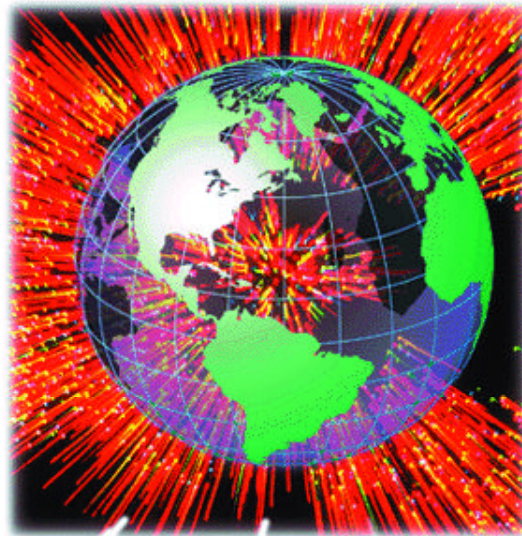


Internet Programming Series

JAVA

**PROGRAMMING LANGUAGE
HANDBOOK**

- ◆ Provides a hands-on guide to the complete Java language
- ◆ Step-by-step examples that explore key Internet programming concepts
- ◆ Includes special tips to help C/C++ programmers master Java



David H. Friedel, Jr. and Anthony Potts



**CORIOLIS
GROUP
BOOKS**

JAVA

PROGRAMMING LANGUAGE
HANDBOOK

Anthony Potts

David H. Friedel, Jr.

 CORIOLIS GROUP BOOKS

Publisher	<i>Keith Weiskamp</i>
Editor	<i>Keith Weiskamp</i>
Proofreader	<i>Kirsten Dewey</i>
Cover Design	<i>Gary Smith</i>
Interior Design	<i>Michelle Stroup</i>
Layout Production	<i>Kim Eoff</i>
Indexer	<i>Kirsten Dewey</i>

Trademarks: Java is a registered trademark of Sun Microsystems, Inc. All other brand names and product names included in this book are trademarks, registered trademarks, or trade names of their respective holders.

Copyright © 1996 by The Coriolis Group, Inc.

All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the written permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to The Coriolis Group.

The Coriolis Group
 7339 E. Acoma Drive, Suite 7
 Scottsdale, AZ 85260
 Phone: (602) 483-0192
 Fax: (602) 483-0193
 Web address: www.coriolis.com

ISBN 1-883577-77-2 : \$24.99

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

To my wife who has been there through it all.

Anthony Potts

To my sister Beth, who has helped make this book possible.

Dave Friedel

Acknowledgments

To Keith Weiskamp who really should be listed as a co-author for all the developmental and editorial work he did for this book.

To John Rodley for helping us get started with Java.

To Neil Bartlett, Alex Leslie, and Steve Simkin for all their help and for letting us have a sneak peek at their book, *Java Programming EXplorer*.

And, to Sun for creating a really cool alternative!

Contents

Foreword	The Crazy Years are Here Again	xv
----------	--------------------------------	----

Chapter 1 Introducing Java 1

The World of Java	4
<i>The Java Development Platform</i>	7
The Roots of Java	8
The Power of Distributed Software	10
<i>The Challenges of Security</i>	12
Java and C++	12
<i>Object-Oriented Quick Tour</i>	12
<i>What's Missing?</i>	13
<i>Gone: Pointers</i>	13
<i>Gone: Header Files</i>	13
<i>Gone: Multiple Inheritance</i>	14
<i>What's New?</i>	14
<i>Garbage Collection</i>	15
<i>Security</i>	15
<i>Exceptions</i>	15
<i>Strings versus Character Arrays</i>	16
<i>The Super Class</i>	16
<i>New Modifiers</i>	16
<i>The instanceof Operator</i>	17
<i>Helper Programs</i>	17

Chapter 2 Writing Your First Java Applet 19

Introducing the TickerTape Applet	22
<i>Running the Applet</i>	25
<i>Where's the Main Program?</i>	27
<i>Introducing Java Comments</i>	30

