

Teach Yourself **JAVA** in 21 Days

Laura Lemay
Charles L. Perkins



201 West 103rd Street
Indianapolis, Indiana 46290

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.

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



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



An Output icon highlights what the same HTML code looks like when viewed by either Netscape or Mosaic.



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

*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 and its licensors

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 2 1

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 Wentz

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 The Java Class Library	483
	C How Java Differs from C and C++	497
	D How Java Differs from C and C++	507
	Index	511

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

	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 Libraries	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

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 SimpleFileNative.c	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

Acknowledgments

From Laura Lemay:

To Sun's Java team, for all their hard work on Java the language and on the browser, and particularly to Jim Graham, who demonstrated Java and HotJava to me on very short notice in May and planted the idea for this book.

To everyone who bought my previous books, and liked them. Buy this one too.

From Charles L. Perkins:

To Patrick Naughton, who first showed me the power and the promise of OAK (Java) in early 1993.

To Mark Taber, who shepherded this lost sheep through his first book.

About the Authors

Laura Lemay is a technical writer and a nerd. After spending six years writing software documentation for various computer companies in Silicon Valley, she decided writing books would be much more fun (but has still not yet made up her mind). In her spare time she collects computers, e-mail addresses, interesting hair colors, and nonrunning motorcycles. She is also the perpetrator of *Teach Yourself Web Publishing with HTML in 14 Days*.

You can reach her by e-mail at lemay@lne.com, or visit her home page at <http://www.lne.com/lemay/>.

Charles L. Perkins is the founder of Virtual Rendezvous, a company building what it spent two years designing: a software layer above Java that will foster socially focused, computer-mediated, real-time filtered interactions between people's personas in the virtual environments of the near future. In previous lives, he has evangelized NeXTSTEP, Smalltalk, and UNIX, and has degrees in both physics and computer science. Before attempting this book, he was an amateur columnist and author. He's done research in speech recognition, neural nets, gestural user interfaces, computer graphics, and language theory, but had the most fun working at Thinking Machines and Xerox PARC's Smalltalk group. In his spare time, he reads textbooks for fun.

You can reach him via e-mail at virtual@rendezvous.com, or visit his Java page at <http://rendezvous.com/java>.

Introduction

The World Wide Web, for much of its existence, has been a method for distributing passive information to a widely distributed number of people. The Web has, indeed, been exceptionally good for that purpose. With the addition of forms and image maps, Web pages began to become interactive—but the interaction was often simply a new way to get at the same information. The limitations of Web distribution were all too apparent once designers began to try to stretch the boundaries of what the Web can do. Even other innovations, such as Netscape's server push to create dynamic animations, were merely clever tricks layered on top of a framework that wasn't built to support much other than static documents with images and text.

Enter Java, and the capability for Web pages of containing Java applets. Applets are small programs that create animations, multimedia presentations, real-time (video) games, multi-user networked games, and real interactivity—in fact, most anything a small program can do, Java applets can. Downloaded over the net and executed inside a Web page by a browser that supports Java, applets are an enormous step beyond standard Web design.

The disadvantage of Java is that to create Java applets right now, you need to write them in the Java language. Java is a programming language, and as such, creating Java applets is more difficult than creating a Web page or a form using HTML. Soon there will be tools and programs that will make creating Java applets easier—they may be available by the time you read this. For now, however, the only way to delve into Java is to learn the language and start playing with the raw Java code. Even when the tools come out, you may want to do more with Java than the tools can provide, and you're back to learning the language.

That's where *Teach Yourself Java in 21 Days* comes in. This book teaches you all about the Java language and how to use it to create not only applets, but also applications, which are more general Java programs that don't need to run inside a Web browser. 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

Teach Yourself Java in 21 Days is intended for people with at least some basic programming background—which includes people with years of programming experience and 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 one or more of the following is true:

- ☐ 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 onto the next level in Web page design.

- ☐ You had some Basic or Pascal in school, you've got 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. *Teach Yourself Java in 21 Days* assumes no background in object-oriented design. If you know object-oriented programming, the first couple of days will be easy for you.

What if you're a rank beginner? This book might move a little fast for you. Java is a good language to start with, though, and if you take it slow and work through all the examples, you may still be able to pick up Java and start creating your own applets.

How This Book Is Organized

Teach Yourself Java in 21 Days describes Java primarily in its current state—what's known as the beta API (Application Programming Interface). This is the version of Java that Netscape and other browsers, such as Spyglass's Mosaic, support. A previous version of Java, the alpha API, was significantly different from the version described in this book, and the two versions are not compatible with each other. There are other books that describe only the alpha API, and there may still be programs and browsers out there that can only run using alpha Java programs.

