore files and, 78
mobility concepts and, 122
Java 2 Enterprise Edition (J2EE) and, 28–29
Java 2 Platform, Micro Edition (J2ME) and, 28–29
Java 2 Standard Edition (J2SE), 28–29
platform independence and, 30–31
Java API, security and, 63–64, 67–58
security manager and, 63–64, 63
Java applets (*see* applets)
Java Archive (JAR) files (*see* JAR files)
Java Authentication and Authorization Service (JAAS), 109–110
Java Card Platform, platform independence and, 27
Java Embedded Platform, platform independence and, 27
Java interpreter and, 7
Java methods, 7
Java Native Interface (JNI) (*see* Jini technology)
Java Personal Platform, platform independence and, 27
Java stack (*see also* operand stack; stack frames), 55, 137–138, 142, 162–163
abrupt completion of methods in, 163
constant pool in, 168, 195–201, **196**, 205–214, 270
current constant pool, 162
current frame, 162
current method in, 162
exception tables in, 168
Fibonacci Forever applet for, 386, 394–397, **395**
frame data in, 167–168
generic operations for, 389, 390
implementation of, 168–171, **169, 171**
local variables and, 163–166, **164**
memory allocation in, 163

native method stacks in, 172–173, **173**
normal completion of methods in, 163
objects versus, 438
operand stack and, 166–167, **167**, 171, 177, 179
popping to local variables in, 390–392
pushing constants onto operand stack, 386–389
pushing local variables onto operand stack, 389–390
stack frame in, 163–171
wide instruction for, 392–394
Java types, 139–141, **140**, 178–180, 237–268
active use of, 239, 251–253
array data type in, 141
binary data for, 240–241
class data type, 141
class initialization method, 246–251
class instantiation and, 254–264
class loader and, 241
class loader reference in, 149–150
classlife directory for, on companion disk, 268
<clinit> ( ) method for initialization of, 247–251
computation types, 180
constant pool in, 148, 152–153
constant variables in, 149
Conversion Diversion applet for, 399, 402–405, **403**
conversion of, 399–405
double integers in, 209
field information for types in, 148
finalization of objects and, 264–265
floating-point data types, 140, 208
fully quantified names in, 147
garbage collection and, 264
initial values for, 244–245
initialization of, 238–240, **238**, 245–253
instance initialization method for, 258–264
integers, 140, 207
interface data type, 141
interface initialization method, 246–251
lifetime of, 237–268
linking of, 238–240, **238**
loading of, 238–240, **238**, 240–241, 280–285
long integers in, 208–209
memory allocation for, 244–245
method information for data in, 148–149
numeric data types, 140
object lifetime and, 253–265
passive use of, 239, 251–253
preparation of, 238–240, **238**, 244–245
primitive values/primitive data types, 139–141, 193–194, 244–245, 438
reference values/ reference data types, 139–141
resolution of, 238–240, **238**, 245
resource Web page for, 268
security and, 343–353
semantic checks, class file verifier and, 54
simple names in, 147
storage types in, 180
strings in, 210, 211
unloading of, 265–268
verification of, 238–240, **238**, 241–244
Java Virtual Machine Specification, The, 134
Java/Java 2 Platform, 4–5, 5
platform independence and, 25
Jini service object, 125–131, 125
Jini technology, 186
attributes of service objects in, 128
benefits of service object use in, 129–131

discovery protocol for, 127
IP address and, 127
join protocol for, 127–128
lookup protocol for, 127, 128–129
lookup services, 126, 127
mobility concepts and, 125–131
native methods and, 186
platform independence and, 24, 39–40
presence announcement in, 127
runtime infrastructure of, 126–127
security and, 42, 109
service IDs in, 128
service items in, 127–128
service objects and, 128, 129–131
service registrar for, 127
service templates in, 128
services provided by, 126
join protocol for Jini technology and, 127–128
jsr, jsr_w opcode, 470
jumps, 449–458, 453–455
just-in-time compiler of, 6

K

keystore filess, 78, 82
keytool program, 81

L

ladd opcode, 410
laload opcode, 443
land opcode, 419
last-in first-out (LIFO) entry set for synchronization, 503
lastore opcode, 444
late resolution and, 271
layout managers, 13
lcomp opcode, 451
lconst_*xxx* opcodes, 387
ldc, ldc_*xxx* opcodes, 389
ldiv opcode, 411
lifetime of type (*see* Java types)
Lindholm, Tim, 134
line_number_info table for, 230–231
LineNumberTable attribute, 229–231
linking, 57, 238–240, **238**, 269–353
arrays, 278–279
class loaders and, 143, 276–277, 323–329, 329–333
classes and, 279–287
compile-time resolution of constants and, 294–296
constant pool resolution and, 277–278
direct references and, 296–305, **300, 301, 302, 303**
dynamic extension and, 272–275, 318–323
dynamic linking and, 270–272
early resolution and, 271
fields and, 287–288
forName ( ) dynamic extension and, 333–336
Greet Application and, dynamic extension of, 318–323
interfaces and, 289–290
late resolution and, 271
loading constraints and, 292–294, 343–353
loading of Java types for, 280–285
method tables and, 300–303, **300, 301, 302, 303**
methods and, 288–289
parent-delegation model for, 276–277