<i>What's in a Package?</i>	30
<i>Classes, Inheritance, and Interfaces</i>	32
<i>Types, Objects, and Constructors</i>	35
<i>Thank Goodness for Garbage Collection</i>	37
<i>Using Methods</i>	38
<i>Methods and Method Overriding</i>	43
<i>Graphic Methods</i>	46
<i>Working with Threads</i>	49
<i>Processing User Input</i>	52
One Last Thing	53
<i>That's It—Run It</i>	54

Chapter 3 Java Language Fundamentals 55

What Makes a Java Program? 58

Lexical Structure 58

<i>Comments</i>	59
<i>Identifiers</i>	65
<i>Keywords</i>	68
<i>Literals</i>	71
<i>Operators</i>	74
<i>Separators</i>	75

Types and Variables 76

<i>byte</i>	76
<i>short</i>	76
<i>int</i>	77
<i>long</i>	77
<i>float</i>	78
<i>double</i>	78
<i>boolean</i>	78
<i>char</i>	79
<i>string</i>	79

Variable Declarations 79

Using Arrays 82

<i>Declaring Arrays</i>	82
<i>Sizing Arrays</i>	83
<i>Accessing Array Elements</i>	83
<i>Multidimensional Arrays</i>	85

Using Command-Line Arguments 86

Passing Arguments 87
Reading in Arguments 87
Accessing Arguments 88
Dealing with Numeric Arguments 89

Chapter 4 Operators, Expressions, and Control Structures 91

Using Java Operators 93

Operator Precedence 93
Assignment Operators 95
Integer Operators 97
Boolean Operators 100
Floating-Point Number Operators 102

Using Casts 103

Writing Expressions and Statements 104

Control Flow Statements 106

if..else 106
while and do..while 108
switch 109
for 110
labels 111
Moving Ahead 112

Chapter 5 Java Classes and Methods 113

Understanding Classes 115

Declaring a Class 116

Using a Class 117

Components of a Class Declaration 118

Documentation Comment 119
Class Modifiers 119
Class Identifiers 124
Extending Classes 124
Using the implements Clause to Create Class Interfaces 126

<i>Class Body</i>	128
Methods	130
Declaring a Method	130
<i>Components of a Method Declaration</i>	130
<i>Method Modifiers</i>	131
<i>Return Type of a Method</i>	133
<i>Parameter Lists for a Method</i>	133
<i>Method Throws</i>	133
<i>Method Body</i>	134
<i>Using the this and super Keywords</i>	135
<i>Overloading and Overriding Methods</i>	137
Constructors—The Special Methods	138
<i>Components of a Constructor Declaration</i>	140
<i>Parameter List and Throws Clause</i>	146
<i>Constructor Body</i>	146
<i>Object Creation</i>	148
Variables for Classes	148
The Art of Casting with Classes	150

Chapter 6 Interfaces and Packages 155

Understanding Interfaces	158
<i>Declaring an Interface</i>	161
<i>Implementing an Interface</i>	161
<i>The Art of Casting with Interfaces</i>	165
<i>Tips on Implementing Interfaces</i>	167
Creating and Using Packages	168
<i>Naming and Referencing Packages</i>	170
<i>Declaration for Creating Packages</i>	171
Using Packages	174
<i>Declaration for Importing Packages</i>	176
<i>Standard Java Packages</i>	177
<i>Hiding Classes Using the Wild Card</i>	177

Chapter 7 Java Exceptions 179

Understanding Exceptions	182
Do You Really Need Exceptions?	183
<i>Defining a Try Clause</i>	186

<i>Using the catch Statement</i>	187
<i>When to Use the finally Statement</i>	189
<i>The Hierarchy of Exceptions</i>	190
<i>Declaring a Method Capable of Throwing Exceptions</i>	194
<i>Throwing Exceptions</i>	197
<i>When to Catch and When to Throw</i>	198
Knowing When to Create Your Own Exceptions	200

Chapter 8 Threads 203

What Is a Thread?	205
<i>Creating a Thread</i>	210
<i>Subclassing the Thread Class</i>	210
<i>Implementing the Runnable Interface</i>	211
Initializing a Thread	213
<i>Who Goes First; Who Finishes Last?</i>	214
<i>Priority versus FIFO Scheduling</i>	215
Controlling the Life of a Thread	216
<i>The start() Method</i>	216
<i>The run() Method</i>	217
<i>The sleep() Method</i>	218
<i>The suspend() Method</i>	218
<i>The resume() Method</i>	218
<i>The yield() Method</i>	219
<i>The stop() Method</i>	219
<i>The destroy() Method</i>	220
Multiple Objects Interacting with One Source	220
<i>Synchronizing Revisited</i>	220
Wait() a Second... Notify() Me When...	222
Grouping Your Threads	223

