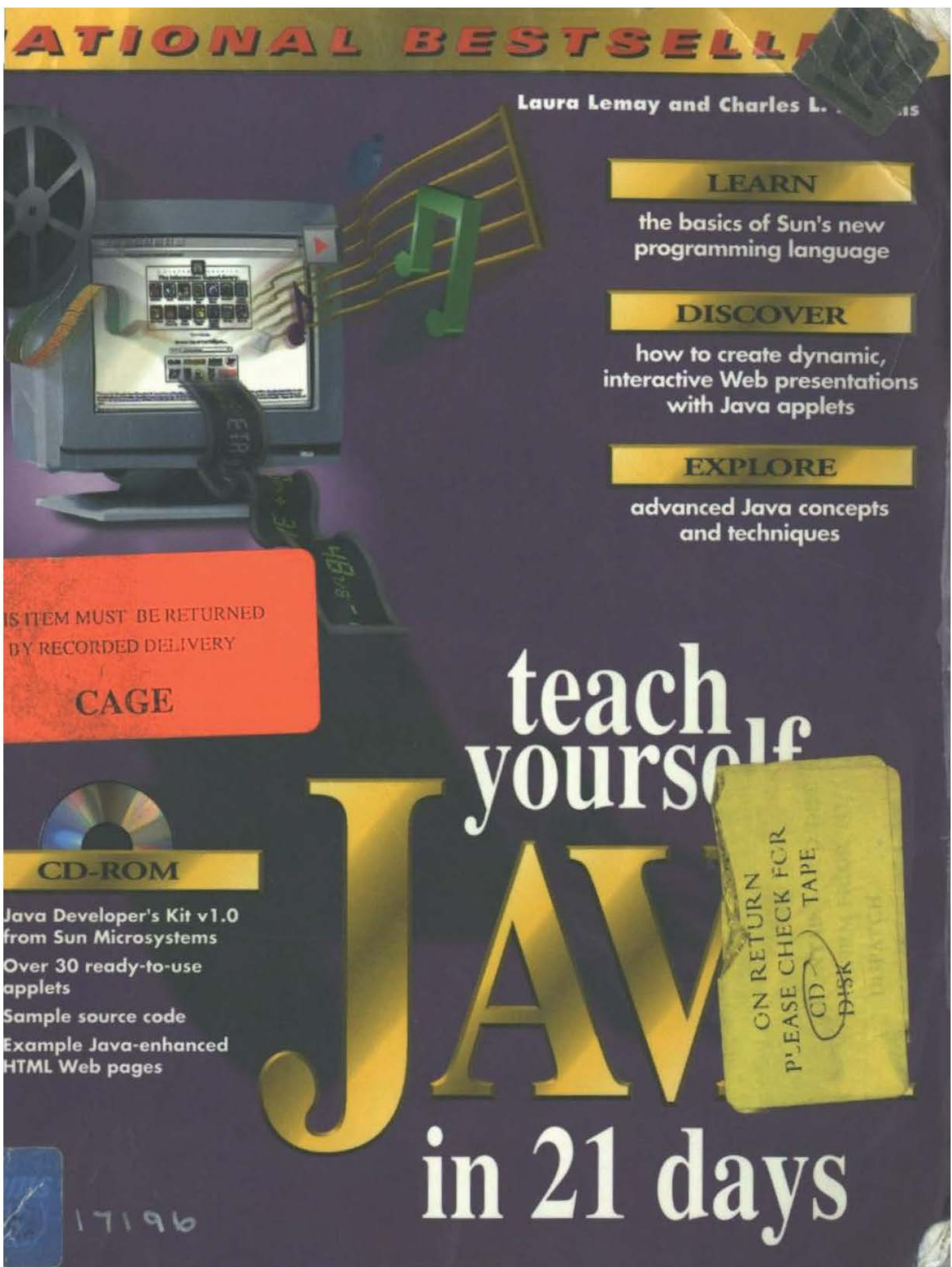




Your 21-Day Java Calendar

Week 1 The Java Language						
Chapter 1	Chapter 2	Chapter 3	Chapter 4	Chapter 5	Chapter 6	Chapter 7
An Introduction to Java Programming - Platform independence - The Java compiler and the Java interpreter	Object-Oriented Programming and Java - Objects and classes - Encapsulation - Modularity	Java Basics - Java statements and expressions - Variables and data types - Comparisons and logical operators	Working with Objects - Testing and debugging - Class variables - Class objects	Arrays, Conditionals, and Loops - Conditional tests - Iteration - Block statements	Creating Classes and Applications in Java Defining constants, instance and class variables, and methods	More About Methods - Overloading methods - Constructor methods - Overriding methods
Week 2 Developing Applets						
Chapter 8	Chapter 9	Chapter 10	Chapter 11	Chapter 12	Chapter 13	Chapter 14
Java Applet Basics - Including an applet on a Web page - Passing parameters	Graphics, Fonts, and Color - Graphics primitives - The Color class	Sound and Animation - Sound - Animation - Repositioning and scaling - Storing and retrieving data	More Images, Sound, and Animation - Scaling and clipping - Executing and scheduling - Interfering	Managing Simple Events and Interactivity - mouseDown and mouseUp - The Java event model	The Java Abstract Windowing Toolkit Canvases, text components, widgets, and window construction components	Windows, Networking, and Other Tidbits Programming menus and creating links inside applets
Week 3 Advanced Topics						
Chapter 15	Chapter 16	Chapter 17	Chapter 18	Chapter 19	Chapter 20	Chapter 21
Modifiers - Method and variable access control - Four P's of protection	Packages and Interfaces - Hiding classes - Design versus implementation - Inheritance	Exceptions - try, catch, throws - Using the finally clause	Multithreading - Thinking Multithreaded - Creating and using threads	Streams - RandomAccessFile and other related classes - Input and output streams	Native Methods and Libraries - Built-in optimizations - Generating header and stub files	Under the Hood - The big picture - Java bytecodes and verification - Security and garbage collection

About This Book

This book teaches you all about the Java language and how to use it to create applets and applications. By the time you get through with this book, you'll know enough about Java to do just about anything, inside an applet or out.

Who Should Read This Book

This book is intended for people with at least some basic programming background, which includes people with years of programming experience or people with only a small amount of experience. If you understand what variables, loops, and functions are, you'll be just fine for this book. The sorts of people who might want to read this book include you, if

- ☐ You're a real whiz at HTML, understand CGI programming (in Perl, AppleScript, Visual Basic, or some other popular CGI language) pretty well, and want to move on to the next level in Web page design.
- ☐ You had some Basic or Pascal in school and you have a basic grasp of what programming is, but you've heard Java is easy to learn, really powerful, and very cool.
- ☐ You've programmed C and C++ for many years, you've heard this Java thing is becoming really popular and you're wondering what all the fuss is all about.
- ☐ You've heard that Java is really good for Web-based applets, and you're curious about how good it is for creating more general applications.

What if you know programming, but you don't know object-oriented programming? Fear not. This book assumes no background in object-oriented design. If you know object-oriented programming, in fact, the first couple of days will be easy for you.

How This Book Is Structured

This book is intended to be read and absorbed over the course of three weeks. During each week, you'll read seven chapters that present concepts related to the Java language and the creation of applets and applications.

ON LOAN FROM BLDSC BOSTON SPA.
FOR USE AT THE BRITISH LIBRARY
ST. PANCRAS. READING ROOM ONLY.

RETURN DATE

18 OCT 2001

AEC-1

use.

ette. Please
the disk.
handling the

and its
blisher. The
nsibility for
ment by its

Conventions



Note: A Note box presents interesting pieces of information related to the surrounding discussion.



Technical Note: A Technical Note presents specific technical information related to the surrounding discussion.



Tip: A Tip box offers advice or teaches an easier way to do something.



Caution: A Caution box alerts you to a possible problem and gives you advice to avoid it.



Warning: A Warning box advises you about potential problems and helps you steer clear of disaster.

NEW TERM

New terms are introduced in New Term boxes, with the term in italics.



A type icon identifies some new code that you can type in yourself.



An Output icon highlights what the same code looks like when run.



An analysis icon alerts you to the author's line-by-line analysis.

THIS ITEM MUST BE RETURNED
BY RECORDED DELIVERY

CAGE

Teach Yourself JAVA

in 21 Days



56986

96 000238

THIS ITEM MUST BE RETURNED
BY RECORDED DELIVERY

CAGE

Teach Yourself JAVA in 21 Days

Laura Lemay
Charles L. Perkins



201 West 103rd Street
Indianapolis, Indiana 46290

BRITISH LIBRARY
DOCUMENT SUPPLY CENTRE

17 JUN 1996

V96/17196

Supplied by the British Library 03 Jun 2021, 08:19 (BST)

Petitioner's Exhibit 1005
Page 0006

IPR No. 2022-00572
Petitioner's Exhibit 1005
Page 0006

*To Eric, for all the usual reasons
(moral support, stupid questions, comfort in dark times).
LL*

*For RKJP, ARL, and NMH
the three most important people in my life.
CLP*

Copyright ©1996 by Sams.net Publishing

FIRST EDITION

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. For information, address Sams.net Publishing, 201 W. 103rd St., Indianapolis, IN 46290.

International Standard Book Number: 1-57521-030-4

Library of Congress Catalog Card Number: 95-78866

99 98 97 96 4 3

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit, the number of the book's printing. For example, a printing code of 96-1 shows that the first printing of the book occurred in 1996.

Composed in AGaramond and MCPdigital by Macmillan Computer Publishing

Printed in the United States of America

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams.net Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

President, Sams Publishing:	<i>Richard K. Swadley</i>
Publisher, Sams.net Publishing:	<i>George Bond</i>
Publishing Manager:	<i>Mark Taber</i>
Managing Editor:	<i>Cindy Morrow</i>
Marketing Manager:	<i>John Pierce</i>

Acquisitions Editor

Mark Taber

Development Editor

Fran Hatton

Software Development Specialist

Merle Newlon

Production Editor

Nancy Albright

Technical Reviewer

Patrick Chan

Editorial Coordinator

Bill Whitmer

Technical Edit Coordinator

Lynette Quinn

Formatter

Frank Sinclair

Editorial Assistant

Carol Ackerman

Cover Designer

Tim Amrhein

Book Designer

Alyssa Yesh

Production Team Supervisor

Brad Chinn

Production

Michael Brumitt

Jason Hand

Cheryl Moore

Ayanna Lacey

Nancy Price

Bobbi Satterfield

Tim Taylor

Susan Van Ness

Mark Walchle

Todd Wente

Indexer

Tim Griffin

Overview

	Introduction	xxi
Week 1 at a Glance		
Day	1 An Introduction to Java Programming	3
	2 Object-Oriented Programming and Java	19
	3 Java Basics	41
	4 Working with Objects	61
	5 Arrays, Conditionals, and Loops	79
	6 Creating Classes and Applications in Java	95
	7 More About Methods	111
Week 2 at a Glance		
Day	8 Java Applet Basics	129
	9 Graphics, Fonts, and Color	149
	10 Simple Animation and Threads	173
	11 More Animation, Images, and Sound	195
	12 Managing Simple Events and Interactivity	217
	13 User Interfaces with the Java Abstract Windowing Toolkit	237
	14 Windows, Networking, and Other Tidbits	279
Week 3 at a Glance		
Day	15 Modifiers	305
	16 Packages and Interfaces	323
	17 Exceptions	341
	18 Multithreading	353
	19 Streams	375
	20 Native Methods and Libraries	403
	21 Under the Hood	421
Appendixes		
	A Language Summary	473
	B Class Hierarchy Diagrams	483
	C The Java Class Library	497
	D How Java Differs from C and C++	507
	Index	511



viii

Contents

	Introduction	xxi
	Week 1 at a Glance	1
Day	1 An Introduction to Java Programming	3
	What Is Java?	4
	Java's Past, Present, and Future	6
	Why Learn Java?	7
	Java Is Platform-Independent	7
	Java Is Object-Oriented	9
	Java Is Easy to Learn	9
	Getting Started with	
	Programming in Java	10
	Getting the Software	10
	Applets and Applications	11
	Creating a Java Application	11
	Creating a Java Applet	13
	Summary	16
	Q&A	16
Day	2 Object-Oriented Programming and Java	19
	Thinking in Objects: An Analogy	20
	Objects and Classes	21
	Behavior and Attributes	23
	Attributes	23
	Behavior	24
	Creating a Class	24
	Inheritance, Interfaces, and Packages	28
	Inheritance	29
	Creating a Class Hierarchy	30
	How Inheritance Works	32
	Single and Multiple Inheritance	34
	Interfaces and Packages	34
	Creating a Subclass	35
	Summary	38
	Q&A	39
Day	3 Java Basics	41
	Statements and Expressions	42
	Variables and Data Types	43
	Declaring Variables	43
	Notes on Variable Names	44



Teach Yourself JAVA in 21 Days

	Variable Types	45
	Assigning Values to Variables	46
	Comments	47
	Literals	47
	Number Literals	47
	Boolean Literals	48
	Character Literals	48
	String Literals	49
	Expressions and Operators	50
	Arithmetic	50
	More About Assignment	52
	Incrementing and Decrementing	52
	Comparisons	54
	Logical Operators	55
	Bitwise Operators	55
	Operator Precedence	56
	String Arithmetic	57
	Summary	58
	Q&A	60
Day	4 Working with Objects	61
	Creating New Objects	62
	Using <i>new</i>	63
	What <i>new</i> Does	64
	A Note on Memory Management	64
	Accessing and Setting Class and Instance Variables	65
	Getting Values	65
	Changing Values	65
	Class Variables	66
	Calling Methods	67
	Class Methods	69
	References to Objects	70
	Casting and Converting Objects and Primitive Types	71
	Casting Primitive Types	71
	Casting Objects	72
	Converting Primitive Types to Objects and Vice Versa	73
	Odds and Ends	73
	Comparing Objects	74
	Copying Objects	75
	Determining the Class of an Object	76
	The Java Class Library	76
	Summary	77
	Q&A	78

Day	5	Arrays, Conditionals, and Loops	79
		Arrays	80
		Declaring Array Variables	80
		Creating Array Objects	81
		Accessing Array Elements	81
		Changing Array Elements	82
		Multidimensional Arrays	83
		Block Statements	83
		<i>if</i> Conditionals	83
		The Conditional Operator	84
		<i>switch</i> Conditionals	85
		<i>for</i> Loops	86
		<i>while</i> and <i>do</i> Loops	88
		<i>while</i> Loops	88
		<i>do...while</i> Loops	89
		Breaking Out of Loops	89
		Labeled Loops	90
		Summary	91
		Q&A	92
Day	6	Creating Classes and Applications in Java	95
		Defining Classes	96
		Creating Instance and Class Variables	96
		Defining Instance Variables	97
		Constants	97
		Class Variables	98
		Creating Methods	99
		Defining Methods	99
		The <i>this</i> Keyword	101
		Variable Scope and Method Definitions	101
		Passing Arguments to Methods	102
		Class Methods	104
		Creating Java Applications	105
		Java Applications and Command-Line Arguments	106
		Passing Arguments to Java Programs	106
		Handling Arguments in Your Java Program	106
		Summary	108
		Q&A	109
Day	7	More About Methods	111
		Creating Methods with the Same Name, Different Arguments	112
		Constructor Methods	115
		Basic Constructors	116
		Calling Another Constructor	117
		Overloading Constructors	117

		Overriding Methods	119
		Creating Methods that Override Existing Methods	119
		Calling the Original Method	121
		Overriding Constructors	122
		Finalizer Methods	123
		Summary	124
		Q&A	124
	Week 2 at a Glance		127
Day	8	Java Applet Basics	129
		How Applets and Applications Are Different	130
		Creating Applets	131
		Major Applet Activities	132
		A Simple Applet	134
		Including an Applet on a Web Page	136
		The <code><APPLET></code> Tag	136
		Testing the Result	137
		Making Java Applets Available to the Web	137
		More About the <code><APPLET></code> Tag	138
		<code>ALIGN</code>	138
		<code>HSPACE</code> and <code>VSPACE</code>	140
		<code>CODE</code> and <code>CODEBASE</code>	141
		Passing Parameters to Applets	141
		Summary	146
		Q&A	147
Day	9	Graphics, Fonts, and Color	149
		The Graphics Class	150
		The Graphics Coordinate System	151
		Drawing and Filling	151
		Lines	152
		Rectangles	152
		Polygons	155
		Ovals	156
		Arc	157
		A Simple Graphics Example	161
		Copying and Clearing	163
		Text and Fonts	163
		Creating Font Objects	163
		Drawing Characters and Strings	164
		Finding Out Information About a Font	166
		Color	168
		Using Color Objects	168
		Testing and Setting the Current Colors	169
		A Single Color Example	170
		Summary	171
		Q&A	171



Day	10	Simple Animation and Threads	173
		Creating Animation in Java	174
		Painting and Repainting	174
		Starting and Stopping an Applet's Execution	175
		Putting It Together	175
		Threads: What They Are and Why You Need Them	177
		The Problem with the Digital Clock Applet	178
		Writing Applets with Threads	179
		Fixing The Digital Clock	180
		Reducing Animation Flicker	182
		Flicker and How to Avoid It	182
		How to Override Update	183
		Solution One: Don't Clear the Screen	183
		Solution Two: Redraw Only What You Have To	186
		Summary	192
		Q&A	192
Day	11	More Animation, Images, and Sound	195
		Retrieving and Using Images	196
		Getting Images	196
		Drawing Images	198
		Modifying Images	201
		Creating Animation Using Images	201
		An Example: Neko	201
		Retrieving and Using Sounds	209
		Sun's Animator Applet	211
		More About Flicker: Double-Buffering	212
		Creating Applets with Double-Buffering	212
		An Example: Checkers Revisited	213
		Summary	214
		Q&A	215
Day	12	Managing Simple Events and Interactivity	217
		Mouse Clicks	218
		<i>mouseDown</i> and <i>mouseUp</i>	219
		An Example: Spots	220
		Mouse Movements	223
		<i>mouseDrag</i> and <i>mouseMove</i>	223
		<i>mouseEnter</i> and <i>mouseExit</i>	223
		An Example: Drawing Lines	224
		Keyboard Events	228
		The <i>keyDown</i> Method	228
		Default Keys	229

		An Example: Entering, Displaying, and Moving Characters	229
		Testing for Modifier Keys	232
		The AWT Event Handler	233
		Summary	235
		Q&A	235
Day	13	The Java Abstract Windowing Toolkit	237
		An AWT Overview	238
		The Basic User Interface Components	240
		Labels	241
		Buttons	242
		Checkboxes	243
		Radio Buttons	244
		Choice Menus	245
		Text Fields	247
		Panels and Layout	249
		Layout Managers	249
		Insets	254
		Handling UI Actions and Events	255
		Nesting Panels and Components	258
		Nested Panels	258
		Events and Nested Panels	258
		More UI Components	259
		Text Areas	259
		Scrolling Lists	261
		Scrollbars and Sliders	262
		Canvases	265
		More UI Events	265
		A Complete Example:	
		RGB to HSB Converter	266
		Create the Applet Layout	267
		Create the Panel Layout	267
		Define the Subpanels	269
		Handle the Actions	272
		Update the Result	272
		The Complete Source Code	274
		Summary	277
		Q&A	277
Day	14	Windows, Networking, and Other Tidbits	279
		Windows, Menus, and Dialog Boxes	280
		Frames	280
		Menus	282
		Dialog Boxes	285
		File Dialogs	287
		Window Events	288
		Using AWT Windows in Stand-Alone Applications	288

		Networking in Java	289
		Creating Links Inside Applets	290
		Opening Web Connections	292
		<i>openStream()</i>	293
		The <i>URLConnection</i> Class	296
		Sockets	296
		Other Applet Hints	297
		The <i>showStatus</i> Method	297
		Applet Information	298
		Communicating Between Applets	298
		Summary	299
		Q&A	300
	Week 3 at a Glance		303
Day	15	Modifiers	305
		Method and Variable Access Control	307
		The Four P's of Protection	307
		The Conventions for Instance Variable Access	312
		Class Variables and Methods	314
		The <i>final</i> Modifier	316
		<i>final</i> Classes	316
		<i>final</i> Variables	317
		<i>final</i> Methods	317
		<i>abstract</i> Methods and Classes	319
		Summary	320
		Q&A	320
Day	16	Packages and Interfaces	323
		Packages	324
		Programming in the Large	324
		Programming in the Small	327
		Hiding Classes	329
		Interfaces	331
		Programming in the Large	331
		Programming in the Small	335
		Summary	338
		Q&A	339
Day	17	Exceptions	341
		Programming in the Large	342
		Programming in the Small	345
		The Limitations Placed on the Programmer	348
		The <i>finally</i> Clause	349
		Summary	350
		Q&A	351