Teach Yourself Java in 21 Days uses primarily Java beta because that is the version that is most current and is the version that will continue to be used in the future. The alpha API is obsolete and will eventually die out. If you learn Java using beta API, you'll be much better prepared for any future changes (which will be minor) than if you have to worry about both APIs at once.

Java is still in development. "Beta" means that Java is not complete and that things may change between the time this book is being written and the time you read this. Keep this in mind as you work with Java and with the software you'll use to create and compile programs. If things aren't behaving the way you expect, check the Web sites mentioned at the end of this introduction for more information.

Teach Yourself Java in 21 Days covers the Java language and its class libraries in 21 days, organized as three separate weeks. Each week covers a different broad area of developing Java applets and applications.

In the first week, you'll learn about the Java language itself:

- ☐ Day 1 is the basic introduction: what Java is, why it's cool, and how to get the software. You'll also create your first Java applications and applets.

- ☐ On Day 2, you'll explore basic object-oriented programming concepts as they apply to Java.
- ☐ On Day 3, you start getting down to details with the basic Java building blocks: data types, variables, and expressions such as arithmetic and comparisons.
- ☐ Day 4 goes into detail about how to deal with objects in Java: how to create them, how to access their variables and call their methods, and how to compare and copy them. You'll also get your first glance at the Java class libraries.
- ☐ On Day 5, you'll learn more about Java with arrays, conditional statements, and loops.
- ☐ Day 6 is the best one yet. You'll learn how to create classes, the basic building blocks of any Java program, as well as how to put together a Java application (an application being a Java program that can run on its own without a Web browser).
- ☐ Day 7 builds on what you learned on Day 6. On Day 7, you'll learn more about how to create and use methods, including overriding and overloading methods and creating constructors.

Week 2 is dedicated to applets and the Java class libraries:

- ☐ Day 8 provides the basics of applets—how they're different from applications, how to create them, and the most important parts of an applet's life cycle. You'll also learn how to create HTML pages that contain Java applets.
- ☐ On Day 9, you'll learn about the Java classes for drawing shapes and characters to the screen—in black, white, or any other color.
- ☐ On Day 10, you'll start animating those shapes you learned about on Day 9, including learning what threads and their uses are.
- ☐ Day 11 covers more detail about animation, adding bitmap images and audio to the soup.
- ☐ Day 12 delves into interactivity—handling mouse and keyboard clicks from the user in your Java applets.
- ☐ Day 13 is ambitious; on that day you'll learn about using Java's Abstract Windowing Toolkit to create a user interface in your applet including menus, buttons, checkboxes, and other elements.
- ☐ On Day 14, you explore the last of the main Java class libraries for creating applets: windows and dialogs, networking, and a few other tidbits.

Week 3 finishes up with advanced topics, for when you start doing larger and more complex Java programs, or when you want to learn more:

- ☐ On Day 15, you'll learn more about the Java language's modifiers—for abstract and final methods and classes as well as for protecting a class's private information from the prying eyes of other classes.

- ☐ Day 16 covers interfaces and packages, useful for abstracting protocols of methods to aid reuse and for the grouping and categorization of classes.
- ☐ Day 17 covers exceptions: errors and warnings and other abnormal conditions, generated either by the system or by you in your programs.
- ☐ Day 18 builds on the thread basics you learned on Day 10 to give a broad overview of multithreading and how to use it to allow different parts of your Java programs to run in parallel.
- ☐ On Day 19, you'll learn all about the input and output streams in Java's I/O library.
- ☐ Day 20 teaches you about native code—how to link C code into your Java programs to provide missing functionality or to gain performance.
- ☐ Finally, on Day 21, you'll get an overview of some of the “behind-the-scenes” technical details of how Java works: the bytecode compiler and interpreter, the techniques Java uses to ensure the integrity and security of your programs, and the Java garbage collector.

Conventions Used in This Book

Text that you type and text that should appear on your screen is presented in `monospace` type:

It will look like this.

to mimic the way text looks on your screen. Variables and placeholders will appear in *monospace italic*.

The end of each chapter offers common questions asked about that day's subject matter with answers from the authors.

Web Sites for Further Information

Before, while, and after you read this book, there are two Web sites that may be of interest to you as a Java developer.

The official Java web site is at <http://java.sun.com/>. At this site, you'll find the Java development software, the HotJava web browser, and online documentation for all aspects of the Java language. It has several mirror sites that it lists online, and you should probably use the site “closest” to you on the Internet for your downloading and Java Web browsing. There is also a site for developer resources, called Gamelan, at <http://www.gamelan.com/>.