_quick instructions for, 305–306
resolution and, 270–272
resolution of CONSTANT_Class_info entries, 278–287
resolution of CONSTANT_Fieldref_info entries, 287–288
resolution of CONSTANT_InterfaceMethodref_info entries, 289–290
resolution of CONSTANT_Methodref_info entries, 288–289
resolution of CONSTANT_String_info entries, 290–292
resolution of other CONSTANT_info entries, 292
resources Web page for, 353
runtime constant pool, 270
Salutation application example of, 306–318, **311, 313, 314, 316, 317**
strings and, 290–292
type safety and, 343–353, 343
unloading Java types and, 336–343
little-endian byte order in class files, 11, 146, 193
lload, lload_*xxx* opcodes, 391
lmul opcode, 411
lneg opcode, 413
loading constraints, linking and, 292–294, 343–353
loading operations (*see also* class loaders), 143, 238–240, **238**, 240–241, 269
local variables, 142, 163–166, **164**, 386
arrays versus, 438
Fibonacci Forever applet demonstrating, 386, 394–397, **395**
LocalVariableTable attribute, 231–233
popping to local variables in operand stack, 390–392
pushing local variables onto operand stack, 389–390
wide instruction for, 392–394
local_variable_info table for, 232–233
LocalVariableTable attribute, 231–233
locking/unlocking, 158–159
synchronization, 497, 503–505
logic, 417–422
Logical Results applet for, 417, 419–421, **419**
opcodes for, 418–419
resources Web page for, 422
Logical Results applet, 417, 419–421, **419**
login security, 109
long integers in, 208–209
lookup protocol, Jini technology and, 127, 128–129
lookup services, Jini technology and, 126, 127
lookupswitch opcode, 454
lor opcode, 419
lrem opcode, 413
lreturn opcode, 495
lshl/lshr opcodes, 418
lstore, lstore_*xxx* opcodes, 393
lsub opcode, 411
lushr opcode, 418
lxor opcode, 419

M

magic number, 194
main ( ) method, virtual machine invocation through, 134–135, 152
main memory, threading, 184
malicious versus trusted classes and, 50–51

mantissa of floating-point arithmetic, 424–427
Marimba Castanet, 122
mark and sweep algorithm, 160
memory (*see also* garbage collection; heap), 138
  access control to, 60
  corrupted memory, 16
  freeing memory, 16
  Java frame and, 163, 168–171, **169, 171**
  Java types allocation and, 244–245
  leaks in memory, 16
  method areas, 60
  native method stacks, 172–173, **173**
  runtime data areas in, 60
  security and, 60, 109
  stack frame and, 163
  threading in, 184
method area, 60, 137
  big-endian byte order in class files, 146
  class class reference in, 150–151
  class loader reference in, 149–150
  constant pool in, 148, 152–153
  constant variables in, 149
  data types in, 147
  dynamic linking in, 150
  field information for types in, 148
  fully quantified names in, 147
  little-endian byte order in class files, 146
  method information for data in, 148–149
  method tables in, 151
  name spaces in, 150
  resolution in, constant pool, 153
  simple names in, 147
  symbolic references in, 153
  threading, 147
  type information, 147
  use of, example, 151–154
  virtual machine and, 146–154
method descriptors in class files, 196, 200–201
method descriptors, 203, **204**
method information for data, 148–149
method tables, 151, 157–158, 300–303, **300, 301, 302, 303**
  virtual table (VTBL) versus, 158
MethodDescriptors in, 203
methods, 7, 216–218, 243
  abrupt completion of methods in, 163
  adaptive optimization in, 181–183
  current method, 162
  dynamic (late) binding in, 480
  <init>( ) and invokespecial instruction in, 483–486
  inlining in, 182–183
  instance versus class methods, invocation of, 480
  invocation and return in, 479–496
  invokeinterface instruction for, 490
  Java method invocation, 481–482
  Java stack operations and, 162–163
  linking and, 288–289
  native method invocation for, 482
  normal completion of methods in, 163
  opcodes for invocation of, 480
  private methods and invokespecial instruction in, 486–487
  remote method invocation (RMI), 3, 25, 125, 131
  resolution and, 288–289
  returning from methods in, 495
  signature of, 485
  special invocation (invokespecial instruction) and, 482–489

static (early) binding in, 480
super keyword and invokespecial instruction in, 487–489
synchronized methods as, 504, 508–511
Microsoft, platform independence and versus., 38–39
miniature subroutine action of finally clauses, 470–471
mnemonic for opcodes, 176, 179
mobility of code and objects in Java, 3, 4, 113–132
administration across distributed systems and, 115–116
applets and, 122–125, **124**
applications for, 114–116
architectural support for, 120–122
bandwidth and, 116
binary compatibility and, 121
bootstrap class loaders and, 132
central processing unit (CPU) and, 114
class files and, 120, 121, 132
class loaders and, 132
client/server applications and, 114–115
compactness of class files, 121, 122, 178
content services and 117–119
distributed processing and, 115, 116–118
download time and processing in, 121–122
dynamic extensions and, 121, 122
dynamic linking and, 120–121, 122
Java Archive (JAR) files and, 122
Jini service object and, 125–131
Jini technology, 125–131
namespaces and, 132
platform independence and, 25, 39–40, 120
resources Web page for, 132
security and, 120, 132
security manager and, 132
service objects and, 125–131
shared applications/files and, 114
software development paradigms and, 119
user-defined class loaders and, 132
modes for floating-point arithmetic, 428–431
monitor for synchronization, 497, 498–503
monitorenter opcode, 506
monitorexit opcode, 506
multianewarray opcode, 442
multicast security, 66
multi-dimensional arrays, 161
multi-thread programming, 14
mutex (*see* locks)
mutual exclusion in synchronization, 498, 499