Teach Yourself JAVA in 21 Days

Day	18	Multithreading	353
		The Problem with Parallelism	354
		Thinking Multithreaded	355
		Points About <i>Points</i>	357
		Protecting a Class Variable	360
		Creating and Using Threads	361
		The <i>Runnable</i> Interface	362
		<i>ThreadTester</i>	363
		<i>NamedThreadTester</i>	365
		Knowing When a Thread has Stopped	366
		Thread Scheduling	367
		Preemptive Versus Nonpreemptive	367
		Testing Your Scheduler	368
		Summary	371
		Q&A	372
Day	19	Streams	375
		Input Streams	377
		The <i>abstract</i> Class <i>InputStream</i>	377
		<i>ByteArrayInputStream</i>	381
		<i>FileInputStream</i>	382
		<i>FilterInputStream</i>	383
		<i>PipedInputStream</i>	389
		<i>SequenceInputStream</i>	389
		<i>StringBufferInputStream</i>	390
		Output Streams	391
		The <i>abstract</i> Class <i>OutputStream</i>	391
		<i>ByteArrayOutputStream</i>	392
		<i>FileOutputStream</i>	393
		<i>FilterOutputStream</i>	394
		<i>PipedOutputStream</i>	399
		Related Classes	399
		Summary	399
		Q&A	400
Day	20	Native Methods and Libraries	403
		Disadvantages of <i>native</i> Methods	404
		The Illusion of Required Efficiency	405
		Built-In Optimizations	407
		Simple Optimization Tricks	407
		Writing <i>native</i> Methods	408
		The Example Class	409
		Generating Header and Stub Files	410
		Creating <i>SimpleFileNative.c</i>	414



		A Native Library	417
		Linking It All	418
		Using Your Library	418
		Summary	418
		Q&A	419
Day	21	Under the Hood	421
		The Big Picture	422
		Why It's a Powerful Vision	423
		The Java Virtual Machine	423
		An Overview	424
		The Fundamental Parts	426
		The Constant Pool	430
		Limitations	430
		Bytecodes in More Detail	431
		The Bytecode Interpreter	431
		The "Just-in-Time" Compiler	432
		The <i>java2c</i> Translator	433
		The Bytecodes Themselves	434
		The <i>_quick</i> Bytecodes	450
		The .class File Format	452
		Method Signatures	454
		The Garbage Collector	455
		The Problem	455
		The Solution	456
		Java's Parallel Garbage Collector	459
		The Security Story	459
		Why You Should Worry	459
		Why You Might Not Have To	460
		Java's Security Model	460
		Summary	470
		Q&A	470
	A	Language Summary	473
		Reserved Words	474
		Comments	475
		Literals	475
		Variable Declaration	476
		Variable Assignment	476
		Operators	477
		Objects	478
		Arrays	478
		Loops and Conditionals	478
		Class Definitions	479
		Method and Constructor Definitions	479
		Packages, Interfaces, and Importing	480
		Exceptions and Guarding	481

B	Class Hierarchy Diagrams	483
	About These Diagrams	495
C	The Java Class Library	497
	<i>java.lang</i>	498
	Interfaces	498
	Classes	498
	<i>java.util</i>	499
	Interfaces	499
	Classes	499
	<i>java.io</i>	500
	Interfaces	500
	Classes	500
	<i>java.net</i>	501
	Interfaces	501
	Classes	502
	<i>java.awt</i>	502
	Interfaces	502
	Classes	502
	<i>java.awt.image</i>	504
	Interfaces	504
	Classes	504
	<i>java.awt.peer</i>	505
	<i>java.applet</i>	505
	Interfaces	505
	Classes	505
D	How Java Differs from C and C++	507
	Pointers	508
	Arrays	508
	Strings	508
	Memory Management	509
	Data Types	509
	Operators	509
	Control Flow	510
	Arguments	510
	Other Differences	510
	Index	511

Learn the basic concepts of object-oriented programming with Java

Use the book's numerous examples and tutorials to get hands-on practice creating Java applets

Add interactivity, animation and full multimedia to your Web presentations, such as 3-D models, "bouncing" graphics, educational tools, games, and stock quotes

Create stand-alone operating system-independent programs in Java

Includes detailed Java class hierarchy diagrams

User Level



New Casual Accomplished Expert

Category: Communications/
On-line—Internet

Covers: Java



7 52063 10304 3

Learn the hottest new language of the Web!

Teach Yourself Java™ in 21 Days carefully steps you through all you need to learn to transform your static Web pages into living, interactive applications with Java. You'll learn the fundamental concepts of the Java language as well as the basics of applet design and integration with Web presentations.

Java opens up an entire new world of possibilities previously unthinkable on the Web. You'll find out how to create applets for animations, computer simulations, interactive games and models, teaching tools, and more.

By the time you finish this book you'll have complete understanding of the Java language and how to design Java applets. You'll know how to create, use, and integrate Java applets into your Web design. *Teach Yourself Java in 21 Days* will provide you with a solid foundation for creating more sophisticated Java programs in the future.



CD-ROM includes Sun's official Java Developer's Kit v1.0 (compiler, debugger and interpreter), the Java language and API documentation, sample Java applets, example files from the book, and more.

Sun's JDK software is compatible with Windows 95, Windows NT, SPARC/Solaris 2.3-2.5

Laura Lemay is the author of the best-selling *Teach Yourself Web Publishing with HTML in a Week* and *Teach Yourself More Web Publishing with HTML in a Week*.

Charles L. Perkins was a researcher in the early days of Thinking Machines Corp. and the later days of Xerox PARC's Smalltalk group. He has programmed, written, and lectured extensively about most modern programming languages, and has been object-oriented for 15 years.

Visit the home page of *Teach Yourself Java in 21 Days* at:
<http://www.mcp.com/samsnet/books/tyjava/>



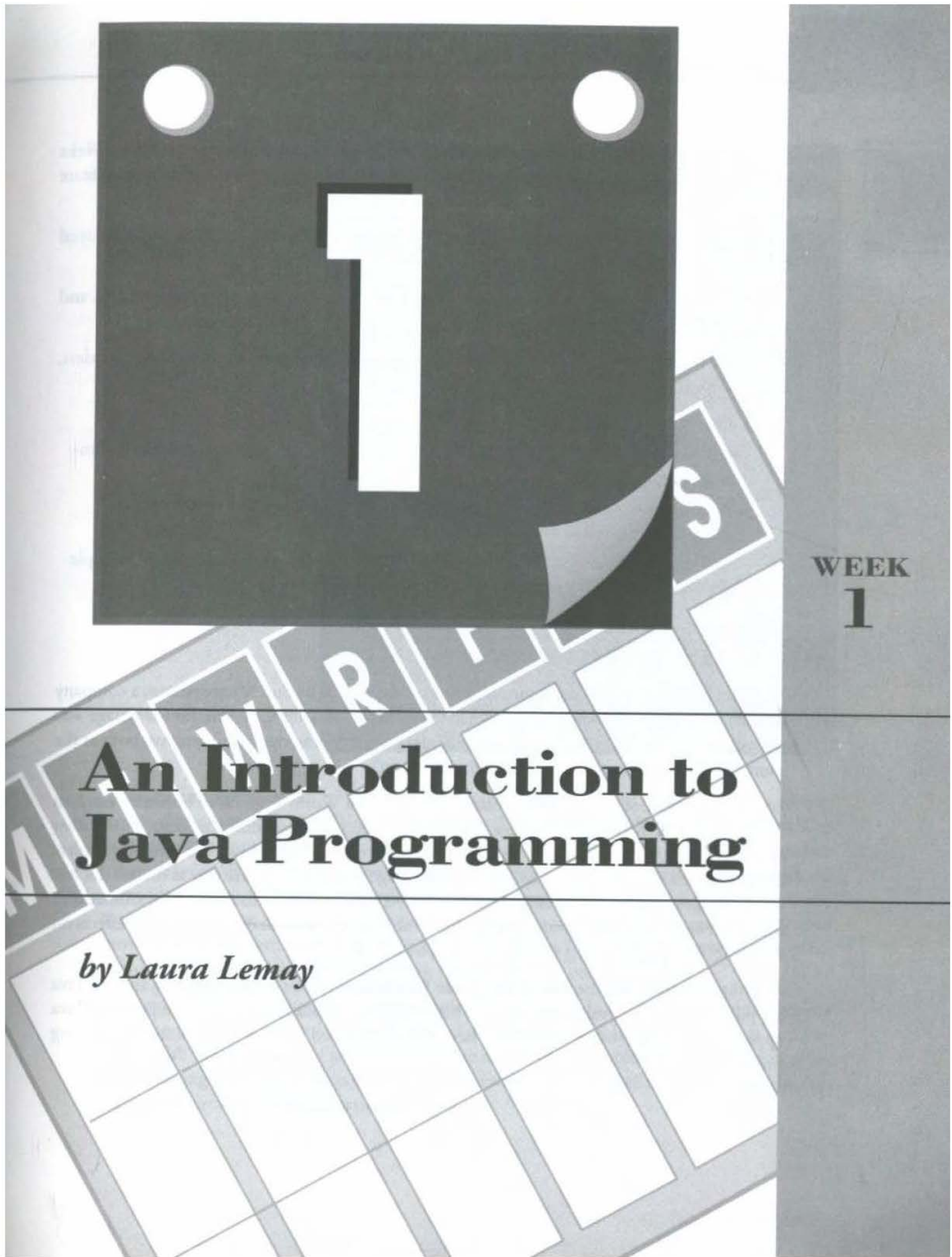
\$39.99 USA / \$53.99 CAN / £37.50 Net UK (inc of VAT)

ISBN 1-57521-030-4



9 781575 210308







An Introduction to Java Programming

Hello and welcome to *Teach Yourself Java in 21 Days*! Starting today and for the next three weeks you'll learn all about the Java language and how to use it to create applets, as well as how to create stand-alone Java applications that you can use for just about anything.

NEW TERM An *applet* is a dynamic and interactive program that can run inside a Web page displayed by a Java-capable browser such as HotJava or Netscape 2.0.

The *HotJava browser* is a World Wide Web browser used to view Web pages, follow links, and submit forms. It can also download and play applets on the reader's system.

That's the overall goal for the next three weeks. Today, the goals are somewhat more modest, and you'll learn about the following:

- ☐ What exactly Java and HotJava are, and their current status
- ☐ Why you should learn Java—its various features and advantages over other programming languages
- ☐ Getting started programming in Java—what you'll need in terms of software and background, as well as some basic terminology
- ☐ How to create your first Java programs—to close this day, you'll create both a simple Java application and a simple Java applet!

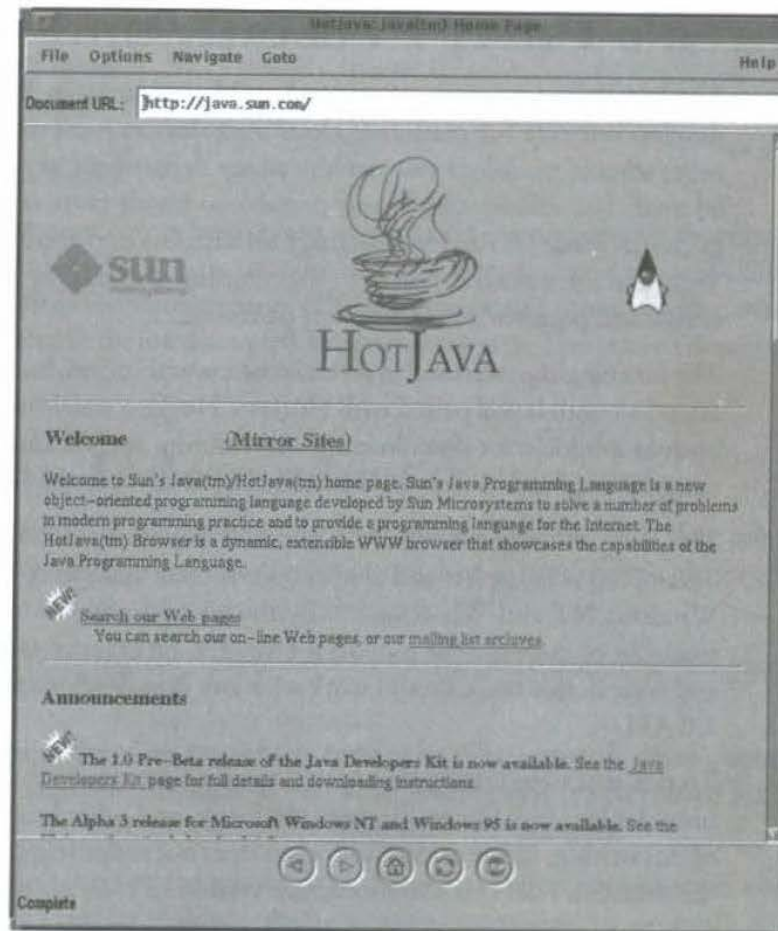
What Is Java?

Java is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end Unix workstations. Modeled after C++, the Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level (more about this later).

Java is often mentioned in the same breath as HotJava, a World Wide Web browser from Sun like Netscape or Mosaic (see Figure 1.1). What makes HotJava different from most other browsers is that, in addition to all its basic Web features, it can also download and play applets on the reader's system. Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive. Applets can be used to create animations, figures, or areas that can respond to input from the reader, games, or other interactive effects on the same Web pages among the text and graphics.

Although HotJava was the first World Wide Web browser to be able to play Java applets, Java support is rapidly becoming available in other browsers. Netscape 2.0 provides support for Java applets, and other browser developers have also announced support for Java in forthcoming products.

Figure 1.1.
The HotJava browser.



To create an applet, you write it in the Java language, compile it using a Java compiler, and refer to that applet in your HTML Web pages. You put the resulting HTML and Java files on a Web site much in the same way that you make ordinary HTML and image files available. Then, when someone using the HotJava browser (or other Java-enabled browser) views your page with the embedded applet, that browser downloads the applet to the local system and executes it, and then the reader can view and interact with your applet in all its glory (readers using other browsers may see text, a static graphic, or nothing). You'll learn more about how applets, browsers, and the World Wide Web work together further on in this book.

The important thing to understand about Java is that you can do so much more with it besides create applets. Java was written as a full-fledged programming language in which you can accomplish the same sorts of tasks and solve the same sorts of problems that you can in other programming languages, such as C or C++. HotJava itself, including all the networking, display, and user interface elements, is written in Java.

Java's Past, Present, and Future

The Java language was developed at Sun Microsystems in 1991 as part of a research project to develop software for consumer electronics devices—television sets, VCRs, toasters, and the other sorts of machines you can buy at any department store. Java's goals at that time were to be small, fast, efficient, and easily portable to a wide range of hardware devices. It is those same goals that made Java an ideal language for distributing executable programs via the World Wide Web, and also a general-purpose programming language for developing programs that are easily usable and portable across different platforms.

The Java language was used in several projects within Sun, but did not get very much commercial attention until it was paired with HotJava. HotJava was written in 1994 in a matter of months, both as a vehicle for downloading and running applets and also as an example of the sort of complex application that can be written in Java.

This book is current as of the 1.0 Java Developer's Kit (JDK). The 1.0 JDK includes tools for developing Java applets and applications on Sun systems running Solaris 2.3 or higher and for Windows NT and Windows 95. By the time you read this, new releases of the JDK may be available, or there may be support for the JDK on other systems and platforms. All the examples and code in this book should work with any Java development environment that supports the 1.0 API.

To run and view Java applets, you'll need a Java-capable browser or other tool. Netscape 2.0 provides Java capabilities for most platforms, and other browsers may also be Java-enabled. As of this writing, however, Sun's HotJava does not support applets written under the 1.0 JDK. By the time you read this a more current version of HotJava may exist.

Sun's 1.0 JDK does include an application called appletviewer that allows you to test your Java applets as you write them. If an applet works in the appletviewer, it should work with any Java-capable browser. You'll learn more about applet viewer later today.

What's in store for the future? In addition to the next Java release from Sun, other companies have announced support for Java in their own World Wide Web browsers. Netscape Communications Corporation has already incorporated Java capabilities into the 2.0 version of their very popular Netscape Navigator Web browser—pages with embedded Java applets can be viewed and played with Netscape. With support for Java available in as popular a browser as Netscape,



tools to help develop Java applications (debuggers, development environments, and so on) most likely will be rapidly available as well.



Why Learn Java?

At the moment, probably the most compelling reason to learn Java—and probably the reason you bought this book—is that HotJava applets are written in Java. Even if that were not the case, Java as a language has significant advantages over other languages and other programming environments that make it suitable for just about any programming task. This section describes some of those advantages.

Java Is Platform-Independent

Platform independence is one of the most significant advantages that Java has over other programming languages, particularly for systems that need to work on many different platforms. Java is platform-independent at both the source and the binary level.

NEW TERM *Platform-independence* is a program's capability of moving easily from one computer system to another.

At the source level, Java's primitive data types have consistent sizes across all development platforms. Java's foundation class libraries make it easy to write code that can be moved from platform to platform without the need to rewrite it to work with that platform.

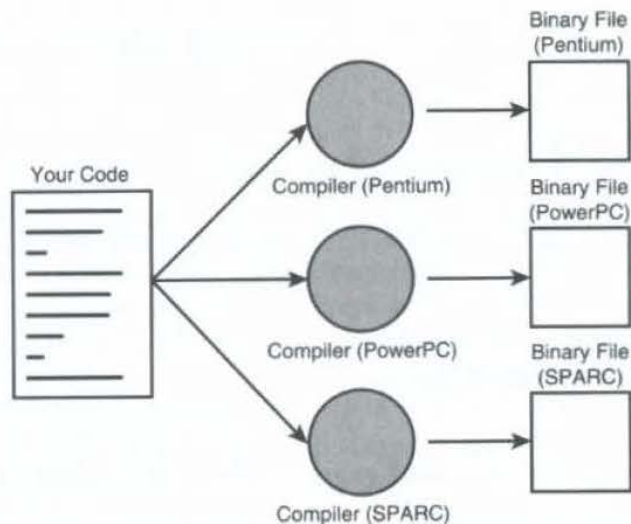
Platform-independence doesn't stop at the source level, however. Java binary files are also platform-independent and can run on multiple platforms without the need to recompile the source. How does this work? Java binary files are actually in a form called bytecodes.

NEW TERM *Bytecodes* are a set of instructions that look a lot like machine code, but are not specific to any one processor.

Normally, when you compile a program written in C or in most other languages, the compiler translates your program into machine codes or processor instructions. Those instructions are specific to the processor your computer is running—so, for example, if you compile your code on a Pentium system, the resulting program will run only on other Pentium systems. If you want to use the same program on another system, you have to go back to your original source, get a compiler for that system, and recompile your code. Figure 1.2 shows the result of this system: multiple executable programs for multiple systems.

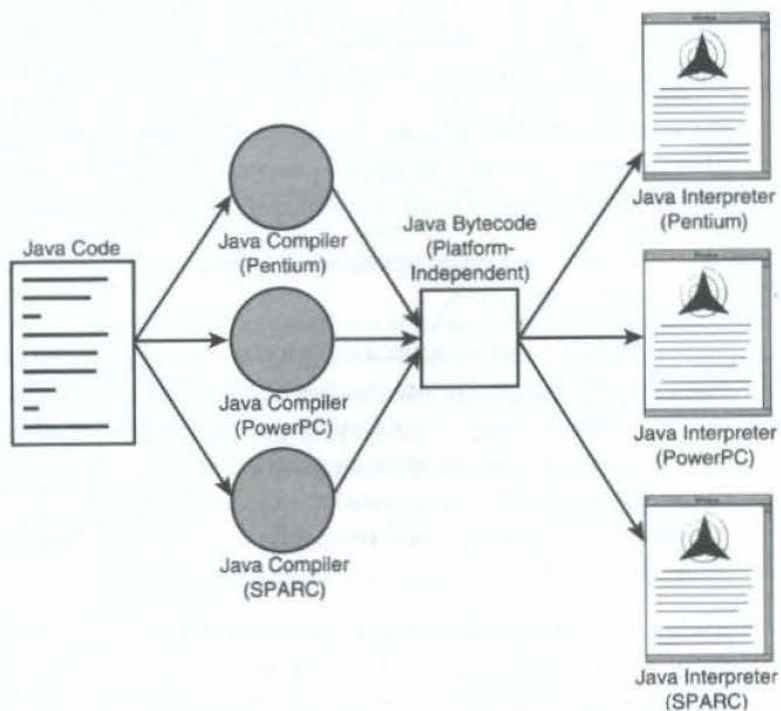
Things are different when you write code in Java. The Java development environment has two parts: a Java compiler and a Java interpreter. The Java compiler takes your Java program and instead of generating machine codes from your source files, it generates bytecodes.

Figure 1.2.
Traditional compiled programs.



To run a Java program, you run a program called a bytecode interpreter, which in turn executes your Java program (see Figure 1.3). You can either run the interpreter by itself, or—for applets—there is a bytecode interpreter built into HotJava and other Java-capable browsers that runs the applet for you.

Figure 1.3.
Java programs.





Why go through all the trouble of adding this extra layer of the bytecode interpreter? Having your Java programs in bytecode form means that instead of being specific to any one system, your programs can be run on any platform and any operating or window system as long as the Java interpreter is available. This capability of a single binary file to be executable across platforms is crucial to what enables applets to work, because the World Wide Web itself is also platform-independent. Just as HTML files can be read on any platform, so applets can be executed on any platform that is a Java-capable browser.

The disadvantage of using bytecodes is in execution speed. Because system-specific programs run directly on the hardware for which they are compiled, they run significantly faster than Java bytecodes, which must be processed by the interpreter. For many Java programs, the speed may not be an issue. If you write programs that require more execution speed than the Java interpreter can provide, you have several solutions available to you, including being able to link native code into your Java program or using tools to convert your Java bytecodes into native code. Note that by using any of these solutions, you lose the portability that Java bytecodes provide. You'll learn about each of these mechanisms on Day 20.

Java Is Object-Oriented

To some, object-oriented programming (OOP) technique is merely a way of organizing programs, and it can be accomplished using any language. Working with a real object-oriented language and programming environment, however, enables you to take full advantage of object-oriented methodology and its capabilities for creating flexible, modular programs and reusing code.

Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well. Like most object-oriented programming languages, Java includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions. These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions. Because these class libraries are written in Java, they are portable across platforms as all Java applications are.

You'll learn more about object-oriented programming and Java tomorrow.

Java Is Easy to Learn

In addition to its portability and object-orientation, one of Java's initial design goals was to be small and simple, and therefore easier to write, easier to compile, easier to debug, and, best of all, easy to learn. Keeping the language small also makes it more robust because there are fewer chances for programmers to make difficult-to-find mistakes. Despite its size and simple design, however, Java still has a great deal of power and flexibility.



An Introduction to Java Programming

Java is modeled after C and C++, and much of the syntax and object-oriented structure is borrowed from the latter. If you are familiar with C++, learning Java will be particularly easy for you, because you have most of the foundation already.

Although Java looks similar to C and C++, most of the more complex parts of those languages have been excluded from Java, making the language simpler without sacrificing much of its power. There are no pointers in Java, nor is there pointer arithmetic. Strings and arrays are real objects in Java. Memory management is automatic. To an experienced programmer, these omissions may be difficult to get used to, but to beginners or programmers who have worked in other languages, they make the Java language far easier to learn.

Getting Started with Programming in Java

Enough background! Let's finish off this day by creating two real Java programs: a stand-alone Java application and an applet that you can view in either in the appletviewer (part of the JDK) or in a Java-capable browser. Although both these programs are extremely simple, they will give you an idea of what a Java program looks like and how to compile and run it.

Getting the Software

In order to write Java programs, you will, of course, need a Java development environment. At the time this book is being written, Sun's Java Development Kit provides everything you need to start writing Java programs. The JDK is available for Sun SPARC systems running Solaris 2.3 or higher and for Windows NT and Windows 95. You can get the JDK from several places:

- ☐ The CD-ROM that came with this book contains the full 1.0 JDK distribution. See the CD information for installation instructions.
- ☐ The latest JDK can be downloaded from Sun's Java FTP site at <ftp://java.sun.com/pub/> or from a mirror site (<ftp://www.blackdown.org/pub/Java/pub/> is one).



Note: The Java Development Kit is currently in release 1.0. By the time you read this, the JDK may be available for other platforms, and other organizations may be selling Java development tools as well.

Although Netscape and other Java-enabled browsers provide an environment for running Java applets, they do not provide a mechanism for developing Java applications. For that, you need separate tools—merely having a browser is not enough.

Applets and Applications



Java programs fall into two main groups: applets and applications.

Applets, as you have learned, are Java programs that are downloaded over the World Wide Web and executed by a Web browser on the reader's machine. Applets depend on a Java-capable browser in order to run (although they can also be viewed using a tool called the appletviewer, which you'll learn about later today).

Java applications are more general programs written in the Java language. Java applications don't require a browser to run, and in fact, Java can be used to create all the kinds of applications that you would normally use a more conventional programming language to create. HotJava itself is a Java application.

A single Java program can be an applet or an application or both, depending on how you write that program and the capabilities that program uses. Throughout this first week, you'll be writing mostly HotJava applications; then you'll apply what you've learned to write applets in Week 2. If you're eager to get started with applets, be patient. Everything that you learn while you're creating simple Java applications will apply to creating applets, and it's easier to start with the basics before moving onto the hard stuff. You'll be creating plenty of applets in Week 2.

Creating a Java Application

Let's start by creating a simple Java application: the classic Hello World example that all language books use to begin.

As with all programming languages, your Java source files are created in a plain text editor, or in an editor that can save files in plain ASCII without any formatting characters. On Unix, emacs, pico, or vi will work; on Windows, Notepad or DOS Edit are both text editors.

Fire up your editor of choice, and enter the Java program shown in Listing 1.1. Type this program, as shown, in your text editor. Be careful that all the parentheses, braces, and quotes are there.

Type

Listing 1.1. Your first Java application.

```
1: class HelloWorld {
2:     public static void main (String args[]) {
3:         System.out.println("Hello World!");
4:     }
5: }
```

Warning: The numbers before each line are part of the listing and not part of the program; they're there so I can refer to specific line numbers when I explain what's going on in the program. Do not include them in your own file.



This program has two main parts:

- ☐ All the program is enclosed in a class definition—here, a class called `HelloWorld`.
- ☐ The body of the program (here, just the one line) is contained in a method (function) called `main()`. In Java applications, as in a C or C++ program, `main()` is the first method (function) that is run when the program is executed.

You'll learn more about both these parts of a Java application as the book progresses.

Once you finish typing the program, save the file. Most of the time, Java source files are named the same name as the class they define, with an extension of `.java`. This file should therefore be called `HelloWorld.java`.

Now, let's compile the source file using the Java compiler. In Sun's JDK, the Java compiler is called `javac`.

To compile your Java program, Make sure the `javac` program is in your execution path and type `javac` followed by the name of your source file:

```
javac HelloWorld.java
```



Note: In these examples, and in all the examples throughout this book, we'll be using Sun's Java compiler, part of the JDK. If you have a third-party development environment, check with the documentation for that program to see how to compile your Java programs.

The compiler should compile the file without any errors. If you get errors, go back and make sure that you've typed the program exactly as it appears in Listing 1.1.

When the program compiles without errors, you end up with a file called `HelloWorld.class`, in the same directory as your source file. This is your Java bytecode file. You can then run that bytecode file using the Java interpreter. In the JDK, the Java interpreter is called simply `java`. Make sure the `java` program is in your path and type `java` followed by the name of the file without the `.class` extension:

```
java HelloWorld
```

If your program was typed and compiled correctly, you should get the string "Hello World!" printed to your screen as a response.



Note: Remember, the Java compiler and the Java interpreter are different things. You use the Java compiler (`javac`) for your Java source files to create `.class` files, and you use the Java interpreter (`java`) to actually run your class files.

Creating a Java Applet

Creating applets is different from creating a simple application, because Java applets run and are displayed inside a Web page with other page elements and as such have special rules for how they behave. Because of these special rules for applets in many cases (particularly the simple ones), creating an applet may be more complex than creating an application.

For example, to do a simple Hello World applet, instead of merely being able to print a message, you have to create an applet to make space for your message and then use graphics operations to paint the message to the screen.

Note: If you run the Hello World application as an applet, the Hello World message prints to a special window or to a log file, depending on how the browser has screen messages set up. It will not appear on the screen unless you write your applet to put it there.

In the next example, you create that simple Hello World applet, place it inside a Web page, and view the result.

First, you set up an environment so that your Java-capable browser can find your HTML files and your applets. Much of the time, you'll keep your HTML files and your applet code in the same directory. Although this isn't required, it makes it easier to keep track of each element. In this example, you use a directory called HTML that contains all the files you'll need.

```
mkdir HTML
```

Now, open up that text editor and enter Listing 1.2.

Type

Listing 1.2. The Hello World applet.

```
1: import java.awt.Graphics;
2:
3: public class HelloWorldApplet extends java.applet.Applet {
4:
5:     public void paint(Graphics g) {
6:         g.drawString("Hello world!", 5, 25);
7:     }
8: }
```

Save that file inside your HTML directory. Just like with Java applications, give your file a name that has the same name as the class. In this case, the filename would be `HelloWorldApplet.java`.

Features to note about applets? There are a couple I'd like to point out:

- ☐ The `import` line at the top of the file is somewhat analogous to an `#include` statement in C; it enables this applet to get access to the JDK classes for creating applets and for drawing graphics on the screen.
- ☐ The `paint()` method displays the content of the applet onto the screen. Here, the string `Hello World` gets drawn. Applets use several standard methods to take the place of `main()`, which include `init()` to initialize the applet, `start()` to start it running, and `paint()` to display it to the screen. You'll learn about all of these in Week 2.

Now, compile the applet just as you did the application, using `javac`, the Java compiler.

```
javac HelloWorldApplet.java
```

Again, just as for applications, you should now have a file called `HelloWorldApplet.class` in your HTML directory.

To include an applet in a Web page, you refer to that applet in the HTML code for that Web page. Here, you create a very simple HTML file in the HTML directory (see Listing 1.3).

Type

Listing 1.3. The HTML with the applet in it.

```
1: <HTML>
2: <HEAD>
3: <TITLE>Hello to Everyone!</TITLE>
4: </HEAD><BODY>
5: <P>My Java applet says:
6: <APPLET CODE="HelloWorldApplet.class" WIDTH=150 HEIGHT=25><APPLET>
7: </BODY>
8: </HTML>
```

Analysis

You refer to an applet in your HTML files with the `<APPLET>` tag. You'll learn more about `<APPLET>` later on, but here are two things to note:

- ☐ Use the `CODE` attribute to indicate the name of the class that contains your applet.
- ☐ Use the `WIDTH` and `HEIGHT` attributes to indicate the size of the applet. The browser uses these values to know how big a chunk of space to leave for the applet on the page. Here, a box 150 pixels wide and 25 pixels high is created.

Save the HTML file in your HTML directory, with a descriptive name (for example, you might name your HTML file the same name as your applet—`HelloWorldApplet.html`).

And now, you're ready for the final test—actually viewing the result of your applet. To view the applet, you need one of the following:

- ☐ A browser that supports Java applets, such as Netscape 2.0.
- ☐ The appletviewer application, which is part of the JDK. The appletviewer is not a Web browser and won't enable you to see the entire Web page, but it's acceptable for testing to see how an applet will look and behave if there is nothing else available.



Note: Do not use the alpha version of HotJava to view your applets; applets developed with the beta JDK and onward cannot be viewed by the alpha HotJava. If, by the time you read this, there is a more recent version of HotJava, you can use that one instead.

If you're using a Java-capable browser such as Netscape to view your applet files, you can use the Open Local... item under the File menu to navigate to the HTML file containing the applet (make sure you open the HTML file and not the class file). You don't need to install anything on a Web server yet; all this works on your local system.

If you don't have a Web browser with Java capabilities built into it, you can use the appletviewer program to view your Java applet. To run appletviewer, just indicate the path to the HTML file on the command line:

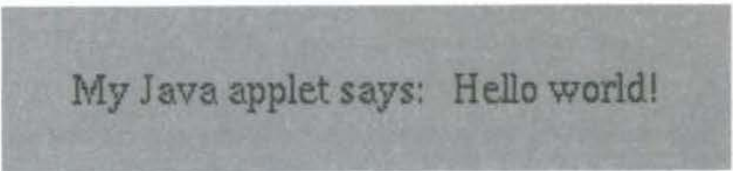
```
appletviewer HTML/HelloWorldApplet.html
```



Tip: Although you can start appletviewer from the same directory as your HTML and class files, you may not be able to reload that applet without quitting appletviewer first. If you start appletviewer from some other directory (as in the previous command line), you can modify and recompile your Java applets and then just use the Reload menu item to view the newer version.

Now, if you use the browser to view the applet, you see something similar to the image shown in Figure 1.4. If you're using appletviewer, you won't see the text around the applet (My Java applet says...), but you will see the Hello World itself.

Figure 1.4.
The Hello World applet.



My Java applet says: Hello world!

Summary

Today, you got a basic introduction to the Java language and its goals and features. Java is a programming language, similar to C or C++, in which you can develop a wide range of programs. The most common use of Java at the moment is in creating applets for HotJava, an advanced World Wide Web browser also written in Java. Applets are Java programs that are downloaded and run as part of a Web page. Applets can create animations, games, interactive programs, and other multimedia effects on Web pages.

Java's strengths lie in its portability—both at the source and at the binary level, in its object-oriented design—and in its simplicity. Each of these features help make applets possible, but they also make Java an excellent language for writing more general-purpose programs that do not require HotJava or other Java-capable browser to run. These general-purpose Java programs are called applications. HotJava itself is a Java application.

To end this day, you experimented with an example applet and an example application, getting a feel for the differences between the two and how to create, compile, and run Java programs—or, in the case of applets, how to include them in Web pages. From here, you now have the foundation to create more complex applications and applets.

Q&A

Q I'd like to use HotJava as my regular Web browser. You haven't mentioned much about HotJava today.

A The focus of this book is primarily on programming in Java and its library of standard classes, rather than on using HotJava itself. Documentation for using the HotJava browser comes with HotJava (see <http://java.sun.com/>).

Q I know a lot about HTML, but not much about computer programming. Can I still write Java programs?



A If you have no programming experience whatsoever, you most likely will find programming Java significantly more difficult. However, Java is an excellent language to learn programming with, and if you patiently work through the examples and the exercises in this book, you should be able to learn enough to get started with Java.

Q According to today's lesson, Java applets are downloaded via a Java-enabled browser such as HotJava or Netscape 2.0 and run on the reader's system. Isn't that an enormous security hole? What stops someone from writing an applet that compromises the security of my system—or worse, that damages my system?

A Sun's Java team has thought a great deal about the security of applets within Java-capable browsers and has implemented several checks to make sure applets cannot do nasty things:

- ☐ Java applets cannot read or write to the disk on the local system.
- ☐ Java applets cannot execute any programs on the local system.
- ☐ Java applets cannot connect to any machines on the Web except for the server from which they are originally downloaded.

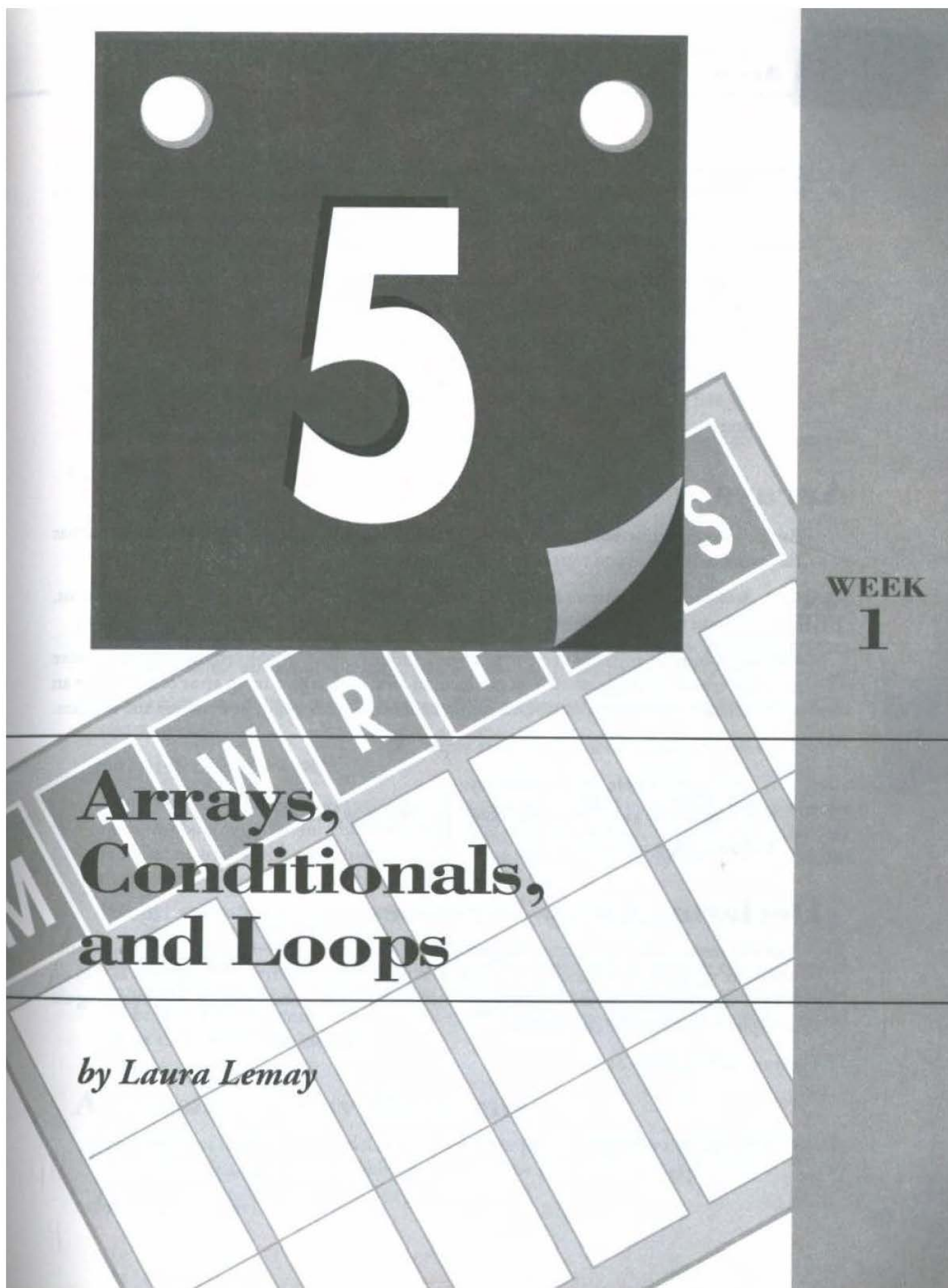
Note that some of these restrictions may be allowed in some browsers or may be turned on in the browser configuration. However, you cannot expect any of these capabilities to be available.

In addition, the Java compiler and interpreter check both the Java source code and the Java bytecodes to make sure that the Java programmer has not tried any sneaky tricks (for example, overrunning buffers or stack frames).

These checks obviously cannot stop every potential security hole (no system can promise that!), but they can significantly reduce the potential for hostile applets. You'll learn more about security issues later on in this book.

Q I followed all the directions you gave for creating a Java applet. I loaded it into HotJava, but Hello World didn't show up. What did I do wrong?

A I'll bet you're using the alpha version of HotJava to view the applet. Unfortunately, between alpha and 1.0, significant changes were made as to how applets are written. The result is that you can't view 1.0 applets (as this one was) in the alpha version of HotJava, nor can you view alpha applets in browsers that expect 1.0 applets. To view the applet, either use a different browser, or use the appletviewer application that comes with the JDK.



Although you could write Java programs using what you've learned so far, those programs would be pretty dull. Much of the good stuff in Java or in any programming language results when you have arrays to store values in and control-flow constructs (loops and conditionals) to execute different bits of a program based on tests. Today, you'll find out about the following:

- ☐ Arrays, one of the most useful objects in Java, which enable you to collect objects or primitive types into an easy-to-manage list
- ☐ Block statements, for grouping together related statements
- ☐ `if` and `switch`, for conditional tests
- ☐ `for` and `while` loops, for iteration or repeating a statement or statements multiple times

Arrays

Arrays in Java are different than they are in other languages. Arrays in Java are actual objects that can be passed around and treated just like other objects.

NEW TERM *Arrays* are a way to store a list of items. Each slot of the array holds an individual element, and you can place elements into or change the contents of those slots as you need to.

Arrays can contain any type of element value (primitive types or objects), but you can't store different types in a single array. You can have an array of integers, or an array of strings, or an array of arrays, but you can't have an array that contains, for example, both strings and integers.

To create an array in Java, you use three steps:

1. Declare a variable to hold the array.
2. Create a new array object and assign it to the array variable.
3. Store things in that array.

Declaring Array Variables

The first step to creating an array is creating a variable that will hold the array, just as you would any other variable. Array variables indicate the type of object the array will hold (just as they do for any variable) and the name of the array, followed by empty brackets (`[]`). The following are all typical array variable declarations:

```
String difficultWords[];
Point hits[];
int temps[];
```

An alternate method of defining an array variable is to put the brackets after the type instead of after the variable. They are equivalent, but this latter form is often much more readable. So, for example, these three declarations could be written like this:

```
String[] difficultWords;

Point[] hits;

int[] temps;
```

Creating Array Objects

The second step is to create an array object and assign it to that variable. There are two ways to do this:

- ☐ Using `new`
- ☐ Directly initializing the contents of that array

The first way is to use the `new` operator to create a new instance of an array:

```
String[] names = new String[10];
```

That line creates a new array of `Strings` with ten *slots*, containing elements. When you create a new array object using `new`, you must indicate how many slots that array will hold.

Array objects can contain primitive types such as integers or booleans, just as they can contain objects:

```
int[] temps = new int[99];
```

When you create an array object using `new`, all its slots are initialized for you (0 for numeric arrays, `false` for boolean, `'\0'` for character arrays, and `null` for objects). You can also create and initialize an array at the same time. Instead of using `new` to create the new array object, enclose the elements of the array inside braces, separated by commas:

```
String[] chiles = { "jalapeno", "anaheim", "serrano",
    "habanero", "thai" };
```

Each of the elements inside the braces must be of the same type and must be the same type as the variable that holds that array. An array the size of the number of elements you've included will be automatically created for you. This example creates an array of `String` objects named `chiles` that contains five elements.

Accessing Array Elements

Once you have an array with initial values, you can test and change the values in each slot of that array. To get at a value stored within an array, use the array *subscript* expression:

```
myArray[subscript];
```

The *myArray* part of this expression is a variable holding an array object, although it can also be an expression that results in an array). The *subscript* expression specifies the slot within the array to access. Array subscripts start with 0, as they do in C and C++. So, an array with ten elements has ten array slots accessed using subscript 0 to 9.

Note that all array subscripts are checked to make sure that they are inside the boundaries of the array (greater than or equal to 0 but less than the array's length) either when your Java program is compiled or when it is run. It is impossible in Java to access or assign a value to an array slot outside of the boundaries of the array. Note the following two statements, for example:

```
String[] arr = new String[10];
arr[10] = "eggplant";
```

A program with that last statement in it produces a compiler error at that line when you try to compile it. The array stored in *arr* has only ten slots numbered from 0, the element at subscript 10 doesn't exist, and the Java compiler will check for that.