This book also has a companion Web site at <http://www.lne.com/Web/Java/>. Information at that site includes examples, more information and background for this book, corrections to this book, and other tidbits that were not included here.

1

- ☐ 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 modifying instance variables
 - Converting objects
- ☐ Arrays, Conditionals, and Loops
 - Conditional tests
 - Iteration
 - Block statements

WEEK

AT A GLANCE

1

2

3

4

5

6

7



Week 1 at a Glance

- ☐ Creating Classes and Applications in Java
 - Defining constants, instance and class variables, and methods
- ☐ More About Methods
 - Overloading methods
 - Constructor methods
 - Overriding methods



WEEK
1

An Introduction to Java Programming

by Laura Lemay



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-aware 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 won't see anything). 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.

At the time this book is being written, Sun has released the beta version of the Java Developer's Kit (JDK), which includes tools for developing Java applets and applications on Sun systems running Solaris 2.3 or higher for Windows NT and for Windows 95. By the time you read this, support for Java development may have appeared on other platforms, either from Sun or from third-party companies.

Note that because the JDK is currently in beta, it is still subject to change between now and when it is officially released. Applets and applications you write using the JDK and using the examples in this book may require some changes to work with future versions of the JDK. However, because the Java language has been around for several years and has been used for several projects, the language itself is quite stable and robust and most likely will not change excessively. Keep this beta status in mind as you read through this book and as you develop your own Java programs.

Support for playing Java programs is a little more confusing at the moment. Sun's HotJava is not currently included with the Beta JDK; the only available version of HotJava is an older alpha version, and, tragically, applets written for the alpha version of Java do not work with the beta JDK, and vice versa. By the time you read this, Sun may have released a newer version of HotJava which will enable you to view applets.

The 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 `appletviewer` later today.

What's in store for the future? In addition to the final 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.

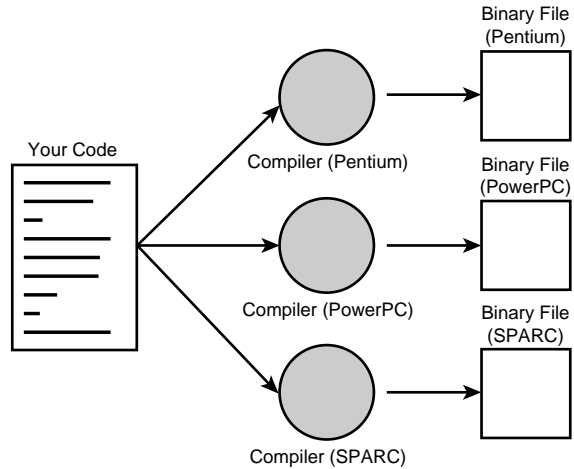
Platform-independence doesn't stop at the source level, however. Java binary files are also platform-independent and can run on multiple problems without the need to recompile the source. How does this work? Java binary files are actually in a form called bytecodes.

NEW TERM *Bytecodes* are a set of instructions that looks a lot like some machine codes, but that is 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.

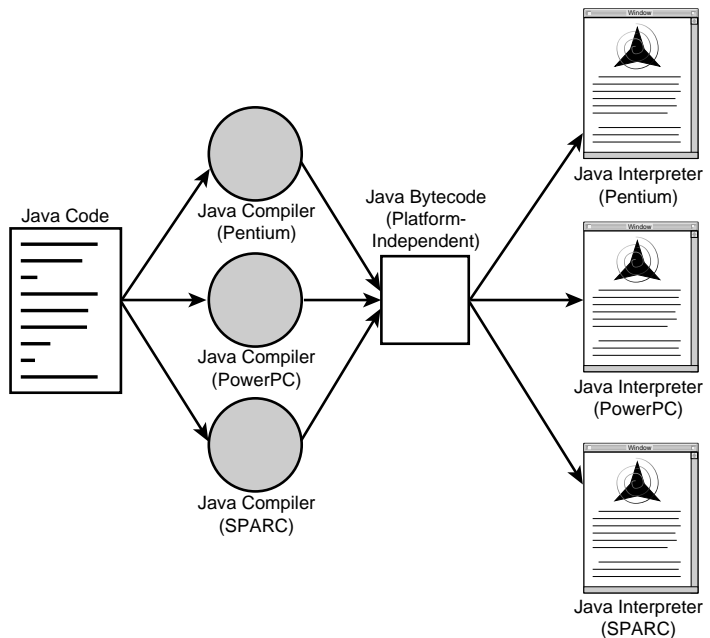
Things are different when you write code in Java. The Java development environment has two parts: a Java compiler and a Java interpreter. The Java compiler takes your Java program and instead of generating machine codes from your source files, it generates bytecodes.