N

name spaces, 10–11
class loaders and, 142, 145–146
current, 278
dynamic linking, 150
method area and, 150
mobility concepts and, 132
security, 45–46, **46**
virtual machine and, 150
naming conventions
fully qualified names in, 147, 201, 202
simple names in, 201, 202
native method stack, 138, 172–173, **173**
native methods, 7, 186–187
invocation of, 482
Jini technology for, 186
native method stacks for, 138, 172–173, **173**
platform independence and, 31–32, 35
run-time libraries and, 32–33
security and versus, 61–62, 66

network-mobile objects (*see* mobility)
networks and Java, 2–4
new opcode, 439
newarray opcode, 442
newInstance( ) method, 254–255
non-daemon threads, virtual machine and, 135–136
non-standard run-time libraries, platform independence and, 32–33, 35
nop opcode, 390
normal completion of methods in, 163
normalization floating-point arithmetic, 424–427
not a number (NaN) value in floating-point arithmetic, 425, 426
notify command for synchronization, 499–500, 502–503, 512
null reference checking, 60
numeric data types, 140

O

obfuscation of class files, 21
object-oriented programming, 14, 15
object pool in, 156, **156**
objects, 253–265, 437–447
  finalization of objects and, 368–370
  garbage collection and, 264
  <init> ( ) method, 258–264
  instance initialization method for, 258–264
  Java stack versus, 438
  opcodes for, 438–440
  reachability and garbage collection and, 370–378
ObjectType descriptors in, 203
op codes, 55, 174–181
  _quick instructions, 305–306
  array, 440–443
  conditional branching, 450–452
  conversion, 400–402
  finally clause, 470
  floating-point arithmetic, 431–434
  generic stack operations with, 389, 390
  integer arithmetic, 409–412
  jump, 454
  logic, 418–419
  method invocation, 480
  mnemonic for, 176, 179
  object, 438–440
  pushing constants onto stack with, 386–389
  returning from methods, 495
  unconditional branching, 453
operand stack, 55, 142, 166–167, **167**, 171, 177, 179, 386
  Fibonacci Forever applet for, 386, 394–397, **395**
  generic operations for, 389, 390
  popping to local variables in, 390–392
  pushing constants onto, 386–389
  pushing local variables onto, 389–390
  resources Web page for, 397
  string literals and, 389
  wide instruction for, 392–394
operands, 55, 174–181
OR (*see* logic)
owning the monitor for synchronization, 498

P

parent-delegation model for class loaders, 47–49, 276–277
passive use of Java types, 239, 251–253

password security, 109
PC register (*see* program counter)
permissions, 64–65, 67–68, 81
  Access Controller, 86–109
  checkPermission( ) method, 87
  implies( ) method, 87–89
phantom reachable states and garbage collection, 371–373, **372**, 375–376, 377–378
platform independence, 2, 4, 7, 11–12, 19–20, 23–40
  100 percent Pure Java certification and, 36
  Abstract Windows Toolkit (AWT) and, 12–13
  applets and, 123
  architectural support for, 25–29
  bugs in Java implementations and, 34
  class files and, 26
  deployment of Java Platform and, 29
  disadvantages of, 24–25
  distributed applications and, 24, 29, 35–36
  embedded devices and, 24, 26, 27, 28, 39–40
  factors influencing, 29
  garbage collection and, 33, 35
  graphical user interface (GUI) and, 12, 36
  Java 2 Enterprise Edition (J2EE) and, 28–29
  Java 2 Platform, Micro Edition (J2ME) and, 28–29
  Java 2 Standard Edition (J2SE), 28–29, 30–31
  Java Card Platform and, 27
  Java Embedded Platform and, 27
  Java Personal Platform and, 27
  Java/Java 2 Platform and, 25
  Jini technology and, 24, 39–40
  Microsoft versus, 38–39
  mobility and, 25, 39–40, 120
  native methods and, 31–32, 35
  non-standard run-time libraries and, 32–33, 35
  porting applications and, 32
  programming language and, 26
  remote method invocation (RMI) and, 25
  resource Web site for, 40
  run-time libraries and, 32–33, 35
  scalability and, 26–29
  Standard API and, 30–31
  Standard Extension APIs and, 30–31
  standardization versus, 36–39
  steps to, 35–36
  Swing and, 12–13
  testing and, 34–35, 36
  threading and, 33–34, 35
  user interface dependencies and, 34
  vendors versus, politics of, 36–39
  version and edition differences and, 30–31
  virtual machine dependencies and, 33–34
Play Ball simulation applet, 459, 465–468, **466**
pointers (*see also* referencing), 15, 155, 156, **157**
policies and policy files, 44, 64, 82–84
  code source and, 80
  security and, 79–84
pop opcode, 390
pop2 opcode, 390
popping to local variables in operand stack, 390–392
popular objects and garbage collection, 367, 368
port number security, 65, 66
porting applications, platform independence and, 32
pre-mortem cleanup and garbage collection, 376–378

preparation of Java types, 238–240, **238**, 244–245
presence announcement in Jini technology and, 127
Prime Time prime number simulation applet, 412–416, **412**
primitive values/primitive data types, 139–141, 193–194, 244–245, 438
primordial class loaders, 47–49
printing security, 66
private methods and invokespecial instruction in, 486–487
program counter, 137–138, 161–162
programming language for Java, 4, 14–16
  class files, 20, 192
  platform independence and, 26
protection domains, 65, 84–86, **86**, 145
public/private key, 68–75
  keystore files for, 78, 82
pushing constants onto operand stack, 386–389
pushing local variables onto operand stack, 389–390
putfield opcode, 439, 440
putstatic opcode, 439, 440