If the array subscript is calculated at run-time (for example, as part of a loop) and ends up outside the boundaries of the array, the Java interpreter also produces an error (actually, to be technically correct, it throws an exception). You'll learn more about exceptions later on next week and on Day 18.

How can you keep from overrunning the end of an array accidentally in your own programs? You can test for the length of the array in your programs using the *length* instance variable—it's available for all array objects, regardless of type:

```
int len = arr.length // returns 10
```

Changing Array Elements

To assign an element value to a particular array slot, merely put an assignment statement after the array access expression:

```
myarray[1] = 15;
sentence[0] = "The";
sentence[10] = sentence[0];
```

An important thing to note is that an array of objects in Java is an array of references to those objects (similar in some ways to an array of pointers in C or C++). When you assign a value to a slot in an array, you're creating a reference to that object, just as you do for a plain variable. When you move values around inside arrays (as in that last line), you just reassign the reference; you don't copy the value from one slot to another. Arrays of primitive types such as *ints* or *floats* do copy the values from one slot to another.

Arrays of references to objects, as opposed to the objects themselves, are particularly useful because it means you can have multiple references to the same objects both inside and outside arrays—for example, you can assign an object contained in an array to a variable and refer to that same object by using either the variable or the array position.

Multidimensional Arrays

Java does not support multidimensional arrays. However, you can declare and create an array of arrays (and those arrays can contain arrays, and so on, for however many dimensions you need), and access them as you would C-style multidimensional arrays:

```
int coords[][] = new int[12][12];
coords[0][0] = 1;
coords[0][1] = 2;
```

Block Statements

A block statement is a group of other statements surrounded by braces (`{}`). You can use a block anywhere a single statement would go, and the new block creates a new local scope for the statements inside it. This means that you can declare and use local variables inside a block, and those variables will cease to exist after the block is finished executing. For example, here's a block inside a method definition that declares a new variable `y`. You cannot use `y` outside the block in which it's declared:

```
void testblock() {
    int x = 10;
    { // start of block
        int y = 50;
        System.out.println("inside the block:");
        System.out.println("x: " + x);
        System.out.println("y: " + y);
    } // end of block
}
```

Blocks are not usually used in this way—alone in a method definition. You've mostly seen blocks up to this point surrounding class and method definitions, but another very common use of block statements is in the control flow constructs you'll learn about in the remainder of today's lesson.

5

if Conditionals

The `if` conditional, which enables you to execute different bits of code based on a simple test in Java, is nearly identical to `if` statements in C. `if` conditionals contain the keyword `if`, followed by a boolean test, followed by a statement (often a block statement) to execute if the test is true:

```
if (x < y)
    System.out.println("x is smaller than y");
```

An optional `else` keyword provides the statement to execute if the test is false:

```
if (x < y)
    System.out.println("x is smaller than y");
else System.out.println("y is bigger");
```

Technical Note: The difference between `if` conditionals in Java and C or C++ is that the test must return a boolean value (true or false). Unlike in C, the test cannot return an integer.

```
if (engineState == true )
    System.out.println("Engine is already on.");
else {
    System.out.println("Now starting Engine.");
    if (gasLevel >= 1)
        engineState = true;
    else System.out.println("Low on gas! Can't start engine.");
}
```

This example uses the test `(engineState == true)`. For boolean tests of this type, a common shortcut is merely to include the first part of the expression, rather than explicitly testing its value against true or false:

```
if (engineState)
    System.out.println("Engine is on.");
else System.out.println("Engine is off.");
```

The Conditional Operator

An alternative to using the `if` and `else` keywords in a conditional statement is to use the conditional operator, sometimes called the ternary operator.

NEW TERM The *conditional operator* is a *ternary operator* because it has three terms.

The conditional operator is an expression, meaning that it returns a value (unlike the more general `if`, which only can result in a statement or block being executed). The conditional operator is most useful for very short or simple conditionals, and looks like this:

```
test ? trueresult : falseresult
```

The *test* is an expression that returns true or false, just like the test in the `if` statement. If the test is true, the conditional operator returns the value of *trueresult*; if it's false, it returns the value of *falseresult*. For example, the following conditional tests the values of `x` and `y`, returns the smaller of the two, and assigns that value to the variable `smaller`:

```
int smaller = x < y ? x : y;
```

The conditional operator has a very low precedence; that is, it's usually evaluated only after all its subexpressions are evaluated. The only operators lower in precedence are the assignment operators. See the precedence chart in Day 3's lesson for a refresher on precedence of all the operators.

switch Conditionals

A common programming practice in any language is to test a variable against some value, and if it doesn't match that value, to test it again against a different value, and if it doesn't match that one to make yet another test, and so on. Using only `if` statements, this can become unwieldy, depending on how it's formatted and how many different options you have to test. For example, you might end up with a set of `if` statements something like this or longer:

```
if (oper == '+')
    addargs(arg1, arg2);
else if (oper == '-')
    subargs(arg1, arg2);
else if (oper == '*')
    multargs(arg1, arg2);
else if (oper == '/')
    divargs(arg1, arg2);
```

This form of `if` statement is called a nested `if`, because each `else` statement in turn contains yet another `if`, and so on, until all possible tests have been made.

A common shorthand mechanism for nested `ifs` that you can use in some cases allows you to group tests and actions together in a single statement. This is the `switch` or `case` statement; in Java it's `switch` and behaves as it does in C:

```
switch (test) {
    case valueOne:
        resultOne;
        break;
    case valueTwo:
        resultTwo;
        break;
    case valueThree:
        resultThree;
        break;
    ...
    default: defaultresult;
}
```



In the `switch` statement, the test (a primitive type of `byte`, `char`, `short`, or `int`) is compared with each of the case values in turn. If a match is found, the statement, or statements after the test is executed. If no match is found, the `default` statement is executed. The `default` is optional, so if there isn't a match in any of the cases and `default` doesn't exist, the `switch` statement completes without doing anything.

Note that the significant limitation of the `switch` in Java is that the tests and values can be only simple primitive types (and then only primitive types that are castable to `int`). You cannot use larger primitive types (`long`, `float`), strings, or other objects within a `switch`, nor can you test for any relationship other than equality. This limits the usefulness of `switch` to all but the simplest cases; nested `ifs` can work for any kind of test on any type.

Here's a simple example of a switch statement similar to the nested if shown earlier:

```
switch (oper) {
    case '+':
        addargs(arg1, arg2);
        break;
    case '*':
        subargs(arg1, arg2);
        break;
    case '-':
        multargs(arg1, arg2);
        break;
    case '/':
        divargs(arg1, arg2);
        break;
}
```

Note the `break` statement included in every line. Without the explicit `break`, once a match is made, the statements for that match and also all the statements further down in the `switch` are executed until a `break` or the end of the `switch` is found (and then execution continues after the end of the `switch`). In some cases, this may be exactly what you want to do, but in most cases, you'll want to make sure to include the `break` so that only the statements you want to be executed are executed.

One handy use of falling through occurs when you want multiple values to execute the same statements. In this instance, you can use multiple case lines with no result, and the `switch` will execute the first statements it finds. For example, in the following `switch` statement, the string "x is an even number." is printed if x has values of 2, 4, 6, or 8. All other values of x print the string "x is an odd number."

```
switch (x) {
    case 2:
    case 4:
    case 6:
    case 8:
        System.out.println("x is an even number.");
        break;
    default: System.out.println("x is an odd number.");
}
```

for Loops

The `for` loop, as in C, repeats a statement or block of statements some number of times until a condition is matched. `for` loops are frequently used for simple iteration in which you repeat a block of statements a certain number of times and then stop, but you can use `for` loops for just about any kind of loop.

The `for` loop in Java looks roughly like this:

```
for (initialization; test; increment){
```

```
    statements;
}
```

The start of the `for` loop has three parts:

- ☐ **initialization** is an expression that initializes the start of the loop. If you have a loop index, this expression might declare and initialize it, for example, `int i = 0`. Variables that you declare in this part of the `for` loop are local to the loop itself; they cease existing after the loop is finished executing.
- ☐ **test** is the test that occurs after each pass of the loop. The test must be a boolean expression or function that returns a boolean value, for example, `i < 10`. If the test is true, the loop executes. Once the test is false, the loop stops executing.
- ☐ **increment** is any expression or function call. Commonly, the increment is used to change the value of the loop index to bring the state of the loop closer to returning false and completing.

The statement part of the `for` loop is the statements that are executed each time the loop iterates. Just as with `if`, you can include either a single statement here or a block; the previous example used a block because that is more common. Here's an example of a `for` loop that initializes all the values of a `String` array to null strings:

```
String strArray[] = new String[10];
int i; // loop index

for (i = 0; i < strArray.length; i++)
    strArray[i] = "";
```

Any of the parts of the `for` loop can be empty statements, that is, you can simply include a semicolon with no expression or statement, and that part of the `for` loop will be ignored. Note that if you do use a null statement in your `for` loop, you may have to initialize or increment any loop variables or loop indices yourself elsewhere in the program.

You can also have an empty statement for the body of your `for` loop, if everything you want to do is in the first line of that loop. For example, here's one that finds the first prime number higher than 4000:

```
for (i = 4001; notPrime(i); i += 2)
    ;
```

Note that a common mistake in C that also occurs in Java is to accidentally put a semicolon after the first line of the `for` loop:

```
for (i = 0; i < 10; i++);
    System.out.println("Loop!");
```

Because the first semicolon ends the loop with an empty statement, the loop doesn't actually do anything. The `println` function will be printed only once, because it's actually outside the `for` loop entirely. Be careful not to make this mistake in your own Java programs.



while and *do* Loops

Finally, there are *while* and *do* loops. *while* and *do* loops, like *for* loops, enable a block of Java code to be executed repeatedly until a specific condition is met. Whether you use a *for* loop, a *while*, or a *do* is mostly a matter of your programming style.

while and *do* loop, are exactly the same as in C and C++ except their test condition must be a *boolean*.

while Loops

The *while* loop is used to repeat a statement or block of statements as long as a particular condition is true. *while* loops look like this:

```
while (condition){
    bodyOfLoop;
}
```

The *condition* is a *boolean* expression. If it returns *true*, the *while* loop executes the statements in *bodyOfLoop* and then tests the condition again, repeating until the condition is *false*. I've shown the *while* loop here with a block statement, because it's most commonly used, although you can use a single statement in place of the block.

Here's an example of a *while* loop that copies the elements of an array of integers (in *array1*) to an array of floats (in *array2*), casting each element to a *float* as it goes. The one catch is that if any of the elements in the first array is 0, the loop will immediately exit at that point. To cover both the cases wherein all the elements have been copied and an element is 0, you can use a compound test with the *&&* operator:

```
int count = 0;
while ( count < array1.length && array1[count] !=0) {
    array2[count] = (float) array1[count++];
}
```

Note that if the condition is initially false the first time it is tested (for example, if the first element in that first array is 0), the body of the *while* loop will never be executed. If you need to execute the loop at least once, you can do one of two things:

- ☐ Duplicate the body of the loop outside the *while* loop.
- ☐ Use a *do* loop (described below).

The *do* loop is considered the better solution of the two.

do...while Loops

The `do` loop is just like a `while` loop, except that `do` executes a given statement or block until the condition is `false`. The main difference is that `while` loops test the condition before looping, making it possible that the body of the loop will never execute if the condition is `false` the first time it's tested. `do` loops run the body of the loop at least once before testing the condition. `do` loops look like this:

```
do {
    bodyOfLoop;
} while (condition);
```

Here, the `bodyOfLoop` part is the statements that are executed with each iteration. It's shown here with a block statement because it's most commonly used that way, but you can substitute the braces for a single statement as you can with the other control-flow constructs. The condition is a boolean test. If it returns `true`, the loop is run again. If it returns `false`, the loop exits. Keep in mind that with `do` loops, the body of the loop executes at least once.

Here's a simple example of a `do` loop that prints a message each time the loop iterates:

```
int x = 1;
do {
    System.out.println("Looping, round " + x);
    x++;
} while (x <= 10);
```

Here's the output of these statements:



```
Looping, round 1
Looping, round 2
Looping, round 3
Looping, round 4
Looping, round 5
Looping, round 6
Looping, round 7
Looping, round 8
Looping, round 9
Looping, round 10
```



Breaking Out of Loops

In all the loops (`for`, `while`, and `do`), the loop ends when the condition you're testing for is met. What happens if something odd occurs within the body of the loop and you want to exit the loop early? For that, you can use the `break` and `continue` keywords.

You've already seen `break` as part of the `switch` statement; it stops execution of the `switch`, and the program continues. The `break` keyword, when used with a loop, does the same thing—it immediately halts execution of the current loop. If you've nested loops within loops, execution picks up in the next outer loop; otherwise, the program merely continues executing the next statement after the loop.

For example, take that `while` loop that copied elements from an integer array into an array of floats until the end of the array or until a `0` is reached. You can test for that latter case inside the body of the `while` and then use a `break` to exit the loop:

```
int count = 0;
while (count < array1.length) {
    if (array1[count] == 0) {
        break;
    }
    array2[count] = (float) array1[count++];
}
```

`continue` is similar to `break` except that instead of halting execution of the loop entirely, the loop starts over at the next iteration. For `do` and `while` loops, this means the execution of the block starts over again; for `for` loops, the increment expression is evaluated and then the block is executed. `continue` is useful when you want to special-case elements within a loop. With the previous example of copying one array to another, you can test for whether the current element is `0` and restart the loop if you find it so that the resulting array will never contain zero. Note that because you're skipping elements in the first array, you now have to keep track of two different array counters:

```
int count1 = 0;
int count2 = 0;
while (count1 < array1.length) {
    if (array1[count1] == 0)
        continue;

    array2[count2++] = (float)array1[count1++];
}
```

Labeled Loops

Both `break` and `continue` can have an optional label that tells Java where to break to. Without a label, `break` jumps outside the nearest loop (to an enclosing loop or to the next statement outside the loop), and `continue` restarts the enclosing loop. Using labeled breaks and continues enables you to break outside nested loops or to continue a loop outside the current loop.

To use a labeled loop, add the label before the initial part of the loop, with a colon between them. Then, when you use `break` or `continue`, add the name of the label after the keyword itself:

```
out:
    for (int i = 0; i < 10; i++) {
        while (x < 50) {
            if (i * x == 400)
                break out;
            ...
        }
        ...
    }
```

In this snippet of code, the label `out` labels the outer loop. Then, inside both the `for` and the `while` loop, when a particular condition is met, a `break` causes the execution to break out of both loops.

Here's another example: the following program contains a nested `for` loop. Inside the innermost loop, if the summed values of the two counters is greater than four, both loops exit at once:

```
foo:
    for (int i = 1; i <= 5; i++)
        for (int j = 1; j <= 3; j++) {
            System.out.println("i is " + i + ", j is " + j);
            if ((i + j) > 4)
                break foo;
        }
    System.out.println("end of loops");
```

Here's the output from this program:

Output

```
i is 1, j is 1
i is 1, j is 2
i is 1, j is 3
i is 2, j is 1
i is 2, j is 2
i is 2, j is 3
end of loops
```

Analysis As you can see, the loop iterated until the sum of `i` and `j` was greater than 4, and then both loops exited back to the outer block and the final message was printed.

Summary

Today, you learned about three main topics that you'll most likely use quite often in your own Java programs: arrays, conditionals, and loops.

You learned how to declare an array variable, create and assign an array object to that variable, and access and change elements within that array.

Conditionals include the `if` and `switch` statements, with which you can branch to different parts

of your program based on a boolean test.

Finally, you learned about the `for`, `while`, and `do` loops, each of which enable you to execute a portion of your program repeatedly until a given condition is met.

Now that you've learned the small stuff, all that's left is to go over the bigger issues of declaring classes and creating methods within which instances of those classes can communicate with each other by calling methods. Get to bed early tonight, because tomorrow is going to be a wild ride.

Q&A

Q If arrays are objects, and you use `new` to create them, and they have an instance variable `length`, where is the `Array` class? I didn't see it in the Java class libraries.

A Arrays are implemented kind of weirdly in Java. The `Array` class is constructed automatically when your Java program runs; `Array` provides the basic framework for arrays, including the `length` variable. Additionally, each primitive type and object has an implicit subclass of `Array` that represents an array of that class or object. When you create a new array object, it may not have an actual class, but it behaves as if it does.

Q Does Java have `gotos`?

A The Java language defines the keyword `goto`, but it is not currently used for anything. In other words, no, Java does not have `gotos`.

Q I declared a variable inside a block statement for an `if`. When the `if` was done, the definition of that variable vanished. Where did it go?

A In technical terms, block statements inside braces form a new lexical scope. What this means is that if you declare a variable inside a block, it's only visible and usable inside that block. Once the block finishes executing, all the variables you declared go away. It's a good idea to declare most of your variables in the outermost block in which they'll be needed—usually at the top of a block statement. The exception might be very simple variables, such as index counters in `for` loops, where declaring them in the first line of the `for` loop is an easy shortcut.

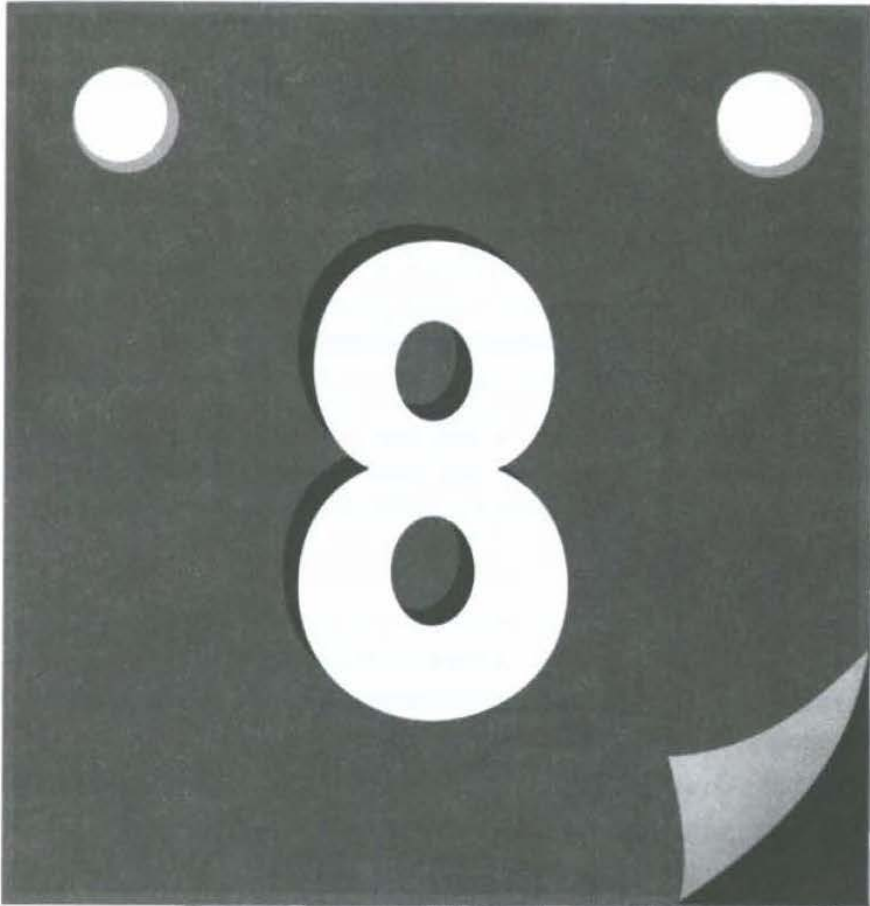
You'll learn more about variables and scope tomorrow.

Q Why can't you use `switch` with strings?

A Strings are objects, and `switch` in Java works only for the primitive types `byte`, `char`, `short`, and `int`. To compare strings, you have to use nested `ifs`, which enable more general expression tests, including string comparison.

Q It seems to me that a lot of `for` loops could be written as `while` loops, and vice versa.

A True. The `for` loop is actually a special case of `while` that enables you to iterate a loop a specific number of times. You could just as easily do this with a `while` and then increment a counter inside the loop. Either works equally well. This is mostly just a question of programming style and personal choice.



8

WEEK
2

Java Applet Basics

by Laura Lemay

Much of Java's current popularity has come about because of Java-capable World Wide Web browsers and their support for applets: small programs that run inside a Web page and can be used to create dynamic, interactive Web designs. Applets, as I noted at the beginning of this book, are written in the Java language, and can be viewed in any browser that supports Java, including Sun's HotJava and Netscape's Navigator 2.0. Learning how to create applets is most likely the reason you bought this book, so let's waste no more time.

Last week, you focused on learning about the Java language itself, and most of the little programs you created were Java applications. This week, now that you have the basics down, you move on to creating and using applets, which includes a discussion of many of the classes in the standard Java class library.

Today, you'll start with the basics:

- ☐ A small review of differences between Java applets and applications
- ☐ Getting started with applets: the basics of how an applet works and how to create your own simple applets
- ☐ Including an applet on a Web page by using the `<APPLET>` tag, including the various features of that tag
- ☐ Passing parameters to applets

How Applets and Applications Are Different

Although you explored the differences between Java applications and Java applets in the early part of this book, let's review them.

In short, Java applications are stand-alone Java programs that can be run by using just the Java interpreter, for example, from a command line. Most everything you've used up to this point in the book has been a Java application, albeit a simple one.

Java applets, however, are run from inside a World Wide Web browser. A reference to an applet is embedded in a Web page using a special HTML tag. When a reader, using a Java-enabled browser, loads a Web page with an applet in it, the browser downloads that applet from a Web server and executes it on the local system (the one the browser is running on).