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 of 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 2.2 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 JDK distribution. See the CD information for installation instructions.
- ☐ The 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 beta release. By the time you read this, The JDK may be available for other platforms, or other organizations may be selling Java development tools as well.

Although Netscape and other Java-aware browsers provide an environment for playing 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 applications 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 most other 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, ped, 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: }
```





An Introduction to Java Programming



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 routine called `main()`. In Java applications, as in a C or C++ program, `main()` is the first routine 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. Conventionally, 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
```



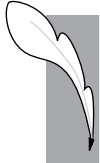
Note: In these examples, and in all the examples throughout this book, we'll be using Sun's Java compiler, part of the JDK. If you have a third-party development environment, check with the documentation for that program to see how to compile your Java programs.

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.



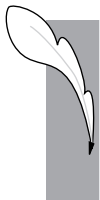
Note: Remember, the Java compiler and the Java interpreter are different things. You use the Java compiler (`javac`) for your Java source files to create `.class` files, and you use the Java interpreter (`java`) to actually run your class files.



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: Actually, 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.



An Introduction to Java Programming

Type

Listing 1.2. The Hello World applet.

```
1: import java.awt.Graphics;
2:
3: 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 interact with 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>
7: </BODY>
8: </HTML>
```



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.

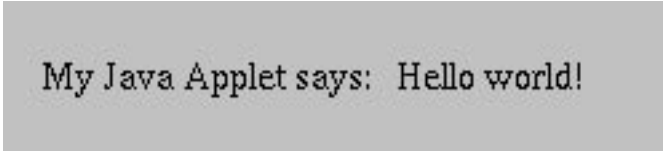




An Introduction to Java Programming

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 in the HotJava classes, rather than on using HotJava itself. Documentation for using the HotJava browser comes with the HotJava package.
- 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 HotJava 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.

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



WEEK
1

Object-Oriented Programming and Java

by Laura Lemay

Object-oriented programming (OOP) is one of the bigger programming buzzwords of recent years, and you can spend years learning all about object-oriented programming methodologies and how they can make your life easier than The Old Way of programming. It all comes down to organizing your programs in ways that echo how things are put together in the real world.

Today, you'll get an overview of object-oriented programming concepts in Java and how they relate to how you structure your own programs:

- ☐ What classes and objects are, and how they relate to each other
- ☐ The two main parts of a class or object: its behaviors and its attributes
- ☐ Class inheritance and how inheritance affects the way you design your programs
- ☐ Some information about packages and interfaces

If you're already familiar with object-oriented programming, much of today's lesson will be old hat to you. You may want to skim it and go to a movie today instead. Tomorrow, you'll get into more specific details.

Thinking in Objects: An Analogy

Consider, if you will, Legos. Legos, for those who do not spend much time with children, are small plastic building blocks in various colors and sizes. They have small round bits on one side that fit into small round holes on other Legos so that they fit together snugly to create larger shapes. With different Lego bits (Lego wheels, Lego engines, Lego hinges, Lego pulleys), you can put together castles, automobiles, giant robots that swallow cities, or just about anything else you can create. Each Lego bit is a small object that fits together with other small objects in predefined ways to create other larger objects.

Here's another example. You can walk into a computer store and, with a little background and often some help, assemble an entire PC computer system from various components: a motherboard, a CPU chip, a video card, a hard disk, a keyboard, and so on. Ideally, when you finish assembling all the various self-contained units, you have a system in which all the units work together to create a larger system with which you can solve the problems you bought the computer for in the first place.

Internally, each of those components may be vastly complicated and engineered by different companies with different methods of design. But you don't need to know how the component works, what every chip on the board does, or how, when you press the A key, an "A" gets sent to your computer. As the assembler of the overall system, each component you use is a self-contained unit, and all you are interested in is how the units interact with each other. Will this video card fit into the slots on the motherboard and will this monitor work with this video card? Will each particular component speak the right commands to the other components it interacts with so that each part of the computer is understood by every other part? Once you know what

the interactions are between the components and can match the interactions, putting together the overall system is easy.

What does this have to do with programming? Everything. Object-oriented programming works in exactly this same way. Using object-oriented programming, your overall program is made up of lots of different self-contained components (objects), each of which has a specific role in the program and all of which can talk to each other in predefined ways.