Chapter 9 The Java AWT 225

Introducing the AWT	227
<i>Introducing the Layout Manager</i>	228
<i>What About Menus?</i>	229
The AWT Hierarchy	229

x Contents

The Component Class 231

Key Component Class Methods 231

The Frame Class 235

Hierarchy for Frame 235

Declaration for Frame 235

Methods for the Frame Class 237

The Panel Class 238

Hierarchy for Panel 240

Declaration for Panel 240

Methods for Panel 241

The Label Class 241

Hierarchy for Label 241

Declaration for Label 241

Methods for Label 242

Button Class 243

Hierarchy for Button 243

Declaration for Button 243

Methods for Button 244

The Canvas Class 244

Hierarchy for Canvas 244

Declaration for Canvas 245

Methods for Canvas 245

The Checkbox Class 245

Hierarchy for Checkbox 246

Methods for Checkbox 247

The Choice Class 247

Hierarchy for Choice 248

Declaration for Choice 248

Methods for Choice 248

The List Class 250

Hierarchy for List 250

Declaration for List 251

Methods for List 251

TextField and TextArea Classes 253

Hierarchy for TextField and TextArea 254

Declaration for TextField and TextArea 254

Methods for TextField and TextArea 255

The Scrollbar Class 258

Hierarchy for Scrollbar 260

Declaration for Scrollbar 260

Methods for Scrollbar 260

Building Menus 261

The MenuBar Class 262

Hierarchy for MenuBar 262

Declaration for MenuBar 262

Methods for MenuBar 262

The Menu Class 263

Hierarchy for Menu 263

Declaration for Menu 264

Methods for Menu 264

The MenuItem Class 265

Hierarchy for MenuItem 266

Declaration for MenuItem 266

Methods for MenuItem 267

Creating a Sample Menu Application 268

Working with the Layout Manager 270

The FlowLayout Class 271

Declaration for FlowLayout 271

Methods for FlowLayout 273

The BorderLayout Class 274

Declaration for BorderLayout 275

Methods for BorderLayout 275

The GridLayout Class 276

Declaration for GridLayout 277

Methods for GridLayout 277

The GridBagLayout Class 278

Declaration for GridBagLayout 281

Methods for GridBagLayout 281

The CardLayout Class 282

Declaration for CardLayout 282

Methods for CardLayout 282

Chapter 10 Java Applet Programming Techniques 285

Applet Basics 287

Hierarchy of the Applet Class 288

Applet Drawbacks 291

Let's Play 293

Interacting with the Browser 294

Changing the Status Bar 296

Playing Sounds 297

Displaying Images 298

Chapter 11 Event Handling 301

Introducing Events 303

Event Types 304

The Event Class 304

Mouse Events 307

Keyboard Events 311

Hierarchy of Events 313

Processing System Events 315

Chapter 12 Streams and File I/O 319

Introducing the System Class 321

Different Flavors of Streams 323

InputStream and OutputStream Classes 324

BufferedInputStream and

BufferedOutputStream Classes 326

ByteArrayInputStream and ByteArrayOutputStream Classes 328

DataInputStream and DataOutputStream Classes 330

FileInputStream and FileOutputStream Classes 333

FilterInputStream and

FilterOutputStream Classes 335

LineNumberInputStream Class 337

PipedInputStream and

PipedOutputStream Classes 339

PrintStream Class 340

PushbackInputStream Class 342

SequenceInputStream Class 342

StringBufferInputStream Class 343

Chapter 13 Networking with Java 345

Understanding the Basics of Networking 347

TCP/IP 348

SMTP 348

FTP 349

HTTP 349

NNTP 349

Finger 349

WhoIs 349

The Client/Server Model 350

Ports of Interest 350

Introducing the java.net Package 352

Networking Applications 353

Working with Internet Addressing 353

The Role of the Client 355

Creating a Socket 355

Using Sockets 355

Creating a Sample Client Application 356

Bring on the Server 360

Creating a Sample Server Application 361

Web Content Only Please 364

Using the URLConnection Class 366

Networking between Applets 367

Passing Information to a Fellow Applet 367

Appendix A Online Java Resources 375

Appendix B Java Database Connectivity (JDBC) 387

Index 403

Foreword

The Crazy Years Are Here Again