Because Java applets run inside a Java browser, they have the advantage of the structure the browser provides: an existing window, an event-handling and graphics context, and the surrounding user interface. Java applications can also create this structure, but they don't require it (you'll learn how to create Java applications that use applet-like graphics and UI features on Day 14).

The convenience that applets have over applications in terms of structure and UI capabilities, however, is hampered by restrictions on what applets can do. Given the fact that Java applets can be downloaded from anywhere and run on a client's system, restrictions are necessary to prevent an applet from causing system damage or security breaches. Without these restrictions in place, Java applets could be written to contain viruses or trojan horses (programs that seem friendly but do some sort of damage to the system), or be used to compromise the security of the system that runs them. The restrictions on what an applet can do include the following:

- ☐ Applets can't read or write to the reader's file system, except in specific directories (which are defined by the user through an access control list that, by default, is empty). Some browsers may not even allow an applet to read or write to the file system at all, or at the same time as using the network.
- ☐ Applets can't usually communicate with a server other than the one that had originally stored the applet. (This may be configurable by the browser; however, you should not depend on having this behavior available.)
- ☐ Applets can't run any programs on the reader's system. For Unix systems, this includes forking a process.
- ☐ Applets can't load programs native to the local platform, including shared libraries such as DLLs.

In addition, Java itself includes various forms of security and consistency checking in the Java compiler and interpreter to prevent unorthodox use of the language (you'll learn more about this on Day 21). This combination of restrictions and security features make it more difficult for a rogue Java applet to do damage to the client's system.

Note: These restrictions prevent all of the traditional ways of causing damage to a client's system, but it's impossible to be absolutely sure that a clever programmer cannot somehow work around these restrictions, violate privacy, use CPU resources, and be annoying. Sun has asked the Net at large to try to break Java's security and to create an applet that can work around the restrictions imposed on it. You'll learn about more issues in Java security on Day 21.

Creating Applets

For the most part, all the Java programs you've created up to this point have been Java applications—simple programs with a single `main()` method that created objects, set instance variables, and ran methods. Today and in the days following, you'll be creating applets

exclusively, so you will need a good grasp of how an applet works, the sorts of features an applet has, and where to start when you first create your own applets. Without further ado, let's get on with it.

To create an applet, you create a subclass of the class `Applet`, in the `java.applet` package. The `Applet` class provides behavior to enable your applet not only to work within the browser itself, but also to take advantage of the capabilities of AWT to include UI elements, to handle mouse and keyboard events, and to draw to the screen. Although your applet can have as many "helper" classes as it needs, it's the main applet class that triggers the execution of the applet. That initial applet class always has a signature like this:

```
public class myClass extends java.applet.Applet {
    ...
}
```

Note the `public` keyword. Java requires that your applet subclass be declared `public`. Again, this is true only of your main applet class; any helper classes you create can be `public` or `private` as you wish. `Public`, `private`, and other forms of access control are described on Day 15.

When Java encounters your applet in a Web page, it loads your initial applet class over the network, as well as any other helper classes that first class uses. Unlike with applications, where Java calls the `main()` method directly on your initial class, when your applet is loaded, Java creates an instance of that class, and a series of `Applet` methods are called on that instance. Different applets that use the same class use different instances, so each one can behave differently from the other applets running in the same browser.

Major Applet Activities

To create a basic Java application, your class has to have one method, `main()`, with a specific signature. Then, when your application starts up, `main()` is executed, and from `main()` you can set up the behavior that your program needs. Applets are similar but more complicated. Applets have many different activities that correspond to various major events in the life cycle of the applet—for example, initialization, painting, or mouse events. Each activity has a corresponding method, so when an event occurs, the browser or other Java-capable tool calls those specific methods.

The default implementations of these activity methods do nothing; to provide behavior for an event you must override the appropriate method in your applet's subclass. You don't have to override all of them, of course; different applet behavior requires different methods to be overridden.

You'll learn about the various important methods to override as the week progresses, but, for a general overview, here are five of the more important methods in an applet's execution: initialization, starting, stopping, destroying, and painting.

Initialization

Initialization occurs when the applet is first loaded (or reloaded). Initialization might include creating the objects it needs, setting up an initial state, loading images or fonts, or setting parameters. To provide behavior for the initialization of your applet, override the `init()` method:

```
public void init() {
    ...
}
```

Starting

After an applet is initialized, it is started. Starting can also occur if the applet was previously stopped. For example, an applet is stopped if the reader follows a link to a different page, and it is started again when the reader returns to this page. Note that starting can occur several times during an applet's life cycle, whereas initialization happens only once. To provide startup behavior for your applet, override the `start()` method:

```
public void start() {
    ...
}
```

Functionality that you put in the `start()` method might include starting up a thread to control the applet, sending the appropriate messages to helper objects, or in some way telling the applet to begin running. You'll learn more about starting applets on Day 10.

Stopping

Stopping and starting go hand in hand. Stopping occurs when the reader leaves the page that contains a currently running applet, or you can stop the applet yourself by calling `stop()`. By default, when the reader leaves a page any threads the applet had started will continue running. You'll learn more about threads in Day 10. By overriding `stop()`, you can suspend execution of these threads and then restart them if the applet is viewed again:

```
public void stop() {
    ...
}
```

Destroying

Destroying sounds more violent than it is. Destroying enables the applet to clean up after itself just before it is freed or the browser exits—for example, to kill any running threads or to release any other running objects. Generally, you won't want to override `destroy()` unless you have specific resources that need to be released—for example, threads that the applet has created. To provide clean up behavior for your applet, override the `destroy()` method:

```
public void destroy() {
    ...
}
```

Technical Note: How is `destroy()` different from `finalize()`, which was described on Day 7? First, `destroy()` applies only to applets. `finalize()` is a more general-purpose way for a single object of any type to clean up after itself.

Painting

Painting is how an applet actually draws something on the screen, be it text, a line, a colored background, or an image. Painting can occur many hundreds of times during an applet's life cycle—for example, once after the applet is initialized, if the browser is placed behind another window on the screen and then brought forward again, if the browser window is moved to a different position on the screen, or perhaps repeatedly in the case of animations. You override the `paint()` method for your applet to have an actual appearance on the screen. The `paint()` method looks like this:

```
public void paint(Graphics g) {
    ...
}
```

Note that unlike the other major methods in this section, `paint()` takes an argument, an instance of the class `Graphics`. This object is created and passed to `paint` by the browser, so you don't have to worry about it. However, you will have to make sure that the `Graphics` class (part of the `java.awt` package) gets imported into your applet code, usually through an `import` statement at the top of your Java file:

```
import java.awt.Graphics;
```

A Simple Applet

On Day 2, you created a simple applet called `HelloAgainApplet` (this was the one with the big red `Hello Again`). There, you created and used that applet as an example of creating a subclass. Let's go over the code for that applet again, this time looking at it slightly differently in light of the things you just learned about applets. Listing 8.1 shows the code for that applet.

Type

Listing 8.1. The Hello Again applet.

```

1: import java.awt.Graphics;
2: import java.awt.Font;
3: import java.awt.Color;
4:
5: public class HelloAgainApplet extends java.applet.Applet {
6:
7:     Font f = new Font("TimesRoman", Font.BOLD, 36);
8:
9:     public void paint(Graphics g) {
10:         g.setFont(f);
11:         g.setColor(Color.red);
12:         g.drawString("Hello again!", 5, 50);
13:     }
14: }

```

8

Analysis

This applet overrides `paint()`, one of the major methods described in the previous section. Because the applet doesn't actually do much (all it does is print a couple of words to the screen), and there's not really anything to initialize, you don't need a `start()` or a `stop()` or an `init()` method.

The `paint` method is where the real work of this applet (what little work goes on) really occurs. The `Graphics` object passed into the `paint()` method holds the graphics state—that is, the current features of the drawing surface. Lines 10 and 11 set up the font and color for this graphics state (here, the font object held in the `f` instance variable, and an object representing the color red that's stored in the `Color` class's variable `red`).

Line 12 then draws the string "Hello Again!" by using the current font and color at the position 5, 50. Note that the 0 point for `x`, `y` is at the top left of the applet's drawing surface, with positive `y` moving downward, so 50 is actually at the bottom of the applet. Figure 8.1 shows how the applet's bounding box and the string are drawn on the page.

Figure 8.1.
Drawing the applet.



Including an Applet on a Web Page

After you create a class or classes that contain your applet and compile them into class files as you would any other Java program, you have to create a Web page that will hold that applet by using the HTML language. There is a special HTML tag for including applets in Web pages; Java-capable browsers use the information contained in that tag to locate the compiled class files and execute the applet itself. In this section, you'll learn about how to put Java applets in a Web page and how to serve those files to the Web at large.



Note: The following section assumes you have at least a passing understanding of writing HTML pages. If you need help in this area, you may find the book *Teach Yourself Web Publishing with HTML in 14 Days* useful. It is also from Sams.Net (and also written by one of the authors of this book).

The <APPLET> Tag

To include an applet on a Web page, use the <APPLET> tag. <APPLET> is a special extension to HTML for including applets in Web pages. Listing 8.2 shows a very simple example of a Web page with an applet included in it.

Type

Listing 8.2. A simple HTML page.

```
1: <HTML>
2: <HEAD>
3: <TITLE>This page has an applet on it</TITLE>
4: </HEAD>
5: <BODY>
6: <P>My second Java applet says:
7: <BR>
8: <APPLET CODE="HelloAgainApplet.class" WIDTH=200 HEIGHT=50>
9: There would be an applet here if your browser
10: supported Java.
11: </APPLET>
12: </BODY>
13: </HTML>
```

Analysis

There are three things to note about the <APPLET> tag in this page:

- ☐ The CODE attribute indicates the name of the class file that contains this applet, including the .class extension. In this case, the class file must be in the same directory

as this HTML file. To indicate applets are in a specific directory, use CODEBASE, described later today.

- ☐ WIDTH and HEIGHT are required and are used to indicate the bounding box of the applet—that is, how big a box to draw for the applet on the Web page. Be sure you set WIDTH and HEIGHT to be an appropriate size for the applet; depending on the browser, if your applet draws outside the boundaries of the space you've given it, you may not be able to see or get to those parts of the applet outside the bounding box.
- ☐ The text between the <APPLET> and </APPLET> tags is displayed by browsers that do not understand the <APPLET> tag (which includes most browsers that are not Java-capable). Because your page may be viewed in many different kinds of browsers, it is a very good idea to include alternate text here so that readers of your page who don't have Java will see something other than a blank line. Here, you include a simple statement that says There would be an applet here if your browser supported Java.

Note that the <APPLET> tag, like the tag, is not itself a paragraph, so it should be enclosed inside a more general text tag, such as <P> or one of the heading tags (<H1>, <H2>, and so on).

Testing the Result

Now with a class file and an HTML file that refers to your applet, you should be able to load that HTML file into your Java-capable browser (using either the Open File... menu item or a file URL, or by indicating the filename on a command line). The browser loads and parses your HTML file, and then loads and executes your applet class.

Figure 8.2 shows the Hello Again applet, in case you've forgotten what it looks like.

Figure 8.2.
The Hello Again applet.

Hello again!

Making Java Applets Available to the Web

After you have an applet and an HTML file, and you've verified that everything is working correctly on your local system, the last step is making that applet available to the World Wide Web at large so that anyone with a Java-capable browser can view that applet.

Java applets are served by a Web server the same way that HTML files, images, and other media are. You don't need special server software to make Java applets available to the Web; you don't even need to configure your server to handle Java files. If you have a Web server up and running, or space on a Web server available to you, all you have to do is move your HTML and compiled class files to that server, as you would any other file.

If you don't have a Web server, you have to rent space on one or set one up yourself. (Web server setup and administration, as well as other facets of Web publishing in general, are outside the scope of this book.)

More About the `<APPLET>` Tag

In its simplest form, by using `CODE`, `WIDTH`, and `HEIGHT`, the `<APPLET>` tag merely creates a space of the appropriate size and then loads and runs the applet in that space. The `<APPLET>` tag, however, does include several attributes that can help you better integrate your applet into the overall design of your Web page.



Note: The attributes available for the `<APPLET>` tag are almost identical to those for the HTML `<IMG>` tag.

ALIGN

The `ALIGN` attribute defines how the applet will be aligned on the page. This attribute can have one of nine values: `LEFT`, `RIGHT`, `TOP`, `TEXTTOP`, `MIDDLE`, `ABSMIDDLE`, `BASELINE`, `BOTTOM`, and `ABSBOTTOM`.

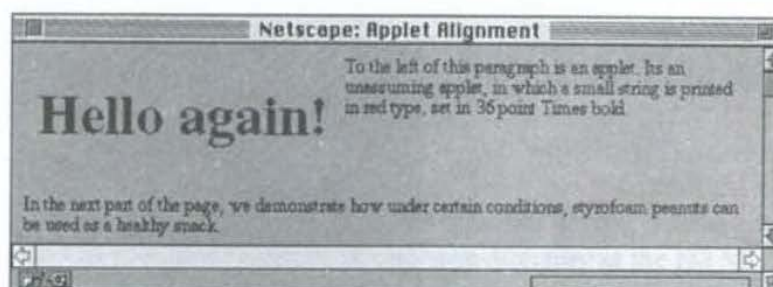
In the case of `ALIGN=LEFT` and `ALIGN=RIGHT`, the applet is placed at the left or right margins of the page, respectively, and all text following that applet flows in the space to the right or left of that applet. The text will continue to flow in that space until the end of the applet, or you can use a line break tag (`<BR>`) with the `CLEAR` attribute to start the left line of text below that applet. The `CLEAR` attribute can have one of three values: `CLEAR=LEFT` starts the text at the next clear left margin, `CLEAR=RIGHT` does the same for the right margin, and `CLEAR=ALL` starts the text at the next line where both margins are clear.

For example, here's a snippet of HTML code that aligns an applet against the left margin, has some text flowing alongside it, and then breaks at the end of the paragraph so that the next bit of text starts below the applet:

```
<P><APPLET CODE="HelloAgainApplet" WIDTH=300 HEIGHT=200
ALIGN=LEFT>Hello Again!</APPLET>
To the left of this paragraph is an applet. It's a
simple, unassuming applet, in which a small string is
printed in red type, set in 36 point Times bold.
<BR CLEAR=ALL>
<P>In the next part of the page, we demonstrate how
under certain conditions, styrofoam peanuts can be
used as a healthy snack.
```

Figure 8.3 shows how this applet and the text surrounding it might appear in a Java-capable browser.

Figure 8.3.
An applet aligned left.

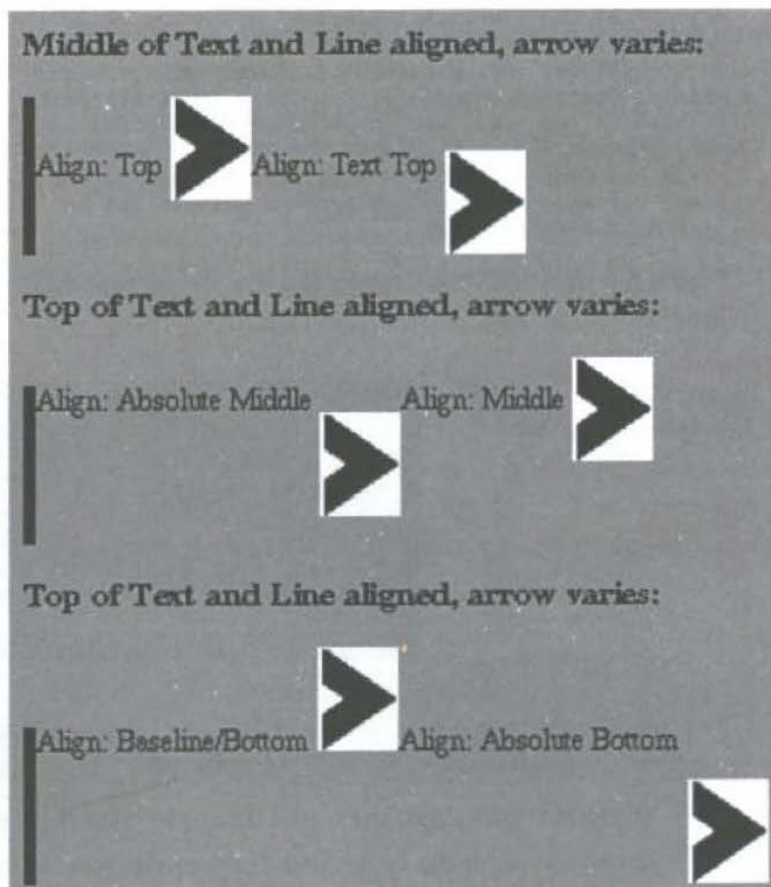


For smaller applets, you may want to include your applet within a single line of text. To do this, there are seven values for ALIGN that determine how the applet is vertically aligned with the text:

- ☐ ALIGN=TEXTTOP aligns the top of the applet with the top of the tallest text in the line.
- ☐ ALIGN=TOP aligns the applet with the topmost item in the line (which may be another applet, or an image, or the top of the text).
- ☐ ALIGN=ABSMIDDLE aligns the middle of the applet with the middle of the largest item in the line.
- ☐ ALIGN=MIDDLE aligns the middle of the applet with the middle of the baseline of the text.
- ☐ ALIGN=BASELINE aligns the bottom of the applet with the baseline of the text.
ALIGN=BASELINE is the same as ALIGN=BOTTOM, but ALIGN=BASELINE is a more descriptive name.
- ☐ ALIGN=ABSBOTTOM aligns the bottom of the applet with the lowest item in the line (which may be the baseline of the text or another applet or image).

Figure 8.4 shows the various alignment options, where the line is an image and the arrow is a small applet.

Figure 8.4.
Applet alignment options.



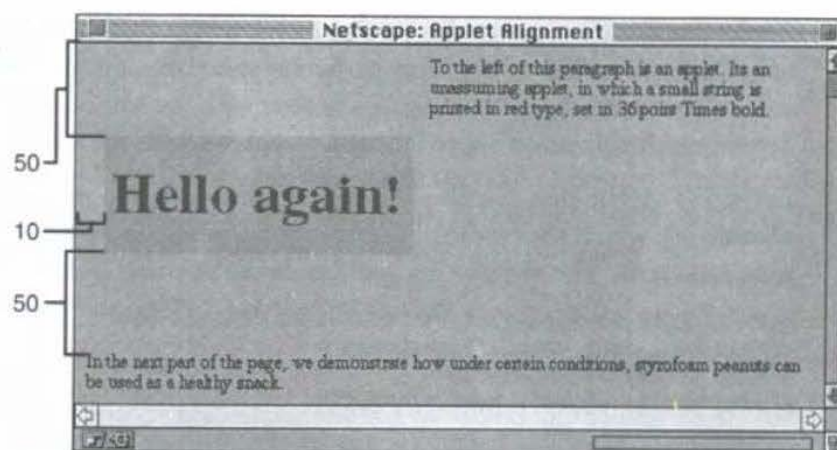
HSPACE and VSPACE

The `HSPACE` and `VSPACE` attributes are used to set the amount of space, in pixels, between an applet and its surrounding text. `HSPACE` controls the horizontal space (the space to the left and right of the applet). `VSPACE` controls the vertical space (the space above and below). For example, here's that sample snippet of HTML with vertical space of 50 and horizontal space of 10:

```
<P><APPLET CODE="HelloAgainApplet" WIDTH=300 HEIGHT=200
ALIGN=LEFT VSPACE=50 HSPACE=10>Hello Again!</APPLET>
To the left of this paragraph is an applet. It's a
simple, unassuming applet, in which a small string is
printed in red type, set in 36 point Times bold.
<BR CLEAR=ALL>
<P>In the next part of the page, we demonstrate how
under certain conditions, styrofoam peanuts can be
used as a healthy snack.
```

The result in a typical Java browser might look like that in Figure 8.5.

Figure 8.5.
*Vertical and horizontal
space.*



8

CODE and CODEBASE

CODE is used to indicate the name of the class file that holds the current applet. If CODE is used alone in the <APPLET> tag, the class file is searched for in the same directory as the HTML file that references it.

If you want to store your class files in a different directory than that of your HTML files, you have to tell the Java-capable browser where to find those class files. To do this, you use CODEBASE. CODE contains only the name of the class file; CODEBASE contains an alternate pathname where classes are contained. For example, if you store your class files in a directory called /classes, which is in the same directory as your HTML files, CODEBASE is the following:

```
<APPLET CODE="myclass.class" CODEBASE="/classes"
  WIDTH=100 HEIGHT=100></APPLET>
```

Passing Parameters to Applets

With Java applications, you can pass parameters to your main() routine by using arguments on the command line. You can then parse those arguments inside the body of your class, and the application acts accordingly based on the arguments it is given.

Applets, however, don't have a command line. How do you pass in different arguments to an applet? Applets can get different input from the HTML file that contains the <APPLET> tag through the use of applet parameters. To set up and handle parameters in an applet, you need two things:

- ☐ A special parameter tag in the HTML file
- ☐ Code in your applet to parse those parameters

Applet parameters come in two parts: a name, which is simply a name you pick, and a value, which determines the value of that particular parameter. So, for example, you can indicate the color of text in an applet by using a parameter with the name `color` and the value `red`. You can determine an animation's speed using a parameter with the name `speed` and the value `5`.