Objects and Classes

Object-oriented programming is modeled on how, in the real world, objects are often made up of many kinds of smaller objects. This capability of combining objects, however, is only one very general aspect of object-oriented programming. Object-oriented programming provides several other concepts and features to make creating and using objects easier and more flexible, and the most important of these features is that of classes.

NEW TERM A *class* is a template for multiple objects with similar features. Classes embody all the features of a particular set of objects.

When you write a program in an object-oriented language, you don't define actual objects. You define classes of objects.

For example, you might have a `Tree` class that describes the features of all trees (has leaves and roots, grows, creates chlorophyll). The `Tree` class serves as an abstract model for the concept of a tree—to reach out and grab, or interact with, or cut down a tree you have to have a concrete instance of that tree. Of course, once you have a tree class, you can create lots of different instances of that tree, and each different tree instance can have different features (short, tall, bushy, drops leaves in Autumn), while still behaving like and being immediately recognizable as a tree (see Figure 2.1).

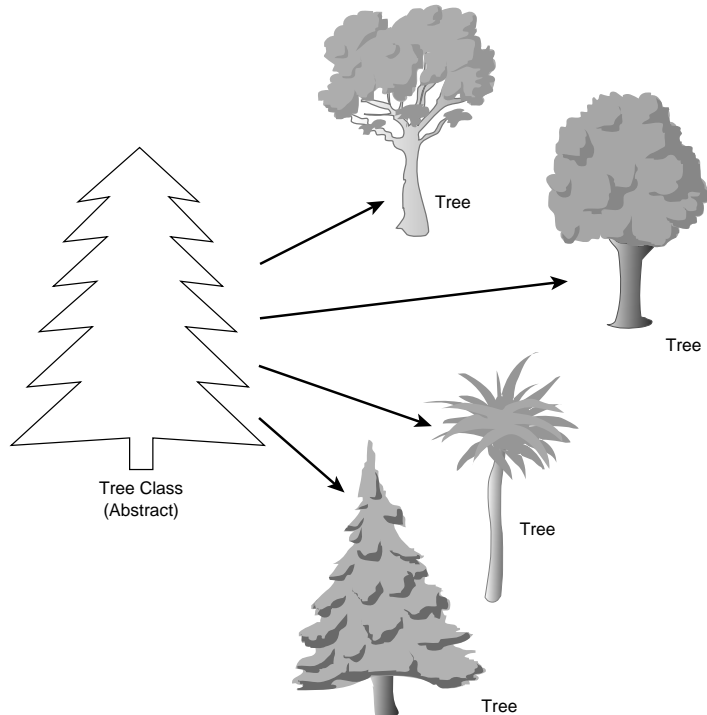
NEW TERM An *instance* of a class is another word for an actual object. If classes are an abstract representation of an object, an instance is its concrete representation.

So what, precisely, is the difference between an instance and an object? Nothing, really. Object is the more general term, but both instances and objects are the concrete representation of a class. In fact, the terms instance and object are often used interchangeably in OOP language. An instance of a tree and a tree object are both the same thing.

In an example closer to the sort of things you might want to do in Java programming, you might create a class for the user interface element called a button. The `Button` class defines the features of a button (its label, its size, its appearance) and how it behaves (does it need a single click or a double click to activate it, does it change color when it's clicked, what does it do when it's activated?). Once you define the `Button` class, you can then easily create instances of that button—that is, button objects—that all take on the basic features of the button as defined by

the class, but may have different appearances and behavior based on what you want that particular button to do. By creating a `Button` class, you don't have to keep rewriting the code for each individual button you want to use in your program, and you can reuse the `Button` class to create different kinds of buttons as you need them in this program and in other programs.

Figure 2.1.
The tree class and tree instances.



Tip: If you're used to programming in C, you can think of a class as sort of creating a new composite data type by using `struct` and `typedef`. Classes, however, can provide much more than just a collection of data, as you'll discover in the rest of today's lesson.

When you write a Java program, you design and construct a set of classes. Then, when your program runs, instances of those classes are created and discarded as needed. Your task, as a Java programmer, is to create the right set of classes to accomplish what your program needs to accomplish.

Fortunately, you don't have to start from the very beginning: the Java environment comes with a library of classes that implement a lot of the basic behavior you need—not only for basic programming tasks (classes to provide basic math functions, arrays, strings, and so on), but also for graphics and networking behavior. In many cases, the Java class libraries may be enough so that all you have to do in your Java program is create a single class that uses the standard class libraries. For complicated Java programs, you may have to create a whole set of classes with defined interactions between them.

NEW TERM A *class library* is a set of classes.