By Jeff Duntemann

As a brand new magazine editor at the end of 1984, I surveyed the IBM PC field and commented under my breath, “These are the Crazy Years.” The IBM PC world had begun exploding in 1983; *PC Magazine* hit 800 pages; new companies were forming every minute and spending huge amounts of money launching new products into an astonishingly hungry and immature market. The machines of the time were almost unimaginably underpowered. 5Mhz. 8 bits. 256K of RAM. Only rich people had hard drives. And yet everyone spoke of their miserable little IBM PCs as though they could do the work of mainframes—we just hadn’t yet figured out quite *how*.

History doesn’t repeat itself—but it echoes, it echoes. Here we sit, eleven years later, I’m still a magazine editor (though not nearly as new) and the Crazy Years are back again. This time, plug the Internet into the place the IBM PC occupied in 1984. Our PCs really are mainframes now; those 166 Mhz Pentiums that we take for granted can handle anything we throw at them. What we’re enraptured with today is the ability to connect to a server and bounce around the world like manic pinballs, grabbing a Web page here, a shareware file there, a picture of Cindy Crawford somewhere else. New Internet magazines are appearing weekly, enormous sums of money are being spent and earned on Internet technology companies, and Internet books have crowded almost everything else off of the computer book shelves at the superstores. The Internet will become the OS of the future. Applications will be fragmented and distributed around the world; a piece in Britain, a piece in Finland; a piece in Rio. Faithful agent software will wander around the globe, sniffing out what we want and paying for it with digital cash. Our Internet boxes will be our phones, our faxes, our stereos, our game platforms, and our personal bank machines.

xvi Foreword

Yikes! Aren't we getting maybe just a *little* bit ahead of ourselves?

Sure. *But admitting we're ahead of ourselves doesn't mean that we won't catch up.*

And that's why things are so crazy. Somewhere south of our conscious minds we understand that this is the way the world is going, even if the results in the here and now fall just a touch short of spectacular. We're making this up as we go along—there's no established body of technical knowledge to fall back on—so everything's being tried, and everything's being trumpeted as the ultimate Magic Bullet.

The trick, of course, is to get behind the right bullet. (Which is always better than being in *front* of it...) The game is far from over (especially with that Big Rich Guy still knocking around in the upper left corner of the country, making trouble) but if you want my penny and a half, the bullet to get behind for the Internet game is Java.

You can look at Java in a number of different ways, from C++ with all the barbed wire stripped out of it, to the ultimate global cross-platform integration scripting language. That's a strong element of its success: Good magic bullets are never too specific, or the Atari personal computers (which were muscular game boxes) would have knocked the underpowered but protean IBM PC right out of the ring. Again, there is that never-quite-fully-expressed feeling that Java was created with exactly the *right* balance of power and specificity, and that it can rise to whatever challenges we might put in front of it, given time for us to pry the devil out of the details and toss him in the junk drawer.

No, there isn't enough bandwidth on the Internet to do all we want to do. *This year.*

No, the majority of people do not have fast, always-there connections to the Internet. *This year.*

No, there is no clean, standard, and acceptably secure way for people to pay for electronic deliverables over the Internet. *This year.*

Care to place bets on next year? Or (for higher stakes yet) the year after that?

I didn't think so.