In the HTML file that contains the embedded applet, you indicate each parameter using the `<PARAM>` tag, which has two attributes for the name and the value, called (surprisingly enough), `NAME` and `VALUE`. The `<PARAM>` tag goes inside the opening and closing `<APPLET>` tags:

```
<APPLET CODE="MyApplet.class" WIDTH=100 HEIGHT=100>
<PARAM NAME=font VALUE="TimesRoman">
<PARAM NAME=size VALUE="36">
A Java applet appears here.</APPLET>
```

This particular example defines two parameters to the `MyApplet` applet: one whose name is `font` and whose value is `TimesRoman`, and one whose name is `size` and whose value is `36`.

Parameters are passed to your applet when it is loaded. In the `init()` method for your applet, you can then get hold of those parameters by using the `getParameter()` method. `getParameter()` takes one argument—a string representing the name of the parameter you're looking for—and returns a string containing the corresponding value of that parameter. (Like arguments in Java applications, all the parameter values are strings.) To get the value of the `font` parameter from the HTML file, you might have a line such as this in your `init()` method:

```
String theFontName = getParameter("font");
```

Note: The names of the parameters as specified in `<PARAM>` and the names of the parameters in `getParameter()` must match identically, including having the same case. In other words, `<PARAM NAME="name">` is different from `<PARAM NAME="Name">`. If your parameters are not being properly passed to your applet, make sure the parameter cases match.

Note that if a parameter you expect has not been specified in the HTML file, `getParameter()` returns `null`. Most often, you will want to test for a `null` parameter and supply a reasonable default:

```
if (theFontName == null)
    theFontName = "Courier"
```

Keep in mind that `getParameter()` returns strings—if you want a parameter to be some other object or type, you have to convert it yourself. To parse the `size` parameter from that same HTML file and assign it to an integer variable called `theSize`, you might use the following lines:

```
int theSize;
String s = getParameter("size");
if (s == null)
    theSize = 12;
else theSize = Integer.parseInt(s);
```

Get it? Not yet? Let's create an example of an applet that uses this technique. You'll modify the `HelloAgainApplet` so that it says hello to a specific name, for example "Hello Bill" or "Hello Alice". The name is passed into the applet through an HTML parameter.

Let's start by copying the original `HelloAgainApplet` class:

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;

public class MoreHelloApplet extends java.applet.Applet {

    Font f = new Font("TimesRoman", Font.BOLD, 36);

    public void paint(Graphics g) {
        g.setFont(f);
        g.setColor(Color.red);
        g.drawString("Hello Again!", 5, 50);
    }
}
```

The first thing you need to add to this class is a place for the name. Because you'll need that name throughout the applet, let's add an instance variable for the name, just after the variable for the font:

```
String name;
```

To set a value for the name, you have to get the parameter. The best place to handle parameters to an applet is inside an `init()` method. The `init()` method is defined similarly to `paint()` (public, with no arguments, and a return type of `void`). Make sure when you test for a parameter that you test for a value of `null`. The default, in this case, if a name isn't indicated, is to say hello to "Laura":

```
public void init() {
    name = getParameter("name");
    if (name == null)
        name = "Laura";
}
```

One last thing to do now that you have the name from the HTML parameters is to modify the name so that it's a complete string—that is, to tack "Hello " onto the beginning, and an exclamation point onto the end. You could do this in the `paint` method just before printing the string to the screen. Here it's done only once, however, whereas in `paint()` it's done every time the screen is repainted—in other words, it's slightly more efficient to do it inside `init()` instead:

```
name = "Hello " + name + "!";
```

And now, all that's left is to modify the `paint()` method. The original `drawString()` method looked like this:

```
g.drawString("Hello Again!", 5, 50);
```

To draw the new string you have stored in the `name` instance variable, all you need to do is substitute that variable for the literal string:

```
g.drawString(name, 5, 50);
```

Listing 8.3 shows the final result of the `MoreHelloApplet` class. Compile it so that you have a class file ready.

Type

Listing 8.3. The `MoreHelloApplet` class.

```
1: import java.awt.Graphics;
2: import java.awt.Font;
3: import java.awt.Color;
4:
5: public class MoreHelloApplet extends java.applet.Applet {
6:
7:     Font f = new Font("TimesRoman", Font.BOLD, 36);
8:     String name;
9:
10:    public void init() {
11:        name = getParameter("name");
12:        if (name == null)
13:            name = "Laura";
14:
15:        name = "Hello " + name + "!";
16:    }
17:
18:    public void paint(Graphics g) {
19:        g.setFont(f);
20:        g.setColor(Color.red);
21:        g.drawString(name, 5, 50);
22:    }
23: }
```

Now, let's create the HTML file that contains this applet. Listing 8.4 shows a new Web page for the `MoreHelloApplet` applet.

Type

Listing 8.4. The HTML file for the `MoreHelloApplet` applet.

```
1: <HTML>
2: <HEAD>
3: <TITLE>Hello!</TITLE>
4: </HEAD>
5: <BODY>
```

```

6: <P>
7: <APPLET CODE="MoreHelloApplet.class" WIDTH=300 HEIGHT=50>
8: <PARAM NAME=name VALUE="Bonzo">
9: Hello to whoever you are!
10: </APPLET>
11: </BODY>
12: </HTML>

```



Note the <APPLET> tag, which points to the class file for the applet with the appropriate width and height (300 and 50). Just below it (line 8) is the <PARAM> tag, which you use to pass in the name. Here, the NAME parameter is simply name, and the VALUE is the string "Bonzo".

Loading up this HTML file produces the result shown in Figure 8.6.

Figure 8.6.
*The result of
MoreHelloApplet, first try.*



Let's try a second example. Remember that in the code for MoreHelloApplet, if no name is specified, the default is the name "Laura". Listing 8.5 creates an HTML file with no parameter tag for name.



Listing 8.5. Another HTML File for the MoreHelloApplet applet.

```

1: <HTML>
2: <HEAD>
3: <TITLE>Hello!</TITLE>
4: </HEAD>
5: <BODY>
6: <P>
7: <APPLET CODE="MoreHelloApplet.class" WIDTH=300 HEIGHT=50>
8: Hello to whoever you are!
9: </APPLET>
10: </BODY>
11: </HTML>

```

Here, because no name was supplied, the applet uses the default, and the result is what you might expect (see Figure 8.7).

Figure 8.7.
*The result of
MoreHelloApplet,
second try.*

Hello Laura!

Summary

Applets are probably the most common use of the Java language today. Applets are more complicated than many Java applications because they are executed and drawn inline within Web pages, but they can access the graphics, user interface, and event structure provided by the Web browser itself. Today, you learned the basics of creating applets, including the following things:

- ☐ All applets you develop using Java inherit from the `Applet` class, part of the `java.applet` package. The `Applet` class provides basic behavior for how the applet will be integrated with and react to the browser and various forms of input from that browser and the person running it. By subclassing `Applet`, you have access to all that behavior.
- ☐ Applets have five main methods, which are used for the basic activities an applet performs during its life cycle: `init()`, `start()`, `stop()`, `destroy()`, and `paint()`. Although you don't need to override all these methods, these are the most common methods you'll see repeated in many of the applets you'll create in this book and in other sample programs.
- ☐ To run a compiled applet class file, you include it in an HTML Web page by using the `<APPLET>` tag. When a Java-capable browser comes across `<APPLET>`, it loads and runs the applet described in that tag. Note that to publish Java applets on the World Wide Web alongside HTML files you do not need special server software; any plain old Web server will do just fine.
- ☐ Unlike applications, applets do not have a common line on which to pass arguments, so those arguments must be passed into the applet through the HTML file that contains it. You indicate parameters in an HTML file by using the `<PARAM>` tag inside the opening and closing `<APPLET>` tags. `<PARAM>` has two attributes: `NAME` for the name of the parameter, and `VALUE` for its value. Inside the body of your applet (usually in `init()`), you can then gain access to those parameters using the `getParameter()` method.

Q&A

8

Q In the first part of today's lesson, you say that applets are downloaded from random Web servers and run on the client's system. What's to stop an applet developer from creating an applet that deletes all the files on that system, or in some other way compromises the security of the system?

A Recall that Java applets have several restrictions that make it difficult for all of the more obvious malicious behavior to take place. For example, because Java applets cannot read or write files on the client system, they cannot delete files or read system files that might contain private information. Because they cannot run programs on the client's system without your express permission, they cannot, for example, run system programs pretending to be you. Nor can they run so many programs that your system crashes.

In addition, Java's very architecture makes it difficult to circumvent these restrictions. The language itself, the Java compiler, and the Java interpreter all have checks to make sure that no one has tried to sneak in bogus code or play games with the system itself. You'll learn more about these checks at the end of this book.

Of course, no system can claim to be 100% secure, and the fact that Java applets are run on your system should make you suspicious—see Day 21 for more on security.

Q Wait a minute. If I can't read or write files or run programs on the system the applet is running on, doesn't that mean I basically can't do anything other than simple animations and flashy graphics? How can I save state in an applet? How can I create, say, a word processor or a spreadsheet as a Java applet?

A For everyone who doesn't believe that Java is secure enough, there is someone who believes that Java's security restrictions are too severe for just these reasons. Yes, Java applets are limited because of the security restrictions. But given the possibility for abuse, I believe that it's better to err on the side of being more conservative as far as security is concerned. Consider it a challenge.

Keep in mind, also, that Java applications have none of the restrictions that Java applets do, but because they are also compiled to bytecode, they are portable across platforms. It may be that the thing you want to create would make a much better application than an applet.

Q I have an older version of HotJava. I followed all the examples in this section, but HotJava cannot read my applets (it seems to ignore them). What's going on?

A You most likely have an alpha version of HotJava. Recall that significant changes were made to the Java API and how Java applets are written between alpha and the 1.0 release. The results of these changes are that browsers that support alpha applets cannot read beta applets, and vice versa. The HTML tags are even different, so an older browser just skips over newer applets, and vice versa.

By the time you read this, there may be a new version of HotJava with support for 1.0. If not, you can use Netscape 2.0 or the JDK's applet viewer to view applets written to the beta specification.

Q I noticed in my documentation that the `<APPLET>` tag also has a `NAME` attribute. You didn't discuss it here.

A `NAME` is used when you have multiple applets on a page that need to communicate with each other. You'll learn about this on Day 12.

Q I have an applet that takes parameters and an HTML file that passes it those parameters. But when my applet runs, all I get are null values. What's going on here?

A Do the names of your parameters (in the `NAME` attribute) match exactly with the names you're testing for in `getParameter()`? They must be exact, including case, for the match to be made. Make sure also that your `<PARAM>` tags are inside the opening and closing `<APPLET>` tags, and that you haven't misspelled anything.



10

WEEK
2

Simple Animation and Threads

by Laura Lemay

The first thing I ever saw Java do was an animation: a large red "Hi there!" that ran across the screen from the right to left. Even that simple form of animation was enough to make me stop and think, "this is really cool."

That sort of simple animation takes only a few methods to implement in Java, but those few methods are the basis for any Java applet that you want to update the screen dynamically. Starting with simple animations is a good way to build up to more complicated applets. Today, you'll learn the fundamentals of animation in Java: how the various parts of the system all work together so that you can create moving figures and dynamically updateable applets. Specifically, you'll explore the following:

- ☐ How Java animations work—the `paint()` and `repaint()` methods, starting and stopping dynamic applets, and how to use and override these methods in your own applets
- ☐ Threads—what they are and how they can make your applets more well-behaved with other applets and with the Java system in general
- ☐ Reducing animation flicker, a common problem with animation in Java

Throughout today, you'll also work with lots of examples of real applets that create animations or perform some kind of dynamic movement.

Creating Animation in Java

Animation in Java involves two steps: constructing a frame of animation, and then asking Java to paint that frame. Repeat as necessary to create the illusion of movement. The basic, static applets that you created yesterday taught you how to accomplish the first part; all that's left is how to tell Java to paint a frame.

Painting and Repainting

The `paint()` method, as you learned yesterday, is called by Java whenever the applet needs to be painted—when the applet is initially drawn, when the window containing it is moved, or when another window is moved from over it. You can also, however, ask Java to repaint the applet at a time you choose. So, to change the appearance of what is on the screen, you construct the image or "frame" you want to paint, and then ask Java to paint this frame. If you do this repeatedly, and fast enough, you get animation inside your Java applet. That's all there is to it.

Where does all this take place? Not in the `paint()` method itself. All `paint()` does is put dots on the screen. `paint()`, in other words, is responsible only for the current frame of the animation. The real work of changing what `paint()` does, of modifying the frame for an animation, actually occurs somewhere else in the definition of your applet.

In that “somewhere else,” you construct the frame (set variables for `paint()` to use, create `Color` or `Font` or other objects that `paint()` will need), and then call the `repaint()` method. `repaint()` is the trigger that causes Java to call `paint()` and causes your frame to get drawn.



Technical Note: Because a Java applet can contain many different components that all need to be painted (as you’ll learn later on this week), and in fact, applets are embedded inside a larger Java application that also paints to the screen in similar ways, when you call `repaint()` (and therefore `paint()`) you’re not actually immediately drawing to the screen as you do in other window or graphics toolkits. Instead, `repaint()` is a *request* for Java to repaint your applet as soon as it can. Also, if too many `repaint()` requests are made in a short amount of time, the system may only call `repaint()` once for all of them. Much of the time, the delay between the call and the actual repaint is negligible.

10

Starting and Stopping an Applet’s Execution

Remember `start()` and `stop()` from Day 8? These are the methods that trigger your applet to start and stop running. You didn’t use `start()` and `stop()` yesterday, because the applets on that day did nothing except paint once. With animations and other Java applets that are actually processing and running over time, you’ll need to make use of `start()` and `stop()` to trigger the start of your applet’s execution, and to stop it from running when you leave the page that contains that applet. For many applets, you’ll want to override `start` and `stop` for just this reason.

The `start()` method triggers the execution of the applet. You can either do all the applet’s work inside that method, or you can call other object’s methods in order to do so. Usually, `start()` is used to create and begin execution of a thread so the applet can run in its own time.

`stop()`, on the other hand, suspends an applet’s execution so when you move off the page on which the applet is displaying, it doesn’t keep running and using up system resources. Most of the time when you create a `start()` method, you should also create a corresponding `stop()`.

Putting It Together

Explaining how to do Java animation is more of a task than actually showing you how it works in code. An example or two will help make the relationship between all these methods clearer.

Listing 10.1 shows a sample applet that, at first glance, uses basic applet animation to display the date and time and constantly updates it every second, creating a very simple animated digital clock (a frame from that clock is shown in Figure 10.1).

175

The words “at first glance” in the previous paragraph are very important: this applet doesn't work! However, despite the fact that it doesn't work, you can still learn a lot about basic animation with it, so working through the code will still be valuable. In the next section, you'll learn just what's wrong with it.

See whether you can figure out what's going on with this code before you go on to the analysis.

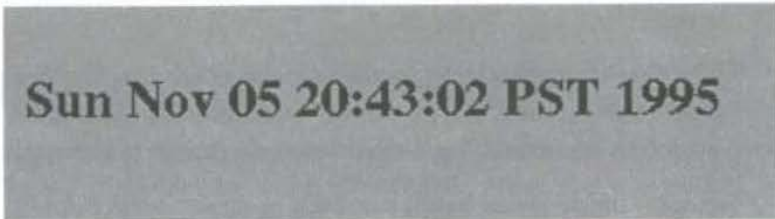
Type

Listing 10.1. The Date applet.

```

1: import java.awt.Graphics;
2: import java.awt.Font;
3: import java.util.Date;
4:
5: public class DigitalClock extends java.applet.Applet {
6:
7:     Font theFont = new Font("TimesRoman", Font.BOLD, 24);
8:     Date theDate;
9:
10:    public void start() {
11:        while (true) {
12:            theDate = new Date();
13:            repaint();
14:            try { Thread.sleep(1000); }
15:            catch (InterruptedException e) { }
16:        }
17:    }
18:
19:    public void paint(Graphics g) {
20:        g.setFont(theFont);
21:        g.drawString(theDate.toString(), 10, 50);
22:    }
23: }
```

Figure 10.1.
The digital clock.



Sun Nov 05 20:43:02 PST 1995

Analysis

Think you've got the basic idea? Let's go through it, line by line.

Lines 7 and 8 define two basic instance variables: `theFont` and `theDate`, which hold objects representing the current font and the current date, respectively. More about these later.

The `start()` method triggers the actual execution of the applet. Note the `while` loop inside this method; given that the test (`true`) always returns `true`, the loop never exits. A single animation frame is constructed inside that `while` loop, with the following steps:

- ☐ The `Date` class represents a date and time (`Date` is part of the `java.util` package—note that it was specifically imported in line three). Line 12 creates a new instance of the `Date` class, which holds the current date and time, and assigns it to the `theDate` instance variable.
- ☐ The `repaint()` method is called.
- ☐ Lines 14 and 15, as complicated as they look, do nothing except pause for 1000 milliseconds (one second) before the loop repeats. The `sleep()` method there, part of the `Thread` class, is what causes the applet to pause. Without a specific `sleep()` method, the applet would run as fast as it possibly could, which, for most computer systems, would be too fast for the eye to see. Using `sleep()` enables you to control exactly how fast the animation takes place. The `try` and `catch` stuff around it enables Java to manage errors if they occur. `try` and `catch` handle exceptions and are described on Day 17, next week.

On to the `paint()` method. Here, inside `paint()`, all that happens is that the current font (in the variable `theFont`) is set, and the date itself is printed to the screen (note that you have to call the `toString()` method to convert the date to a string). Because `paint()` is called repeatedly with whatever value happens to be in `theDate`, the string is updated every second to reflect the new date.

There are a few things to note about this example. First, you might think it would be easier to create the new `Date` object inside the `paint()` method. That way you could use a local variable and not need an instance variable to pass the `Date` object around. Although doing things that way creates cleaner code, it also results in a less efficient program. The `paint()` method is called every time a frame needs to be changed. In this case, it's not that important, but in an animation that needs to change frames very quickly, the `paint()` method has to pause to create that new object every time. By leaving `paint()` to do what it does best—painting the screen—and calculating new objects before hand, you can make painting as efficient as possible. This is precisely the same reason why the `Font` object is also in an instance variable.

Threads: What They Are and Why You Need Them

Depending on your experience with operating systems and with environments within those systems, you may or may not have run into the concept of threads. Let's start from the beginning with some definitions.

When a program runs, it starts executing, runs its initialization code, calls methods or procedures, and continues running and processing until it's complete or until the program is exited. That program uses a single thread—where the thread is a single locus of control for the program.

Multithreading, as in Java, enables several different execution threads to run at the same time inside the same program, in parallel, without interfering with each other.

Here's a simple example. Suppose you have a long computation near the start of a program's execution. This long computation may not be needed until later on in the program's execution—it's actually tangential to the main point of the program, but it needs to get done eventually. In a single-threaded program, you have to wait for that computation to finish before the rest of the program can continue running. In a multithreaded system, you can put that computation into its own thread, enabling the rest of the program to continue running independently.

Using threads in Java, you can create an applet so that it runs in its own thread, and it will happily run all by itself without interfering with any other part of the system. Using threads, you can have lots of applets running at once on the same page. Depending on how many you have, you may eventually tax the system so that all of them will run slower, but all of them will still run independently.

Even if you don't have lots of applets, using threads in your applets is good Java programming practice. The general rule of thumb for well-behaved applets: whenever you have any bit of processing that is likely to continue for a long time (such as an animation loop, or a bit of code that takes a long time to execute), put it in a thread.

The Problem with the Digital Clock Applet

That Digital Clock applet in the last section doesn't use threads. Instead, you put the `while` loop that cycles through the animation directly into the `start()` method so that when the applet starts running it keeps going until you quit the browser or applet viewer. Although this may seem like a good way to approach the problem, the digital clock won't work because the `while` loop in the `start()` method is monopolizing all the resources in the system—including painting. If you try compiling and running the digital clock applet, all you get is a blank screen. You also won't be able to stop the applet normally, because there's no way a `stop()` method can ever be called.

The solution to this problem is to rewrite the applet to use threads. Threads enable this applet to animate on its own without interfering with other system operations, enable it to be started and stopped, and enable you to run it in parallel with other applets.

Writing Applets with Threads

How do you create an applet that uses threads? There are several things you need to do. Fortunately, none of them are difficult, and a lot of the basics of using threads in applets is just boilerplate code that you can copy and paste from one applet to another. Because it's so easy, there's almost no reason *not* to use threads in your applets, given the benefits.

There are four modifications you need to make to create an applet that uses threads:

- ☐ Change the signature of your applet class to include the words `implements Runnable`.
- ☐ Include an instance variable to hold this applet's thread.
- ☐ Modify your `start()` method to do nothing but spawn a thread and start it running.
- ☐ Create a `run()` method that contains the actual code that starts your applet running.

The first change is to the first line of your class definition. You've already got something like this:

```
public class MyAppletClass extends java.applet.Applet {
    ...
}
```

You need to change it to the following:

```
public class MyAppletClass extends java.applet.Applet implements Runnable {
    ...
}
```

What does this do? It includes support for the `Runnable` interface in your applet. If you think way back to Day 2, you'll remember that interfaces are a way to collect method names common to different classes, which can then be mixed in and implemented inside different classes that need to implement that behavior. Here, the `Runnable` interface defines the behavior your applet needs to run a thread; in particular, it gives you a default definition for the `run()` method. By implementing `Runnable` you tell others that they can call the `run()` method on your instances.

The second step is to add an instance variable to hold this applet's thread. Call it anything you like; it's a variable of the type `Thread` (`Thread` is a class in `java.lang`, so you don't have to import it):

```
Thread runner;
```

Third, add a `start()` method or modify the existing one so that it does nothing but create a new thread and start it running. Here's a typical example of a `start()` method:

```
public void start() {
    if (runner == null); {
        runner = new Thread(this);
        runner.start();
    }
}
```



If you modify `start()` to do nothing but spawn a thread, where does the body of your applet go? It goes into a new method, `run()`, which looks like this:

```
public void run() {
    // what your applet actually does
}
```

`run()` can contain anything you want to run in the separate thread: initialization code, the actual loop for your applet, or anything else that needs to run in its own thread. You also can create new objects and call methods from inside `run()`, and they'll also run inside that thread. The `run` method is the real heart of your applet.

Finally, now that you've got threads running and a `start` method to start them, you should add a `stop()` method to suspend execution of that thread (and therefore whatever the applet is doing at the time) when the reader leaves the page. `stop()`, like `start()`, is usually something along these lines:

```
public void stop() {
    if (runner != null) {
        runner.stop();
        runner = null;
    }
}
```

The `stop()` method here does two things: it stops the thread from executing and also sets the thread's variable `runner` to `null`. Setting the variable to `null` makes the `Thread` object it previously contained available for garbage collection so that the applet can be removed from memory after a certain amount of time. If the reader comes back to this page and this applet, the `start` method creates a new thread and starts up the applet once again.

And that's it! Four basic modifications, and now you have a well-behaved applet that runs in its own thread.

Fixing The Digital Clock

Remember the problems you had with the Digital Clock applet at the beginning of this section? Let's fix them so you can get an idea of how a real applet with threads looks. You'll follow the four steps outlined in the previous section.

First, modify the class definition to include the `Runnable` interface (the class is renamed to `DigitalThreads` instead of `DigitalClock`):

```
public class DigitalThreads extends java.applet.Applet implements Runnable {
    ...
}
```

Second, add an instance variable for the `Thread`:

```
Thread runner;
```

For the third step, swap the way you did things. Because the bulk of the applet is currently in a method called `start()`, but you want it to be in a method called `run()`, rather than do a lot of copying and pasting, just rename the existing `start()` to `run()`:

```
public void run() {
    ...
}
```

Finally, add the boilerplate `start()` and `stop()` methods:

```
public void start() {
    if (runner == null); {
        runner = new Thread(this);
        runner.start();
    }

    public void stop() {
        if (runner != null) {
            runner.stop();
            runner = null;
        }
    }
}
```

10

You're finished! One applet converted to use threads in less than a minute flat. The code for the final applet appears in Listing 10.2.

Type

Listing 10.2. The fixed digital clock applet.