2

Behavior and Attributes

Every class you write in Java is generally made up of two components: attributes and behavior. In this section, you'll learn about each one as it applies to a theoretical class called `Motorcycle`. To finish up this section, you'll create the Java code to implement a representation of a motorcycle.

Attributes

Attributes are the individual things that differentiate one object from another and determine the appearance, state, or other qualities of that object. Let's create a theoretical class called `Motorcycle`. The attributes of a motorcycle might include the following:

- ☐ *Color*: red, green, silver, brown
- ☐ *Style*: cruiser, sport bike, standard
- ☐ *Make*: Honda, BMW, Bultaco

Attributes of an object can also include information about its state; for example, you could have features for engine condition (off or on) or current gear selected.

Attributes are defined by variables; in fact, you can consider them analogous to global variables for the entire object. Because each instance of a class can have different values for its variables, each variable is called an instance variable.

NEW TERM *Instance variables* define the attributes of an object. The class defines the kind of attribute, and each instance stores its own value for that attribute.

Each attribute, as the term is used here, has a single corresponding instance variable; changing the value of a variable changes the attribute of that object. Instance variables may be set when an object is created and stay constant throughout the life of the object, or they may be able to change at will as the program runs.

In addition to instance variables, there are also class variables, which apply to the class itself and to all its instances. Unlike instance variables, whose values are stored in the instance, class variables' values are stored in the class itself. You'll learn about class variables later on this week; you'll learn more specifics about instance variables tomorrow.

Behavior

A class's behavior determines what instances of that class do when their internal state changes or when that instance is asked to do something by another class or object. Behavior is the way objects can do anything to themselves or have anything done to them. For example, to go back to the theoretical `Motorcycle` class, here are some behaviors that the `Motorcycle` class might have:

- ☐ Start the engine
- ☐ Stop the engine
- ☐ Speed up
- ☐ Change gear
- ☐ Stall

To define an object's behavior, you create methods, which look and behave just like functions in other languages, but are defined inside a class. Java does not have functions defined outside classes (as C++ does).

NEW TERM

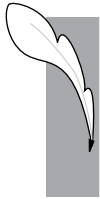
Methods are functions defined inside classes that operate on instances of those classes.

Methods don't always affect only a single object; objects communicate with each other using methods as well. A class or object can call methods in another class or object to communicate changes in the environment or to ask that object to change its state.

Just as there are instance and class variables, there are also instance and class methods. Instance methods (which are so common they're usually just called methods) apply and operate on an instance; class methods apply and operate on a class (or on other objects). You'll learn more about class methods later on this week.

Creating a Class

Up to this point, today's lesson has been pretty theoretical. In this section, you'll create a working example of the `Motorcycle` class so that you can see how instance variables and methods are defined in a class. You'll also create a Java application that creates a new instance of the `Motorcycle` class and shows its instance variables.



Note: I'm not going to go into a lot of detail about the actual syntax of this example here. Don't worry too much about it if you're not really sure what's going on; it will become clear to you later on this week. All you really need to worry about in this example is understanding the basic parts of this class definition.

Ready? Let's start with a basic class definition. Open up that editor and enter the following:

```
class Motorcycle {  
    }
```

Congratulations! You've now created a class. Of course, it doesn't do very much at the moment, but that's a Java class at its very simplest.

First, let's create some instance variables for this class—three of them, to be specific. Just below the first line, add the following three lines:

```
String make;  
String color;  
boolean engineState;
```

Here, you've created three instance variables: two, `make` and `color`, can contain `String` objects (`String` is part of that standard class library mentioned earlier). The third, `engineState`, is a `boolean` that refers to whether the engine is off or on.



Technical Note: `boolean` in Java is a real data type that can have the value `true` or `false`. Unlike C, `booleans` are not numbers. You'll hear about this again tomorrow so you won't forget.

Now let's add some behavior (methods) to the class. There are all kinds of things a motorcycle can do, but to keep things short, let's add just one method—a method that starts the engine. Add the following lines below the instance variables in your class definition:

```
void startEngine() {  
    if (engineState == true)  
        System.out.println("The engine is already on.");  
    else {  
        engineState = true;  
        System.out.println("The engine is now on.");  
    }  
}
```



The `startEngine` method tests to see whether the engine is already running (in the line `engineState == true`) and, if it is, merely prints a message to that effect. If the engine isn't already running, it changes the state of the engine to `true` and then prints a message.