Q

_quick instructions, 305–306

R

radix of floating-point arithmetic, 424–427
reachability for garbage collection, 357, 370–378
reachable state and garbage collection, 370
Read operation security, 66
reference counting garbage collection, 358–359
reference objects and garbage collection, 371–373, **372**
reference queues and garbage collection, 373, **374**
reference values/ reference data types, 139–141
referenced versus referencing class, 10
referencing, 15
  assigning references to objects in heap, 381–382, **381**
  checking, 16
  class class reference in, 150–151
  class file, 20
  direct references, 296–305, **300, 301, 302, 303**
  dynamic linking and, 57
  heap and, 155–161
  null reference checking in, 16, 60
  security and, 60
  symbolic reference in, 20, 56–58, 153, 196, 201, 202–204, 209–210, 211, 242–243, 279–280
  type-safe reference casting and, 60
reflection API, security and, 66
regions for monitor in synchronization, 498
releasing the monitor for synchronization, 498
remembered sets and garbage collection, 367–368
remote method invocation (RMI), 3, 25, 125, 131
resolution, 269
  arrays, 278–279

classes, 279–287
compile-time resolution of constants, 294–296
constant pool, 153, 270, 277–278
CONSTANT_Fieldref_info entries, 287–288
direct references, 296–305, **300, 301, 302, 303**
dynamic extension and, 272–275, 318–323
dynamic linking and, 270–272
early resolution and, 271
field, 287–288
interface, 289–290
Java types, 245, 238–240, **238**
late resolution and, 271
loading constraints, 292–294, 343–353
loading of Java types and, 280–285
method tables and, 300–303, **300, 301, 302, 303**
method, 288–289
resolution of CONSTANT_Methodref_info entries and, 288–289
resolution of other CONSTANT_info entries and, 292
runtime constant pool and, 270
string, 290–292
symbolic reference and, 279–280
resolveClass( ) method, 145
resurrectable state and garbage collection, 370, 375
ret opcode, 470
return in finally clauses, 471–474
return opcode, 495
ReturnDescriptors in, 203
returning from methods, 495
robustness of Java, 3
robustness of programs, class file verifier and, 52–53
roots and root sets garbage collection, 357–358
runtime constant pool, 270
runtime data areas, 60, 136, **137,** 137–138, **138, 139**
runtime infrastructure, Jini technology and, 126–127
run-time instance of virtual machine and, 134–136
run-time libraries
native methods and, 32–33
non-standard, platform independence and, 32–33, 32
platform independence and, 32–33, 35
runtime packages, 51

S

saload opcode, 443
Salutation application example of linking and, 306–318, **311, 313, 314, 316, 317**
sandbox security (*see also* class file verifier; security), 42–45
class file verifier, 52–59
class loader in, 44, 45–52
namespace security in, 45–46, **46**
security manager and, 44, 63
sastore opcode, 444
Saying Tomato applet, 449, 455–457, **455**
scaleability, platform independence and, 26–29
security (*see also* access control), 3, 4, 13–14, 41–111
Abstract Windows Toolkit (AWT), 66
access control and, 44–45, 50–51, 62–68, 80, 86–109

Access Controller for, 14, 86–109
ActiveX and, 79
applets, 123
array bounds checking and, 60
authentication and, 43, 68–75
binary compatibility and, 58–56
bugs and, 42, 44, 79–80
built-in safety features of virtual machine for, 59–62
bytecode verification, class file verifier, 54–56
certificate authorities (CA) and, 74–75, 80–81
check methods for security manager in, 65–68
class file verifier in sandbox and, 44, 52–59
class loader in sandbox and, 44, 45–52
class loaders and, 66, 343–353
clipboard, 66
code signing and, 43, 68–75, 75–79
code sources and, 64–65, 80
Delete operations, 66
denial-of-service attacks and, 109
digital certificates for, 74–75
digital signatures for, 43, 68–75
doPrivileged( ) method for, 100–109
downloading files and, 42
encryption and, 69–75
error handling and, 62
event queue, 66
future of, 109–110
garbage collection and, 60
grant clauses, 82–84
Halting Problem and, 55–56
hash algorithms and, 69
implementing special security in, 110–111
JAR files and code signing in, 69–73, **71, 72,** 77–79
Java API and, 44, 63–64, 67–58
Java applets and, 42
Java Authentication and Authorization Service (JAAS), 109–110
Java types and, 343–353
Jini technology and, 42, 109
keystore files, 78, 82
keytool program, 81
loading constraints and, 343–353
login, 109
malicious versus trusted classes and, 50–51
memory and, 60, 109
mobility concepts and, 120, 132
multicasts, 66
namespace security in, 45–46, **46**
native methods versus, 61–62, 66
null reference checking, 60
passwords, 109
permissions and, 64–65, 67–68, 81
policies and policy files for, 44, 64, 79–84
port numbers and, 65, 66
printing, 66
protection domains and, 65, 84–86, **86**
public/private keys, 68–75
Read operations, 66
referencing and, 60
reflection API, 66
resources Web page on, 111
sandbox for, 42–45
security manager and, 14, 44, 62–68
semantic checks, class file verifier and, 54
sockets and, 65, 66
stack inspection by Access Controller, examples of "yes" and "no," 90–99