```
1: import java.awt.Graphics;
2: import java.awt.Font;
3: import java.util.Date;
4:
5: public class DigitalThreads extends java.applet.Applet
6:     implements Runnable {
7:
8:     Font theFont = new Font("TimesRoman", Font.BOLD, 24);
9:     Date theDate;
10:    Thread runner;
11:
12:    public void start() {
13:        if (runner == null); {
14:            runner = new Thread(this);
15:            runner.start();
16:        }
17:    }
18:
19:    public void stop() {
20:        if (runner != null) {
21:            runner.stop();
22:            runner = null;
23:        }
24:    }
}
```

continues

Listing 10.2. continued

```

25:
26: public void run() {
27:     while (true) {
28:         theDate = new Date();
29:         repaint();
30:         try { Thread.sleep(1000); }
31:         catch (InterruptedException e) { }
32:     }
33: }
34:
35: public void paint(Graphics g) {
36:     g.setFont(theFont);
37:     g.drawString(theDate.toString(), 10, 50);
38: }
39: }
40:

```

Reducing Animation Flicker

If you've been following along with this book and trying the examples as you go, rather than reading this book on an airplane or in the bathtub, you may have noticed that when the date program runs every once in a while, there's an annoying flicker in the animation. (Not that there's anything wrong with reading this book in the bathtub, but you won't see the flicker if you do that, so just trust me—there's a flicker.) This isn't a mistake or an error in the program; in fact, that flicker is a side effect of creating animations in Java. Because it is really annoying, however, you'll learn how to reduce flicker in this part of today's lesson so that your animations run cleaner and look better on the screen.

Flicker and How to Avoid It

Flicker is caused by the way Java paints and repaints each frame of an applet. At the beginning of today's lesson, you learned that when you call the `repaint()` method, `repaint()` calls `paint()`. That's not precisely true. A call to `paint()` does indeed occur in response to a `repaint()`, but what actually happens are the following steps:

1. The call to `repaint()` results in a call to the method `update()`.
2. The `update()` method clears the screen of any existing contents (in essence, fills it with the current background color), and then calls `paint()`.
3. The `paint()` method then draws the contents of the current frame.

It's Step 2, the call to `update()`, that causes animation flicker. Because the screen is cleared between frames, the parts of the screen that don't change alternate rapidly between being painted and being cleared. Hence, flickering.

There are two major ways to avoid flicker in your Java applets:

- ☐ Override `update()` either not to clear the screen at all, or to clear only the parts of the screen you've changed.
- ☐ Override both `update()` and `paint()`, and use double-buffering.

If the second way sounds complicated, that's because it is. Double-buffering involves drawing to an offscreen graphics surface and then copying that entire surface to the screen. Because it's more complicated, you'll explore that one tomorrow. Today, let's cover the easier solution: overriding `update()`.

How to Override Update

The cause of flickering lies in the `update()` method. To reduce flickering, therefore, override `update()`. Here's what the default version of `update()` does (in the `Component` class, which you'll learn more about on Day 13):

```
public void update(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0, 0, width, height);
    g.setColor(getForeground());
    paint(g);
}
```

Basically, `update()` clears the screen (or, to be exact, fills the applet's bounding rectangle with the background color), sets things back to normal, and then calls `paint()`. When you override `update()`, you have to keep these two things in mind and make sure that your version of `update()` does something similar. In the next two sections, you'll work through some examples of overriding `update()` in different cases to reduce flicker.

Solution One: Don't Clear the Screen

The first solution to reducing flicker is not to clear the screen at all. This works only for some applets, of course. Here's an example of an applet of this type. The `ColorSwirl` applet prints a single string to the screen ("All the swirly colors"), but that string is presented in different colors that fade into each other dynamically. This applet flickers terribly when it's run. Listing 10.3 shows the source for this applet, and Figure 10.2 shows the result.

Type

Listing 10.3. The `ColorSwirl` applet.

```
1: import java.awt.Graphics;
2: import java.awt.Color;
3: import java.awt.Font;
4:
```

continues

Listing 10.3. continued

```

5: public class ColorSwirl extends java.applet.Applet
6:     implements Runnable {
7:
8:     Font f = new Font("TimesRoman",Font.BOLD,48);
9:     Color colors[] = new Color[50];
10:    Thread runThread;
11:
12:    public void start() {
13:        if (runThread == null) {
14:            runThread = new Thread(this);
15:            runThread.start();
16:        }
17:    }
18:
19:    public void stop() {
20:        if (runThread != null) {
21:            runThread.stop();
22:            runThread = null;
23:        }
24:    }
25:
26:    public void run() {
27:
28:        // initialize the color array
29:        float c = 0;
30:        for (int i = 0; i < colors.length; i++) {
31:            colors[i] =
32:                Color.getHSBColor(c, (float)1.0,(float)1.0);
33:            c += .02;
34:        }
35:
36:        // cycle through the colors
37:        int i = 0;
38:        while (true) {
39:            setForeground(colors[i]);
40:            repaint();
41:            i++;
42:            try { Thread.sleep(50); }
43:            catch (InterruptedException e) { }
44:            if (i == colors.length ) i = 0;
45:        }
46:    }
47:
48:    public void paint(Graphics g) {
49:        g.setFont(f);
50:        g.drawString("All the Swirly Colors", 15, 50);
51:    }
52: }

```

Figure 10.2.
The ColorSwirl applet.

All the Swirly Colors

Analysis

There are three new things to note about this applet that might look strange to you:

- ☐ When the applet starts, the first thing you do (in lines 28 through 34) is to create an array of `Color` objects that contains all the colors the text will display. By creating all the colors beforehand, you can then just draw text in that color, one at a time; it's easier to precompute all the colors at once.
- ☐ To create the different colors, a method in the `Color` class called `getHSBColor()` creates a color object based on values for hue, saturation, and brightness, rather than the standard red, green, and blue. By incrementing the hue value and keeping saturation and brightness constant you can create a range of colors without having to know the RGB for each one. If you don't understand this, don't worry about it; it's just an easy way to create the color array.
- ☐ The applet then cycles through the array of colors, setting the foreground to each one in turn and calling `repaint`. When it gets to the end of the array, it starts over again (line 44), so the process repeats over and over ad infinitum.

10

Now that you understand what the applet does, let's fix the flicker. Flicker here results because each time the applet is painted, there's a moment where the screen is cleared. Instead of the text cycling neatly from red to a nice pink to purple, it's going from red to grey, to pink to grey, to purple to grey, and so on—not very nice looking at all.

Because the screen clearing is all that's causing the problem, the solution is easy: override `update()` and remove the part where the screen gets cleared. It doesn't really need to get cleared anyhow, because nothing is changing except the color of the text. With the screen clearing behavior removed from `update()`, all `update` needs to do is call `paint()`. Here's what the `update()` method looks like in this applet:

```
public void update(Graphics g) {
    paint(g);
}
```

With that—with one small three-line addition—no more flicker. Wasn't that easy?

Solution Two: Redraw Only What You Have To

For some applets, it won't be quite that easy. Here's another example. In this applet, called Checkers, a red oval (a checker piece) moves from a black square to a white square, as if on a checkerboard. Listing 10.4 shows the code for this applet, and Figure 10.3 shows the applet itself.

Type

Listing 10.4. The Checkers applet.

```

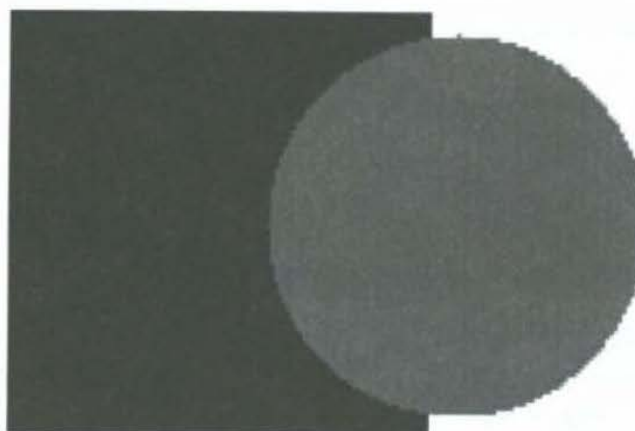
1: import java.awt.Graphics;
2: import java.awt.Color;
3:
4: public class Checkers extends java.applet.Applet
5:     implements Runnable {
6:
7:     Thread runner;
8:     int xpos;
9:
10:    public void start() {
11:        if (runner == null); {
12:            runner = new Thread(this);
13:            runner.start();
14:        }
15:    }
16:
17:    public void stop() {
18:        if (runner != null) {
19:            runner.stop();
20:            runner = null;
21:        }
22:    }
23:
24:    public void run() {
25:        setBackground(Color.blue);
26:        while (true) {
27:            for (xpos = 5; xpos <= 105; xpos+=4) {
28:                repaint();
29:                try { Thread.sleep(100); }
30:                catch (InterruptedException e) { }
31:            }
32:            for (xpos = 105; xpos > 5; xpos -=4) {
33:                repaint();
34:                try { Thread.sleep(100); }
35:                catch (InterruptedException e) { }
36:            }
37:        }
38:    }
39:
40:    public void paint(Graphics g) {
41:        // Draw background
42:        g.setColor(Color.black);

```

```

43:      g.fillRect(0, 0, 100, 100);
44:      g.setColor(Color.white);
45:      g.fillRect(101, 0, 100, 100);
46:
47:      // Draw checker
48:      g.setColor(Color.red);
49:      g.fillOval(xpos, 5, 90, 90);
50:    }
51:  }
  
```

Figure 10.3.
The Checkers applet.



10

Analysis

Here's a quick run-through of what this applet does: an instance variable, `xpos`, keeps track of the current starting position of the checker (because it moves horizontally, the `y` stays constant and the `x` changes). In the `run()` method, you change the value of `x` and repaint, waiting 100 milliseconds between each move. The checker moves from one side of the screen to the other and then moves back (hence the two `for` loops in that method).

In the actual `paint()` method, the background squares are painted (one black and one white), and then the checker is drawn at its current position.

This applet, like the Swirling Colors applet, also has a terrible flicker. (In line 25, the background is blue to emphasize it, so if you run this applet you'll definitely see the flicker.)

However, the solution to solving the flicker problem for this applet is more difficult than for the last one, because you actually want to clear the screen before the next frame is drawn. Otherwise, the red checker won't have the appearance of leaving one position and moving to another; it'll just leave a red smear from one side of the checkerboard to the other.

How do you get around this? You still clear the screen, in order to get the animation effect, but, rather than clearing the entire screen, you clear only the part that you actually changed. By limiting the redraw to only a small area, you can eliminate much of the flicker you get from redrawing the entire screen.

To limit what gets redrawn, you need a couple of things. First, you need a way to restrict the drawing area so that each time `paint()` is called, only the part that needs to get redrawn actually gets redrawn. Fortunately, this is easy by using a mechanism called clipping.

NEW TERM *Clipping*, part of the graphics class, enables you to restrict the drawing area to a small portion of the full screen; although the entire screen may get instructions to redraw, only the portions inside the clipping area are actually drawn.

The second thing you need is a way to keep track of the actual area to redraw. Both the left and right edges of the drawing area change for each frame of the animation (one side to draw the new oval, the other to erase the bit of the oval left over from the previous frame), so to keep track of those two x values, you need instance variables for both the left side and the right.

With those two concepts in mind, let's start modifying the Checkers applet to redraw only what needs to be redrawn. First, you'll add instance variables for the left and right edges of the drawing area. Let's call those instance variables `ux1` and `ux2` (u for update), where `ux1` is the left side of the area to draw and `ux2` the right.

```
int ux1,ux2;
```

Now let's modify the `run()` method so that it keeps track of the actual area to be drawn, which you would think is easy—just update each side for each iteration of the animation. Here, however, things can get complicated because of the way Java uses `paint()` and `repaint()`.

The problem with updating the edges of the drawing area with each frame of the animation is that for every call to `repaint()` there may not be an individual corresponding `paint()`. If system resources get tight (because of other programs running on the system or for any other reason), `paint()` may not get executed immediately and several calls to `paint()` may queue up waiting for their turn to change the pixels on the screen. In this case, rather than trying to make all those calls to `paint()` in order (and be potentially behind all the time), Java catches up by executing only the *most recent* call to `paint()` and skips all the others.

If you update the edges of the drawing area with each `repaint()`, and a couple of calls to `paint()` are skipped, you end up with bits of the drawing surface not being updated and bits of the oval left behind. There's a simple way around this: update the leading edge of the oval each time the frame updates, but only update the trailing edge if the most recent paint has actually occurred. This way, if a couple of calls to `paint()` get skipped, the drawing area will get larger for each frame, and when `paint()` finally gets caught up, everything will get repainted correctly.

Yes, this is horrifyingly complex. If I could have written this applet simpler, I would have, but without this mechanism the applet will not get repainted correctly. Let's step through it slowly in the code so you can get a better grasp of what's going on at each step.

Let's start with `run()`, where each frame of the animation takes place. Here's where you calculate each side of the drawing area based on the old position of the oval and the new position of the



oval. When the oval is moving toward the left side of the screen, this is easy. The value of `ux1` (the left side of the drawing area) is the previous oval's x position (`xpos`), and the value of `ux2` is the x position of the current oval plus the width of that oval (90 pixels in this example).

Here's what the old `run()` method looked like, to refresh your memory:

```
public void run() {
    setBackground(Color.blue);
    while (true) {
        for (xpos = 5; xpos <= 105; xpos += 4) {
            repaint();
            try { Thread.sleep(100); }
            catch (InterruptedException e) { }
        }
        for (xpos = 105; xpos > 5; xpos -= 4) {
            repaint();
            try { Thread.sleep(100); }
            catch (InterruptedException e) { }
        }
    }
}
```

10

In the first for loop in the `run()` method, where the oval is moving towards the right, you first update `ux2` (the right edge of the drawing area):

```
ux2 = xpos + 90;
```

Then, after the `repaint()` has occurred, you update `ux1` to reflect the old x position of the oval. However, you want to update this value only if the paint actually happened. How can you tell if the paint actually happened? You can reset `ux1` in `paint()` to a given value (0), and then test to see whether you can update that value or whether you have to wait for the `paint()` to occur:

```
if (ux1 == 0) ux1 = xpos;
```

Here's the new, completed for loop for when the oval is moving to the right:

```
for (xpos = 5; xpos <= 105; xpos += 4) {
    ux2 = xpos + 90;
    repaint();
    try { Thread.sleep(100); }
    catch (InterruptedException e) { }
    if (ux1 == 0) ux1 = xpos;
}
```

When the oval is moving to the left, everything flips. `ux1`, the left side, is the leading edge of the oval that gets updated every time, and `ux2`, the right side, has to wait to make sure it gets updated. So, in the second for loop, you first update `ux1` to be the x position of the current oval:

```
ux1 = xpos;
```

Then, after the `repaint()` is called, you test to make sure the paint happened and update `ux2`:

```
if (ux2 == 0) ux2 = xpos + 90;
```

Here's the new version of the second for loop inside `run()`:

```
for (xpos = 105; xpos > 5; xpos -= 4) {
    ux1 = xpos;
    repaint();
    try { Thread.sleep(100); }
    catch (InterruptedException e) { }
    if (ux2 == 0) ux2 = xpos + 90;
}
```

Those are the only modifications `run()` needs. Let's override `update()` to limit the region that is being painted to the left and right edges of the drawing area that you set inside `run()`. To clip the drawing area to a specific rectangle, use the `clipRect()` method. `clipRect()`, like `drawRect()`, `fillRect()`, and `clearRect()`, is defined for graphics objects and takes four arguments: `x` and `y` starting positions, and width and height of the region.

Here's where `ux1` and `ux2` come into play. `ux1` is the `x` point of the top corner of the region; then use `ux2` to get the width of the region by subtracting `ux1` from that value. Finally, to finish `update()`, you call `paint()`:

```
public void update(Graphics g) {
    g.clipRect(ux1, 5, ux2 - ux1, 95);
    paint(g);
}
```

Note that with the clipping region in place, you don't have to do anything to the actual `paint()` method. `paint()` goes ahead and draws to the entire screen each time, but only the areas inside the clipping region actually get changed on screen.

You need to update the trailing edge of each drawing area inside `paint()` in case several calls to `paint()` were skipped. Because you are testing for a value of 0 inside `run()`, you merely reset `ux1` and `ux2` to 0 after drawing everything:

```
ux1 = ux2 = 0;
```

Those are the only changes you have to make to this applet in order to draw only the parts of the applet that changed (and to manage the case where some frames don't get updated immediately). Although this doesn't totally eliminate flickering in the animation, it does reduce it a great deal. Try it and see. Listing 10.5 shows the final code for the Checkers applet.

Type

Listing 10.5. The final Checkers applet.