With your methods and variables in place, save the program to a file called `Motorcycle.java` (remember, you should always name your Java files the same names as the class they define). Here's what your program should look like so far:

```
class Motorcycle {

    String make;
    String color;
    boolean engineState;

    void startEngine() {
        if (engineState == true)
            System.out.println("The engine is already on.");
        else {
            engineState = true;
            System.out.println("The engine is now on.");
        }
    }
}
```



Tip: The indentation of each part of the class isn't important to the Java compiler. Using some form of indentation, however, makes your class definition easier for you and for other people to read. The indentation used here, with instance variables and methods indented from the class definition, is the style used throughout this book. The Java class libraries use a similar indentation. You can choose any indentation style that you like.

Before you compile this class, let's add one more method. The `showAtts` method prints the current values of the instance variables in an instance of your `Motorcycle` class. Here's what it looks like:

```
void showAtts() {
    System.out.println("This motorcycle is a "
        + color + " " + make);
    if (engineState == true)
        System.out.println("The engine is on.");
    else System.out.println("The engine is off.");
}
```

The `showAtts` method prints two lines to the screen: the `make` and `color` of the motorcycle object, and whether or not the engine is on or off.

Save that file again and compile it using `javac`:

```
javac Motorcycle.java
```



Note: After this point, I'm going to assume you know how to compile and run Java programs. I won't repeat this information after this.



What happens if you now use the Java interpreter to run this compiled class? Try it. Java assumes that this class is an application and looks for a `main` method. This is just a class, however, so it doesn't have a `main` method. The Java interpreter (`java`) gives you an error like this one:

```
In class Motorcycle: void main(String argv[]) is not defined
```

To do something with the `Motorcycle` class—for example, to create instances of that class and play with them—you're going to need to create a Java application that uses this class or add a `main` method to this one. For simplicity's sake, let's do the latter. Listing 2.1 shows the `main()` method you'll add to the `Motorcycle` class (you'll go over what this does in a bit).



Listing 2.1. The `main()` method for `Motorcycle.java`.

```
1: public static void main (String args[]) {
2:     Motorcycle m = new Motorcycle();
3:     m.make = "Yamaha RZ350";
4:     m.color = "yellow";
5:     System.out.println("Calling showAtts...");
6:     m.showAtts();
7:     System.out.println("-----");
8:     System.out.println("Starting engine...");
9:     m.startEngine();
10:    System.out.println("-----");
11:    System.out.println("Calling showAtts...");
12:    m.showAtts();
13:    System.out.println("-----");
14:    System.out.println("Starting engine...");
15:    m.startEngine();
16:}
```

With the `main()` method, the `Motorcycle` class is now an application, and you can compile it again and this time it'll run. Here's how the output should look:



```
Calling showAtts...
This motorcycle is a yellow Yamaha RZ350
The engine is off.
-----
```

```
Starting engine...
The engine is now on.
-----
Calling showAtts...
This motorcycle is a yellow Yamaha RZ350
The engine is on.
-----
Starting engine...
The engine is already on.
```



The contents of the `main()` method are all going to look very new to you, so let's go through it line by line so that you at least have a basic idea of what it does (you'll get details about the specifics of all of this tomorrow and the day after).

The first line declares the `main()` method. The `main()` method always looks like this; you'll learn the specifics of each part later this week.

Line 2, `Motorcycle m = new Motorcycle()`, creates a new instance of the `Motorcycle` class and stores a reference to it in the variable `m`. Remember, you don't usually operate directly on classes in your Java programs; instead, you create objects from those classes and then modify and call methods in those objects.

Lines 3 and 4 set the instance variables for this motorcycle object: the `make` is now a `Yamaha RZ350` (a very pretty motorcycle from the mid-1980s), and the `color` is `yellow`.

Lines 5 and 6 call the `showAtts()` method, defined in your motorcycle object. (Actually, only 6 does; 5 just prints a message that you're about to call this method.) The new motorcycle object then prints out the values of its instance variables—the `make` and `color` as you set in the previous lines—and shows that the engine is off.

Line 7 prints a divider line to the screen; this is just for prettier output.

Line 9 calls the `startEngine()` method in the motorcycle object to start the engine. The engine should now be on.

Line 12 prints the values of the instance variables again. This time, the report should say the engine is now on.

Line 15 tries to start the engine again, just for fun. Because the engine is already on, this should print the error message.

Inheritance, Interfaces, and Packages

Now that you have a basic grasp of classes, objects, methods, variables, and how to put it all together in a Java program, it's time to confuse you again. Inheritance, interfaces, and packages are all mechanisms for organizing classes and class behaviors. The Java class libraries use all these concepts, and the best class libraries you write for your own programs will also use these concepts.