structural class-file checks, class file verifier and, 53–54
structured memory access and, 60
symbolic reference verification and, 56–58
system properties, 66
threads, 66, 109
trust models for, 43
type-safe reference casting and, 60
user Ids, 109
virtual machine and, 44, 59–62
Write operations, 66
security manager, 14, 44, 62–68
access control through, 62–68
check methods for, 65–68
code sources and, 64–65
customization of, 64
Java API and, 63–64, 67–58
mobility concepts and, 132
permissions and, 64–65, 67–58
policy files and, 64
protection domains and, 65, 84–86, **86**
semantic checks, class file verifier and, 54
serialization of objects, 3, 125, 131
service IDs, Jini technology and, 128
service items in Jini technology and, 127–128
service objects
Jini technology and, 126, 128, 129–131
mobility concepts and, 129–131
service objects, 125–131, 125
service registrar for Jini technology, 127
service templates for Jini technology, 128
shared applications/files, mobility concepts and, 114
sign of floating-point arithmetic, 424–427
signal-and-continue monitor in synchronization, 500
signature of method, 485
simple names in, 147, 201, 202
sipush opcode, 389
Smart Cards, Java Card Platform, 27
socket security, 65, 66
softly reachable states and garbage collection, 371–373, **372**, 375, 376–378
software development, mobility concepts and, 119
SourceFile attribute, 233
speed of execution, 17
_quick instructions, 305–306
stack (*see* Java stack; operand stack; stack frame)
stack frame (*see also* Java stack; operand stack), 138, 163–171
constant pool in, 168, 195–201, **196**, 206–214, 270
exception tables in, 168
frame data in, 167–168
local variables and, 163–166, **164**
memory allocation in, 163
operand stack and, 166–167, **167**, 171, 177, 179
stack inspection, doPrivileged( ) method, 100–109
stack inspection by Access Controller, examples of "yes" and "no," 90–99
Standard API, platform independence and, 30–31
Standard Extension APIs, platform independence and, 30–31
standardization versus, platform independence and, 36–39
static (early) binding in, 480
stop-and-copy garbage collection, 360–361, **361**
storage types in, 180

strings, 210, 211, 243
linking and, 290–292
operand stack operations on, 389
resolution and, 290–292
strongly reachable states and garbage collection, 374
structural class-file checks, class file verifier and, 53–54
subclasses, 243
super keyword and invokespecial instruction in, 487–489
superclasses, 151 198–199, 242, 243
superinterfaces, 199–200
swap opcode, 390
Swing, 12–13
layout managers for, 13
platform independence of, 12–13, 20
symbolic references, 20, 153, 201, 202–204, 209–210, 211, 212, 242–243
applets and, 125
class files, 196
loading constraints. 292–294, 343–353
resolution and, 279–280
Salutation application example of, 306–318, **311, 313, 314, 316, 317**
verification of, class file verifier and, 56–58
synchronization, 497–512
acquiring the monitor for, 498
action of thread through monitor in, 501–503, **501**
class object, coordination support and, 511
cooperation in, 498, 499
coordination in, 511
entering the monitor for, 498
entry sets in, 498, 503
first-in first-out (FIFO) entry set for, 503
instruction set support for, 505–511
last-in first-out (LIFO) entry set for, 503
locking/unlocking in, 497, 503–505
monitor for, 497, 498–503, **501**
mutual exclusion in, 498, 499
notify command for, 499–500, 502–503, 512
owning the monitor for, 498
regions for monitor in, 498
releasing the monitor for, 498
signal-and-continue monitor in, 500
synchronized methods and, 504, 508–511
synchronized statements and, 504, 505–508
timeouts for threads in, 502
wait command for, 499–500, 512
wait-and-notify monitor for, 499–500
synchronized methods, 504, 508–511
synchronized statements, 504, 505–508
Synthetic attribute, 234
system class loaders, 48–50, 144
system property security, 66

T

tableswitch opcode, 454
testing, platform independence and, 34–35, 36
threading, 19, 183–186
atomic types in, 184–185
bytecodes and, 55
current method in, 162
daemon threads in, 135–136
Java stack and, 137–138, 168–171, **169, 171**
locking/unlocking in, 158–159, 503–505
main memory in, 184
method area and, 147
native method stacks, 172–173, **173**

non-daemon threads in, 135–136
platform independence and, 33–34, 35
program counter in, 161–162
security and, 66, 109
synchronization and, 184–186, 497–512
timeouts, 502
time-slicing in, 184
variables in, 184
virtual machine and, 135–136, 138–139, **139**
wait sets, 159–160
working memories in, 184
Three-Dimensional Array applet, 437, 443–447, **444**
throwing exceptions, 460–464
timeouts in threads, 502
time-slicing, 184
tracing garbage collection, 358, 359
train algorithm for garbage collection, 363–368, **365**
trust models, 43
two's complement numbers, 407, 408–409
type conversion, 399–405,
types (*see* Java types),
type-safe reference casting and, 60

U

unconditional branching, 453
uniform resource locator (URL), 124
unloading Java types, 265–268, 336–343
unreachable state and garbage collection, 370, 375
user Id security, 109
user-defined class loaders, 8–11, **9**, 47, 48–50, 142–144, 241, 265–268, 323–333
mobility concepts and, 132
version 1.1, 323–329
version 1.2, 329–333

V

value sets for floating-point arithmetic, 429–431
variables (*see also* local variables)
constant variables in, 149
LocalVariableTable attribute, 231–233
popping to local variables in operand stack, 390–392
pushing local variables onto operand stack, 389–390
wide instruction for local variables, 392–394
vendors versus platform independence and, 36–39
verification of Java types, 238–240, **238**, 241–244
version and edition differences, platform independence and, 30–31
virtual machine, 1, 4, 5–7, **8**, 133–190
abstract specification to define, 134
adaptive optimization in, 6–7, 181–183
ahead-of-time compilers, 18
application programming interface (API) for, 12–14, **13**, 19–20
architecture of, 136–187, **137**
arrays and, 141, 161, **162**, 437–447
big-endian byte order in class files, 146
class class reference in, 150–151
class data type, 141
class files and, 6, 11–12
class loader of, 6, 6, 8–11, **9**, 136, **137**, 142–146, 149–150