```
1: import java.awt.Graphics;
2: import java.awt.Color;
3:
4: public class Checkers2 extends java.applet.Applet implements Runnable {
5:
6:     Thread runner;
7:     int xpos;
```

```

8:     int ux1,ux2;
9:
10:    public void start() {
11:        if (runner == null); {
12:            runner = new Thread(this);
13:            runner.start();
14:        }
15:    }
16:
17:    public void stop() {
18:        if (runner != null) {
19:            runner.stop();
20:            runner = null;
21:        }
22:    }
23:
24:    public void run() {
25:        setBackground(Color.blue);
26:        while (true) {
27:            for (xpos = 5; xpos <= 105; xpos += 4) {
28:                ux2 = xpos + 90;
29:                repaint();
30:                try { Thread.sleep(100); }
31:                catch (InterruptedException e) { }
32:                if (ux1 == 0) ux1 = xpos;
33:            }
34:            for (xpos = 105; xpos > 5; xpos -= 4) {
35:                ux1 = xpos;
36:                repaint();
37:                try { Thread.sleep(100); }
38:                catch (InterruptedException e) { }
39:                if (ux2 == 0) ux2 = xpos + 90;
40:            }
41:        }
42:    }
43:    public void update(Graphics g) {
44:        g.clipRect(ux1, 5, ux2 - ux1, 95);
45:        paint(g);
46:    }
47:
48:    public void paint(Graphics g) {
49:        // Draw background
50:        g.setColor(Color.black);
51:        g.fillRect(0, 0, 100, 100);
52:        g.setColor(Color.white);
53:        g.fillRect(101, 0, 100, 100);
54:
55:        // Draw checker
56:        g.setColor(Color.red);
57:        g.fillOval(xpos, 5, 90, 90);
58:
59:        // reset the drawing area
60:        ux1 = ux2 = 0;
61:    }
62: }

```

Summary

Congratulations on getting through Day 10! This day was a bit rough; you've learned a lot, and it all might seem overwhelming. You learned about a plethora of methods to use and override: `start()`, `stop()`, `paint()`, `repaint()`, `run()`, and `update()`—and you got a solid foundation in creating and using threads.

After today, you're over the worst hurdles in terms of understanding applets. Other than handling bitmap images, which you'll learn about tomorrow, you now have the basic background to create just about any animation you want in Java.

Q&A

Q Why all the indirection with `paint()` and `repaint()` and `update()` and all that? Why not have a simple paint method that just puts stuff on the screen when you want it there?

A The Java AWT toolkit enables you to nest drawable surfaces within other drawable surfaces. When a `paint()` takes place, all the parts of the system are redrawn, starting from the outermost surface and moving downward into the most nested one. Because the drawing of your applet takes place at the same time everything else is drawn, your applet doesn't get any special treatment. Your applet will be painted when everything else is painted. Although with this system you sacrifice some of the immediacy of instant painting, it enables your applet to co-exist with the rest of the system more cleanly.

Q Are Java threads like threads on other systems?

A Java threads have been influenced by other thread systems, and if you're used to working with threads, many of the concepts in Java threads will be very familiar to you. You learned the basics today; you'll learn more next week on Day 18.

Q When an applet uses threads, I just have to tell the thread to start and it starts, and tell it to stop and it stops? That's it? I don't have to test anything in my loops or keep track of its state? It just stops?

A It just stops. When you put your applet into a thread, Java can control the execution of your applet much more readily. By causing the thread to stop, your applet just stops running, and then resumes when the thread starts up again. Yes, it's all automatic. Neat, isn't it?



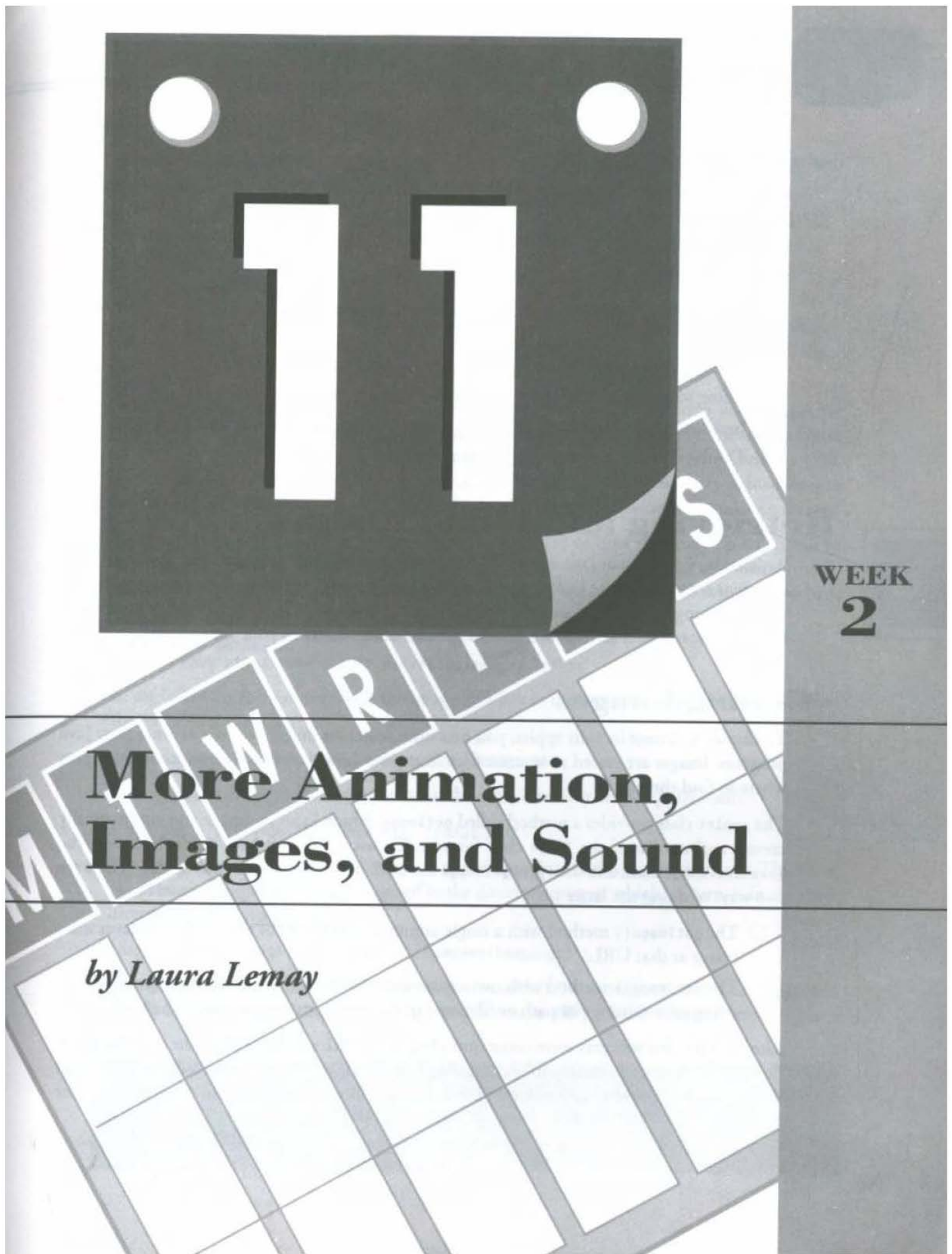
Q The Swirling Colors applet seems to display only five or six colors. What's going on here?

A This is the same problem that you ran into yesterday wherein, on some systems, there might not be enough colors to be able to display all of them reliably. If you're running into this problem, other than upgrading your hardware, you might try quitting other applications running on your system that use color. Other browsers or color tools in particular might be hogging colors that Java wants to be able to use.

Q Even with the changes you made, the Checkers applet still flickers.

A And, unfortunately, it will continue to do so. Reducing the size of the drawing area by using clipping does significantly reduce the flickering, but it doesn't stop it entirely. For many applets, using either of the methods described today may be enough to reduce animation flicker to the point where your applet looks good. To get totally flicker-free animation, you'll need to use a technique called double-buffering, which you'll learn about tomorrow.

10



Animations are fun and easy to do in Java, but there's only so much you can do with the built-in Java methods for lines and fonts and colors. For really interesting animations, you have to provide your own images for each frame of the animation—and having sounds is nice, as well. Today, you'll do more with animations, incorporating images and sounds into Java applets.

Specifically, you'll explore the following topics:

- ☐ Using images—getting them from the server, loading them into Java, and displaying them in your applet
- ☐ Creating animations by using images, including an extensive example
- ☐ Using sounds—getting them and playing them at the appropriate times
- ☐ Sun's Animator applet—an easy way to organize animations and sounds in Java
- ☐ Double-buffering—hardcore flicker avoidance

Retrieving and Using Images

Basic image handling in Java is easy. The `Image` class in `java.awt` provides abstract methods to represent common image behavior, and special methods defined in `Applet` and `Graphics` give you everything you need to load and display images in your applet as easily as drawing a rectangle. In this section, you'll learn about how to get and draw images in your Java applets.

Getting Images

To display an image in your applet, you first must load that image over the net into your Java program. Images are stored as separate files from your Java class files, so you have to tell Java where to find them.

The `Applet` class provides a method called `getImage`, which loads an image and automatically creates an instance of the `Image` class for you. To use it, all you have to do is import the `java.awt.Image` class, and then give `getImage` the URL of the image you want to load. There are two ways of doing the latter step:

- ☐ The `getImage()` method with a single argument (an object of type `URL`) retrieves the image at that URL.
- ☐ The `getImage()` method with two arguments: the base URL (also a `URL` object) and a string representing the path or filename of the actual image (relative to the base).

Although the first way may seem easier (just plug in the URL as a `URL` object), the second is more flexible. Remember, because you're compiling Java files, if you include a hard-coded URL of an image and then move your files around to a different location, you have to recompile all your Java files.

The latter form, therefore, is usually the one to use. The `Applet` class also provides two methods that will help with the base URL argument to `getImage()`:

- ☐ The `getDocumentBase()` method returns a `URL` object representing the directory of the HTML file that contains this applet. So, for example, if the HTML file is located at `http://www.myserver.com/htmlfiles/javahtml/`, `getDocumentBase` returns a `URL` pointing to that path.
- ☐ The `getCodeBase()` method returns a string representing the directory in which this applet is contained—which may or may not be the same directory as the HTML file, depending on whether the `CODEBASE` attribute in `<APPLET>` is set or not.

Whether you use `getDocumentBase()` or `getCodeBase()` depends on whether your images are relative to your HTML files or relative to your Java class files. Use whichever one applies better to your situation. Note that either of these methods is more flexible than hard-coding a URL or pathname into the `getImage()` method; using either `getDocumentBase()` or `getCodeBase()` enables you to move your HTML files and applets around and Java can still find your images.

Here are a few examples of `getImage`, to give you an idea of how to use it. This first call to `getImage()` retrieves the file at that specific URL ("`http://www.server.com/files/image.gif`"). If any part of that URL changes, you have to recompile your Java applet to take the new path into account:

```
Image img = getImage(
    new URL("http://www.server.com/files/image.gif"));
```

In the following form of `getImage`, the `image.gif` file is in the same directory as the HTML files that refer to this applet:

```
Image img = getImage(getDocumentBase(), "image.gif")
```

In this similar form, the file `image.gif` is in the same directory as the applet itself:

```
Image img = getImage(getCodeBase(), "image.gif")
```

If you have lots of image files, it's common to put them into their own subdirectory. This form of `getImage()` looks for the file `image.gif` in the directory `images`, which, in turn, is in the same directory as the Java applet:

```
Image img = getImage(getCodeBase(), "images/image.gif")
```

If `getImage()` can't find the file indicated, it returns `null`. `drawImage()` on a `null` image will simply draw nothing. Using a `null` image in other ways will probably cause an error.



Note: Currently, Java supports images in the GIF and JPEG formats. Other image formats may be available later; however, for now, your images should be in either GIF or JPEG.

Drawing Images

All that stuff with `getImage()` does nothing except go off and retrieve an image and stuff it into an instance of the `Image` class. Now that you have an image, you have to do something with it.

The most likely thing you're going to want to do is display it as you would a rectangle or a text string. The `Graphics` class provides two methods to do just this, all called `drawImage()`.

The first version of `drawImage()` takes four arguments: the image to display, the `x` and `y` positions of the top left corner, and `this`:

```
public void paint() {
    g.drawImage(img, 10, 10, this);
}
```

This first form does what you would expect it to: it draws the image in its original dimensions with the top left corner at the given `x` and `y` positions. Listing 11.1 shows the code for a very simple applet that loads in an image called `ladybug.gif` and displays it. Figure 11.1 shows the obvious result.

Type

Listing 11.1. The Ladybug applet.

```
1: import java.awt.Graphics;
2: import java.awt.Image;
3:
4: public class LadyBug extends java.applet.Applet {
5:
6:     Image bugimg;
7:
8:     public void init() {
9:         bugimg = getImage(getCodeBase(),
10:            "images/ladybug.gif");
11:     }
12:
13:     public void paint(Graphics g) {
14:         g.drawImage(bugimg, 10, 10, this);
15:     }
16: }
```

Analysis

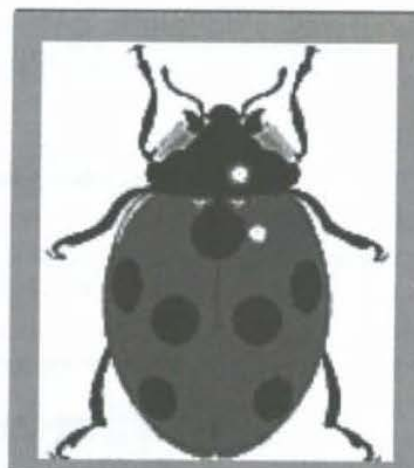
In this example we have an instance variable `bugimg` to hold the ladybug image, which is loaded in the `init()` method. The `paint()` method then draws that image on the screen.

The second form of `drawImage` takes six arguments: the image to draw, the `x` and `y` coordinates, a width and height of the image bounding box, and `this`. If the width and height arguments for the bounding box are smaller or larger than the actual image, the image is automatically scaled to fit. Using those extra arguments enables you to squeeze and expand images into whatever space you need them to fit in (keep in mind, however, that there may be some image degradation from scaling it smaller or larger than its intended size).

One helpful hint for scaling images is to find out the size of the actual image that you've loaded, so you can then scale it to a specific percentage and avoid distortion in either direction. Two

methods defined for the `Image` class enable you to do this: `getWidth()` and `getHeight()`. Both take a single argument, an instance of `ImageObserver`, which is used to track the loading of the image (more about this later). Most of the time, you can use just this as an argument to either `getWidth()` or `getHeight()`.

Figure 11.1.
The Ladybug image.



If you stored the ladybug image in a variable called `bugimg`, for example, this line returns the width of that image, in pixels:

```
theWidth = bugimg.getWidth(this);
```

Listing 11.2 shows another use of the ladybug image, this time scaled several times to different sizes (Figure 11.2 shows the result).

Type

Listing 11.2. More Ladybugs, scaled.

```
1: import java.awt.Graphics;
2: import java.awt.Image;
3:
4: public class LadyBug2 extends java.applet.Applet {
5:
6:     Image bugimg;
7:
8:     public void init() {
9:         bugimg = getImage(getCodeBase(),
10:            "images/ladybug.gif");
11:     }
12:
13:     public void paint(Graphics g) {
14:         int iwidth = bugimg.getWidth(this);
15:         int iheight = bugimg.getHeight(this);
16:         int xpos = 10;
17:
18:         // 25 %
```

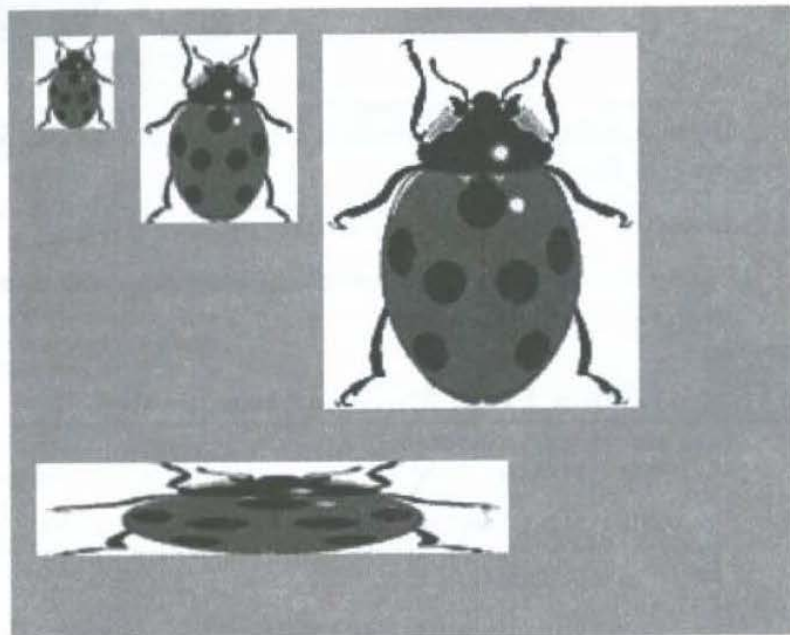
continues

Listing 11.2. continued

```

19:      g.drawImage(bugimg, xpos, 10,
20:                  iwidth / 4, iheight / 4, this);
21:
22:      // 50 %
23:      xpos += (iwidth / 4) + 10;
24:      g.drawImage(bugimg, xpos, 10,
25:                  iwidth / 2, iheight / 2, this);
26:
27:      // 100%
28:      xpos += (iwidth / 2) + 10;
29:      g.drawImage(bugimg, xpos, 10, this);
30:
31:      // 150% x, 25% y
32:      g.drawImage(bugimg, 10, iheight + 30,
33:                  (int)(iwidth * 1.5), iheight / 4, this);
34:  }
35: }
```

Figure 11.2.
The second Ladybug applet.



Analysis

I've been steadfastly ignoring mentioning that last argument to `drawImage()`: the mysterious `this`, which also appears as an argument to `getWidth()` and `getHeight()`. Why is this argument used? Its official use is to pass in an object that functions as an `ImageObserver` (that is, an object that implements the `ImageObserver` interface). Image observers enable you to watch the progress of how far along an image is in the loading process and to make decisions

when the image is only fully or partially loaded. The `Applet` class, which your applet inherits from, contains default behavior for watching for images that should work in the majority of cases—hence, the `this` argument to `drawImage()`, `getWidth()`, and `getHeight()`. The only reason you'll want to use an alternate argument in its place is if you are tracking lots of images loading asynchronously. See the `java.awt.image.ImageObserver` class for more details.

Modifying Images

In addition to the basics of handling images described in this section, the `java.awt.image` package provides more classes and interfaces that enable you to modify images and their internal colors, or to create bitmap images by hand. Most of these classes require background knowledge in image processing, including a good grasp of color models and bitwise operations. All these things are outside the scope of an introductory book on Java, but if you have this background (or you're interested in trying it out), the classes in `java.awt.image` will be helpful to you. Take a look at the example code for creating and using images that comes with the Java development kit for examples of how to use the image classes.

Creating Animation Using Images

Creating animations by using images is much the same as creating images by using fonts, colors, or shapes—you use the same methods, the same procedures for painting, repainting, and reducing flicker that you learned about yesterday. The only difference is that you have a stack of images to flip through rather than a set of painting methods.

Probably the best way to show you how to use images for animation is simply to walk through an example. Here's an extensive one of an animation of a small cat called Neko.

An Example: Neko

Neko was a small Macintosh animation/game written and drawn by Kenji Gotoh in 1989. "Neko" is Japanese for "cat," and the animation is of a small kitten that chases the mouse pointer around the screen, sleeps, scratches, and generally acts cute. The Neko program has since been ported to just about every possible platform, as well as rewritten as a popular screensaver.

For this example, you'll implement a small animation based on the original Neko graphics. Unlike the original Neko the cat, which was autonomous (it could "sense" the edges of the window and turn and run in a different direction), this applet merely causes Neko to run in from the left side of the screen, stop in the middle, yawn, scratch its ear, sleep a little, and then run off to the right.

Note: This is by far the largest of the applets discussed in this book, and if I either print it here and then describe it, or build it up line by line, you'll be here for days. Instead, I'm going to describe the parts of this applet independently, and I'm going to leave out the basics—the stuff you learned yesterday about starting and stopping threads, what the `run()` method does, and so on. All the code is printed later today so that you can put it all together.

Before you begin writing Java code to construct an animation, you should have all the images that form the animation itself. For this version of Neko there are nine of them (the original has 36), as shown in Figure 11.3.

Figure 11.3.
The images for Neko.



I've stored these images in a directory called, appropriately, `images`. Where you store your images isn't all that important, but you should take note of where you've put them because you'll need that information.

Now, onto the applet. The basic idea of animation by using images is that you have a set of images, and you display them one at a time, rapidly, so they give the appearance of movement. The easiest way to manage this in Java is to store the images in an array of class `Image`, and then to have a special variable that stores a reference to the current image.

Technical Note: The `java.util` class contains a class (`HashTable`) that implements a hash table. For large numbers of images, a hash table is faster to find and retrieve images from than an array is. Because there's small number of images here, and because arrays are better for fixed-length, repeating animations, I'll use an array here.

For the Neko applet, you'll include instance variables to implement both these things: an array to hold the images called `nekopics`, and a variable of type `Image` to hold the current image:

```
Image nekopics[] = new Image[9];
Image currentimg;
```

Because you'll need to pass the position of the current image around between the methods in this applet, you'll also need to keep track of the current `x` and `y` positions. The `y` stays constant for this particular applet, but the `x` may vary. let's add two instance variables for those two positions:

```
int xpos;
int ypos = 50;
```

Now, onto the body of the applet. During the applet's initialization, you'll read in all the images and store them in the `nekopics` array. This is the sort of operation that works especially well in an `init()` method.

Given that you have nine images with nine different filenames, you could do a separate call to `getImage` for each one. You can save at least a little typing, however, by creating an array of the file names (`nekosrc`, an array of strings) and then just using a `for` loop to iterate over each one. Here's the `init()` method for the Neko applet that loads all the images into the `nekopics` array:

```
public void init() {
    String nekosrc[] = { "right1.gif", "right2.gif",
        "stop.gif", "yawn.gif", "scratch1.gif",
        "scratch2.gif", "sleep1.gif", "sleep2.gif",
        "awake.gif" };
    for (int i=0; i < nekopics.length; i++) {
        nekopics[i] = getImage(getCodeBase(),
            "images/" + nekosrc[i]);
    }
}
```

Note here in the call to `getImage()` that the directory these images are stored in is included as part of the path.

With the images loaded, the next step is to start animating the bits of the applet. You do this inside the applet's thread's `run()` method. In this applet, Neko does five main things:

- ☐ Runs in from the left side of the screen
- ☐ Stops in the middle and yawns
- ☐ Scratches four times
- ☐ Sleeps
- ☐ Wakes up and runs off to the right side of the screen

Because you could animate this applet by merely painting the right image to the screen at the right time, it makes more sense to write this applet so that many of Neko's activities are contained in individual methods. This way, you can reuse some of the activities (the animation of Neko running, in particular) if you want Neko to do things in a different order.

Let's start by creating a method to make Neko run. Because you're going to be using this one twice, making it generic is a good plan. Let's create the `nekorun()` method, which takes two arguments: the `x` position to start, and the `x` position to end. Neko then runs between those two positions (the `y` remains constant).

There are two images that represent Neko running; so, to create the running effect, you need to alternate between those two images (stored in positions 0 and 1 of the image array), as well