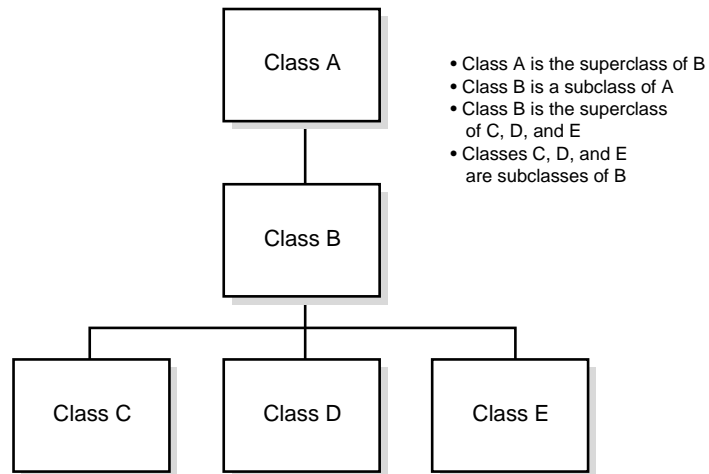
Inheritance

Inheritance is one of the most crucial concepts in object-oriented programming, and it has a very direct effect on how you design and write your Java classes. Inheritance is a powerful mechanism that means when you write a class you only have to specify how that class is different from some other class, while also giving you dynamic access to the information contained in those other classes.

NEW TERM With inheritance, all classes—those you write, those from other class libraries that you use, and those from the standard utility classes as well—are arranged in a strict hierarchy (see Figure 2.2).

Each class has a superclass (the class above it in the hierarchy), and each class can have one or more subclasses (classes below that class in the hierarchy). Classes further down in the hierarchy are said to inherit from classes further up in the hierarchy.

Figure 2.2.
A class hierarchy.



Subclasses inherit all the methods and variables from their superclasses—that is, in any particular class, if the superclass defines behavior that your class needs, you don't have to redefine it or copy that code from some other class. Your class automatically gets that behavior from its superclass, that superclass gets behavior from its superclass, and so on all the way up the hierarchy. Your class becomes a combination of all the features of the classes above it in the hierarchy.

At the top of the Java class hierarchy is the class `Object`; all classes inherit from this one superclass. `Object` is the most general class in the hierarchy; it defines behavior specific to all objects in the Java class hierarchy. Each class farther down in the hierarchy adds more information and becomes more tailored to a specific purpose. In this way, you can think of a class hierarchy as

defining very abstract concepts at the top of the hierarchy and those ideas becoming more concrete the farther down the chain of superclasses you go.

Most of the time when you write new Java classes, you'll want to create a class that has all the information some other class has, plus some extra information. For example, you may want a version of a `Button` with its own built-in label. To get all the `Button` information, all you have to do is define your class to inherit from `Button`. Your class will automatically get all the behavior defined in `Button` (and in `Button`'s superclasses), so all you have to worry about are the things that make your class different from `Button` itself. This mechanism for defining new classes as the differences between them and their superclasses is called *subclassing*.

NEW TERM *Subclassing* involves creating a new class that inherits from some other class in the class hierarchy. Using subclassing, you only need to define the differences between your class and its parent; the additional behavior is all available to your class through inheritance.

What if your class defines entirely new behavior, and isn't really a subclass of another class? Your class can also inherit directly from `Object`, which still allows it to fit neatly into the Java class hierarchy. In fact, if you create a class definition that doesn't indicate its superclass in the first line, Java automatically assumes you're inheriting from `Object`. The `Motorcycle` class you created in the previous section inherited from `Object`.

Creating a Class Hierarchy

If you're creating a larger set of classes, it makes sense for your classes not only to inherit from the existing class hierarchy, but also to make up a hierarchy themselves. This may take some planning beforehand when you're trying to figure out how to organize your Java code, but the advantages are significant once it's done:

- ☐ When you develop your classes in a hierarchy, you can factor out information common to multiple classes in superclasses, and then reuse that superclass's information over and over again. Each subclass gets that common information from its superclass.
- ☐ Changing (or inserting) a class further up in the hierarchy automatically changes the behavior of the lower classes—no need to change or recompile any of the lower classes, because they get the new information through inheritance and not by copying any of the code.

For example, let's go back to that `Motorcycle` class, and pretend you created a Java program to implement all the features of a motorcycle. It's done, it works, and everything is fine. Now, your next task is to create a Java class called `Car`.

`Car` and `Motorcycle` have many similar features—both are vehicles driven by engines. Both have transmissions and headlamps and speedometers. So, your first impulse may be to open up your `Motorcycle` class file and copy over a lot of the information you already defined into the new class `Car`.