compilers in, 6, 18
constant pool in, 148, 152–153, 168, 195–201, **196**, 205–214, 270
constant variables in, 149
control flow in, 449–458
daemon threads in, 135–136
data types for, 139–141, **140**, 147
defined, 134
dependencies of, platform independence versus, 33–34
distributed applications, 17–19
dynamic linking in, 150
error handling and, 62
exception tables in, 168
exceptions, 459–468
execution engine in, 6, 136, **137**, 173–186
field information for types in, 148
flexibility of, 5–6
floating-point arithmetic and, 140, 423–436
frame data in, 167–168
frames and, 138, 142
fully quantified names in, 147
garbage collection, 19, 33, 154–155, 160–161, 355–384
heap and, 137, 153–154, 154–161
holding data in, 138
implementation of, 134
initialization of classes in, 154
instruction set in, 174–181
integer arithmetic, 140, 407–416
interface data type, 141
Java interpreter and, 7
Java methods, 7
Java stack and, 137–138, 142, 162–163, 168–171, **169**
Jini technology and, 7, 186
just-in-time compiler of, 6
lifetime of, 134–136
little-endian byte order in class files, 146
local variables and, 142, 163–166, **164**
logic in, 417–422
main( ) method to invoke, 134–135, 152
memory allocation in (*see* heap)
method area in 137, 146–154
method information for data in, 148–149
method tables in, 151, 157–158
methods and, 7
mobility of, 11–12
name spaces in, 150
native methods and, 7, 12, 61–62, 138, 172–173, **173**, 186–187
non-daemon threads in, 135–136
numeric data types, 140
object representation in, 155–161, 437–447
opcodes in, 174–181
operand stack and, 142, 166–167, **167**, 171, 177, 179
PC register (*see* program counter)
platform independence and, 7, 11–12, 19–20, 23–40,
primitive values/primitive data types, 139–141
program counter, 137–138, 161–162
protection domains and, 65, 84–86, **86**
_quick instructions, 305–306
reference values/ reference data types, 139–141, 155–161
resolution in, constant pool, 153
resources Web page for, 190
roots and root sets, 357–358
runtime data areas, 136, **137**, 137–138, **138, 139**
run-time instance of, 134–136
security and, 44, 59–62

simple names in, 147
specification to define, 134
speed of execution, 17
stack frame in, 138, 163–171
symbolic references in, 153
synchronization and, 184–186
threading in, 33–34, 135–136, 138–139, **139**, 183–186
virtual table (VTBL) versus method table in, 158
word size, 142
virtual math, eteral math simulation, 188–189, **190**
virtual table (VTBL) in, 158

W

wait command for synchronization, 499–500, 512
wait sets, 159–160
wait-and-notify monitor for synchronization, 499–500
weakly reachable states and garbage collection, 371–373, **372**, 375, 376–378
wide instruction, 392–394, 410, 470
word size, virtual machine and, 142
working memory, threading, 184
World Wide Web development and Java, 123
Write operation security, 66

X

XOR (*see* logic)

Y

Yellin, Frank, 134

OBJECTS AND JAVA SEMINAR

A Five-Day Intensive, Hands-On, Seminar Taught by Bill Venners

This course covers the full extent of the Java language and several of the core APIs, including input/output, AWT, Java 2D, Swing, network programming, and RMI. A special emphasis is placed on understanding Java's object-oriented nature and on effective use of the language constructs, such as interfaces, exceptions, and threads.

- Taught in-house by Bill Venners
- Includes in-class programming exercises
- Also available in condensed, lecture-only format
- Can be customized to suit your needs

For more details about this seminar and others offered by Bill Venners, visit www.artima.com.

In this superb piece of work, Bill Venners explains in detail the inner workings of the Java Virtual Machine (JVM) by presenting possible implementations of many parts of that intricate piece of software. It is therefore a welcome complement to Sun's official specification. Each concept is clearly presented, often with the help of sample code. The accompanying CD also contains enlightening demos about the inner workings of the virtual machine. This book will be greatly appreciated not only by VM implementers, but also by anyone just curious to understand a component that's at the very heart of Java.

Antoine Trux, a Project Manager at Nokia Research Center in Helsinki, Finland, *JAVA Report*, December, 1998. *Inside the JAVA Virtual Machine* was the winner of one of *JAVA Report's* 1998 Writer's Choice Awards.

Before I delve into the structure and content of this book, I would like to mention the aspect of Venners's book that impressed me most of all: the sheer attention to detail and consistent accuracy of his writing.

The recurring (and expensive-to-produce) features in these chapters [5 through 20] are the animated, interactive, and enlightening applets that bring to life those chapters' main topics. The garbage collection chapter, for example, not only contains a good introduction to various modern garbage collection algorithms but also includes a `Heap of Fish` applet to let the reader accumulate a real, hands-on understanding of garbage collection issues and possible solutions.

The simple fact is that Venners's book is excellent, and his is the book I have to recommend.

Laurence Vanhelsuwé, *JavaWorld Magazine*, March, 1998.

Thank you for your excellent book. I've been writing programs in Java for a couple of years now, and it has really helped me get insight into the guts of the language. Thanks again for a wonderful read!

Noah S. Friedland, PhD

Recently bought your book, which is worlds easier then reading the JVM Specification! I also love your applets. They make things a lot easier to understand.

Paul Bathen

Your book, *Inside the Java Virtual Machine*, is one of the best-written and most helpful books in my Java collection.

Louis Barton

I just finished your book, *Inside the Java Virtual Machine*, and would like to thank you for a very useful piece of work!

Antoine Trux

A detailed and methodical book on Java Virtual Machine. This book is a must if you are planning on writing a JVM on your own or you have ever been wondering 'What the heck it takes to execute a .class file.' This book is a welcome relief to all those who may have just read the specification on Java Virtual Machine and are looking for something more explanatory.

Gopal Ananthraman

I'm really enjoying reading your book. It has lots of good stuff that I feel will make me a better Java programmer.

Joel Nylund, Principal, American Management Systems

I purchased a copy of *Inside the Java Virtual Machine*. Although I've only read Chapters 7 and 8, I'm extremely pleased and impressed by the detail therein. You answered many questions that surfaced, including, "which class loader does the VM construe to have loaded classes for which a dynamic class loader delegated responsibility by calling `ClassLoader.findSystemClass()`?"

While I was at Lotus Development Corporation I coauthored a text for Prentice-Hall titled *Inside the Lotus Add-In Toolkit*. The technology we discussed was similar to Java—a platform neutral, partially compiled language whose byte codes required a runtime virtual machine on which to execute.

Our goal as authors was to convey technical material with accuracy and good humor. We really sweated the terminology and paid special attention to consistency and technical detail—as developers we wanted this text to be useful and correct. As readers, we polished our English usage because we dreaded reading most popular technical texts.

All this is by way of reinforcing my feedback on your work. When an author takes the time to write complete sentences, to develop a conversational tone, to be consistent in terminology, and to provide real value added rather than simply reiterating (often imprecisely) the published specifications, I sit up and take notice.

David McCall

The best Java book if you really want to go under the hood. *Inside JVM* is an awsome book if you really want to know the ins and outs of JVM. I am amazed with the ability of Mr. Bill as a technical writer. I will strongly recomend this book for any serious Java developer who needs to know Java beyond the buzz words.

Rashid Jilani, on AMAZON.COM

A great book.

This is the best Java book I have read so far. Bill is a great software engineer and writer. If you want to know about the inside of JVM, this is a must have.

Michael Young, on AMAZON.COM

ABOUT THE AUTHOR

Bill Venners provides software consulting services to Silicon Valley and the world under the name Artima Software Company (`http://www.artima.com`). You can reach him at `bv@artima.com`.

EXHIBIT A

JAVA DEVELOPMENT KIT VERSION 1.1.4 BINARY CODE LICENSE

This binary code license ("License") contains rights and restrictions associated with use of the accompanying software and documentation ("Software"). Read the License carefully before installing the Software. By installing the Software you agree to the terms and conditions of this License.

1. Limited License Grant. Sun grants to you ("Licensee") a non-exclusive, non-transferable limited license to use the Software without fee for evaluation of the Software and for development of Java symbol 228 \f "Symbol" \s 12 ‰ compatible applets and applications. Licensee may make one archival copy of the Software. Licensee may not re-distribute the Software in whole or in part, either separately or included with a product. Refer to the Java Runtime Environment Version 1.1.4 binary code license (http://www.javasoft.com/ products/JDK/1.1.4/index.html) for the availability of runtime code which may be distributed with Java compatible applets and applications.

2. Java Platform Interface. Licensee may not modify the Java Platform Interface ("JPI", identified as classes contained within the "java" package or any subpackages of the "java" package), by creating additional classes within the JPI or otherwise causing the addition to or modification of the classes in the JPI. In the event that Licensee creates any Java-related API and distributes such API to others for applet or application development, Licensee must promptly publish an accurate specification for such API for free use by all developers of Java-based software.

3. Restrictions. Software is confidential copyrighted information of Sun and title to all copies is retained by Sun and/or its licensors. Licensee shall not modify, decompile, disassemble, decrypt, extract, or otherwise reverse engineer Software. Software may not be leased, assigned, or sublicensed, in whole or in part. **Software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee warrants that it will not use or redistribute the Software for such purposes.**

4. Trademarks and Logos. This License does not authorize Licensee to use any Sun name, trademark or logo. Licensee acknowledges that Sun owns the Java trademark and all Java-related trademarks, logos and icons including the Coffee Cup and Duke ("Java Marks") and agrees to: (i) to comply with the Java Trademark Guidelines at http://java.com/trademarks.html; (ii) not do anything harmful to or inconsistent with Sun's rights in the Java Marks; and (iii) assist Sun in protecting those rights, including assigning to Sun any rights acquired by Licensee in any Java Mark.

5. Disclaimer of Warranty. Software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED.

6. Limitation of Liability. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE OR ANY THIRD PARTY AS A RESULT OF USING OR DISTRIBUTING SOFTWARE. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination. Licensee may terminate this License at any time by destroying all copies of Software. This License will terminate immediately without notice from Sun if Licensee fails to comply with any provision of this License. Upon such termination, Licensee must destroy all copies of Software.

8. Export Regulation. Software, including technical data, is subject to U.S. export control laws, including the U.S. Export Administration Act and its associated regulations, and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such regulations and acknowledges that it has the responsibility to obtain licenses to export, re-export, or import Software. Software may not be downloaded, or otherwise exported or re-exported (i) into, or to a national or resident of, Cuba, Iraq, Iran, North Korea, Libya, Sudan, Syria or any country to which the U.S. has embargoed goods; or (ii) to anyone on the U.S. Treasury Department's list of Specially Designated Nations or the U.S. Commerce Department's Table of Denial Orders.

9. Restricted Rights. Use, duplication or disclosure by the United States government is subject to the restrictions as set forth in the Rights in Technical Data and Computer Software Clauses in DFARS 252.227-7013(c)(1)(ii) and FAR 52.227-19(c)(2) as applicable.

10. Governing Law. Any action related to this License will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

11. Severability. If any of the above provisions are held to be in violation of applicable law, void, or unenforceable in any jurisdiction, then such provisions are herewith waived to the extent necessary for the License to be otherwise enforceable in such jurisdiction. However, if in Sun's opinion deletion of any provisions of the License by operation of this paragraph unreasonably compromises the rights or increase the liabilities of Sun or its licensors, Sun reserves the right to terminate the License and refund the fee paid by Licensee, if any, as Licensee's sole and exclusive remedy.

SOFTWARE AND INFORMATION LICENSE

The software and information on this diskette (collectively referred to as the "Product") are the property of The McGraw-Hill Companies, Inc. ("McGraw-Hill") and are protected by both United States copyright law and international copyright treaty provision. You must treat this Product just like a book, except that you may copy it into a computer to be used and you may make archival copies of the Products for the sole purpose of backing up our software and protecting your investment from loss.

By saying "just like a book," McGraw-Hill means, for example, that the Product may be used by any number of people and may be freely moved from one computer location to another, so long as there is no possibility of the Product (or any part of the Product) being used at one location or on one computer while it is being used at another. Just a book cannot be read by two different people in two different places at the same time, neither can the Product be used by two different people in two different places at the same time (unless, of course, McGraw-Hill's rights are being violated).

McGraw-Hill reserves the right to alter or modify the contents of the Product at any time.

This agreement is effective until terminated. The Agreement will terminate automatically without notice if you fail to comply with any provisions of this Agreement. In the event of termination by reason of your breach, you will destroy or erase all copies of the Product installed on any computer system or made for backup purposes and shall expunge the Product from your data storage facilities.

LIMITED WARRANTY

McGraw-Hill warrants the physical diskette(s) enclosed herein to be free of defects in materials and workmanship for a period of sixty days from the purchase date. If McGraw-Hill receives written notification within the warranty period of defects in materials or workmanship, and such notification is determined by McGraw-Hill to be correct, McGraw-Hill will replace the defective diskette(s). Send request to:

Customer Service
McGraw-Hill
Gahanna Industrial Park
860 Taylor Station Road
Blacklick, OH 43004-9615

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective diskette(s) and shall not include or extend any claim for or right to cover any other damages, including but not limited to, loss of profit, data, or use of the software, or special, incidental, or consequential damages or other similar claims, even if McGraw-Hill has been specifically advised as to the possibility of such damages. In no event will McGraw-Hill's liability for any damages to you or any other person ever exceed the lower of suggested list price or actual price paid for the license to use the Product, regardless of any form of the claim.

THE McGRAW-HILL COMPANIES, INC. SPECIFICALLY DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Specifically, McGraw-Hill makes no representation or warranty that the Product is fit for any particular purpose and any implied warranty of merchantability is limited to the sixty day duration of the Limited Warranty covering the physical diskette(s) only (and not the software or information) and is otherwise expressly and specifically disclaimed.

This Limited Warranty gives you specific legal rights; you may have others which may vary from state to state. Some states do not allow the exclusion of incidental or consequential damages, or the limitation on how long an implied warranty lasts, so some of the above may not apply to you.

This Agreement constitutes the entire agreement between the parties relating to use of the Product. The terms of any purchase order shall have no effect on the terms of this Agreement. Failure of McGraw-Hill to insist at any time on strict compliance with this Agreement shall not constitute a waiver of any rights under this Agreement. This Agreement shall be construed and governed in accordance with the laws of New York. If any provision of this Agreement is held to be contrary to law, that provision will be enforced to the maximum extent permissible and the remaining provisions will remain in force and effect.

JAVA
Virtual Machine
Second Edition
135095-0
Bill Venners

Java/Programming

$49.99 U.S.A.

Inside the JAVA 2 Virtual Machine

***Inside the Java 2 Virtual Machine* includes coverage of:**

The complete Java landscape and JVM structure

Class files, bytecodes, conversions, and verification during class loading

Arithmetic, logical, and array operations and control flow

Method invocation and return, exceptions, garbage collection and threads

The just-in-time compiler implementation of JVM

Want to write better Java programs? Look under the hood and see what makes the ingenious Java engine run!

This in-depth guide to Java's architecture and internals is your key to writing more effective and efficient Java code. By understanding the Java Virtual Machine (JVM), you will gain insights into the inner workings of Java technology that will help you harness the full range of Java's capabilities in your programs.

In addition, this comprehensive volume explains the architecture of the JVM, including the interaction of the Java stack, the heap, the method area, and the execution engine. It includes in-depth discussion of various implementation techniques, such as interpreting, just-in-time compiling, and adaptive optimization. It also describes the behavior of Java threads and the Java monitor.

It also covers garbage collection, including reference objects, the train algorithm, and object finalization. Finally, the intricacies of the Java security model are discussed, including type safety, the class loader architecture, class verifier, security manager, access controller, and code signing.

With this guide, you'll understand Java's linking model and dynamic extension, and you'll learn how to write class loaders. You'll also uncover the seven steps to writing a platform independent Java program.

ABOUT THE AUTHOR

Bill Venners has been writing software professionally for 14 years. Based in Silicon Valley, he provides software consulting and training services through Artima Software Company. Since 1996, he has written over 40 articles about Java technology. His popular columns in *JavaWorld* magazine have covered Java internals, object-oriented design techniques, and Jini technology. He is also the author and webmaster of artima.com, an online resource for Java and Jini developers. He teaches in-house and public Java training seminars and frequently speaks on Java technology at software conferences around the world.

ON THE VALUE-PACKED CD-ROM:

- Example programs
- Illustrative applets

X001BD73C9
Inside the Java 2 Virtual Machine
Used, Good

9 780071 350938

Cover design: Rich Williams
Illustration: Tom White
www.computing.mcgraw-hill.com
McGraw-Hill
A Division of The McGraw-Hill Companies