Whatever you do, avoid the Clifford Stoll Trap. Stoll, one of the original architects of the Internet, wrote an entire book about how the Internet was a major

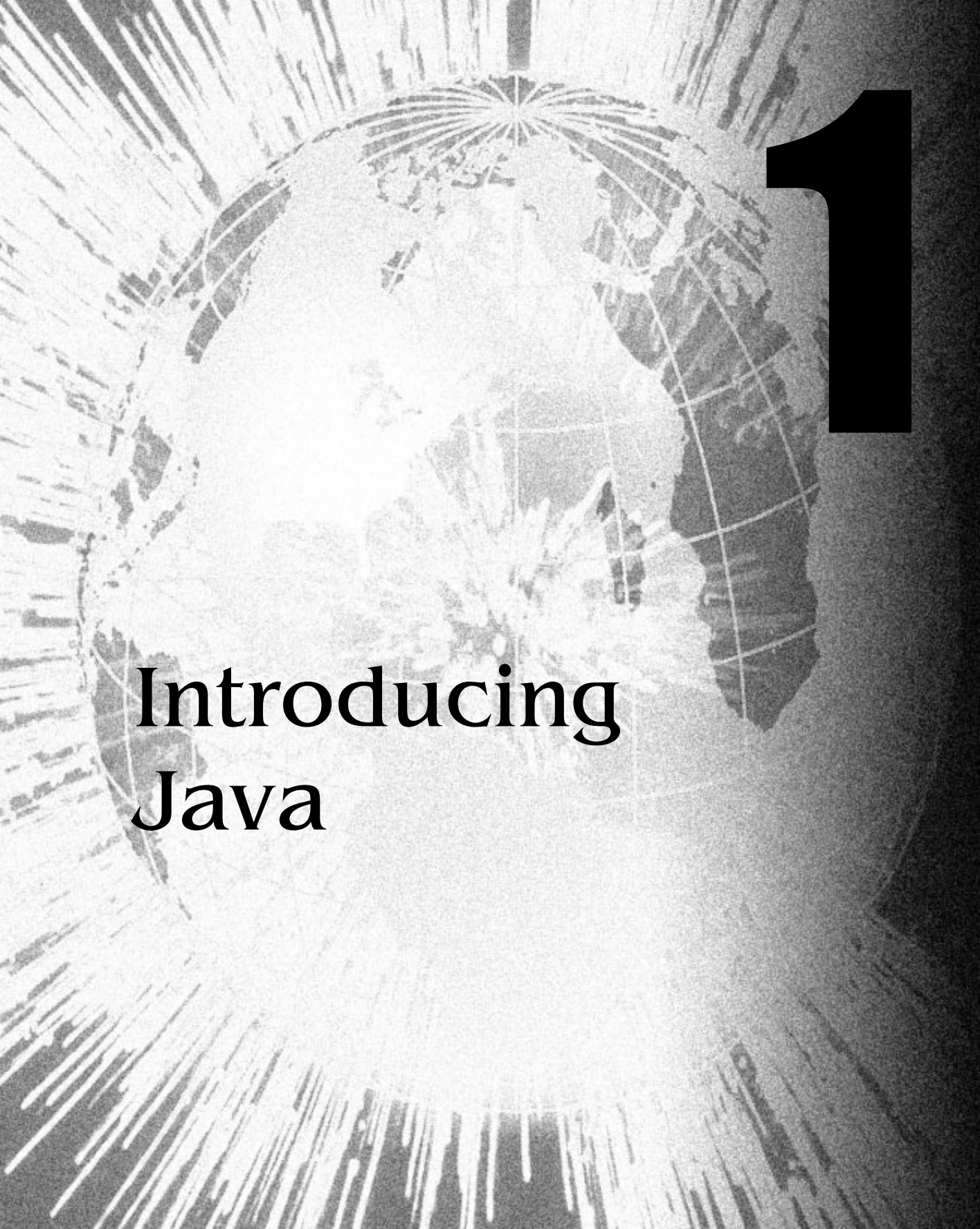
shuck, that it was ruining all our lives, that he was giving it up forever, and don't expect him to come back and rescue us from ourselves, farewell, good bye forever, I'm leaving and I'm serious, don't try to stop me, and on and on and on.

We waved good-bye. Now he's gone. And not only is he not especially missed, I doubt that one person in a hundred even remembers who he was. That's how our business works. If you stand back and let the train go by, you will not be missed, and the train generally passes through your station only once.

That's why I encourage you to stick with this stuff, no matter how crazy it gets. As best we can tell right now, Java is the brand of pipe that we'll be using to plumb global communications software for the foreseeable future. No, we don't have the whole foundation poured, and no, we don't have a stack of plans to go by. Still, without the pipes in place, it won't work. You have to develop the skills you'll need to do the work next year and the year after. What better time than now?

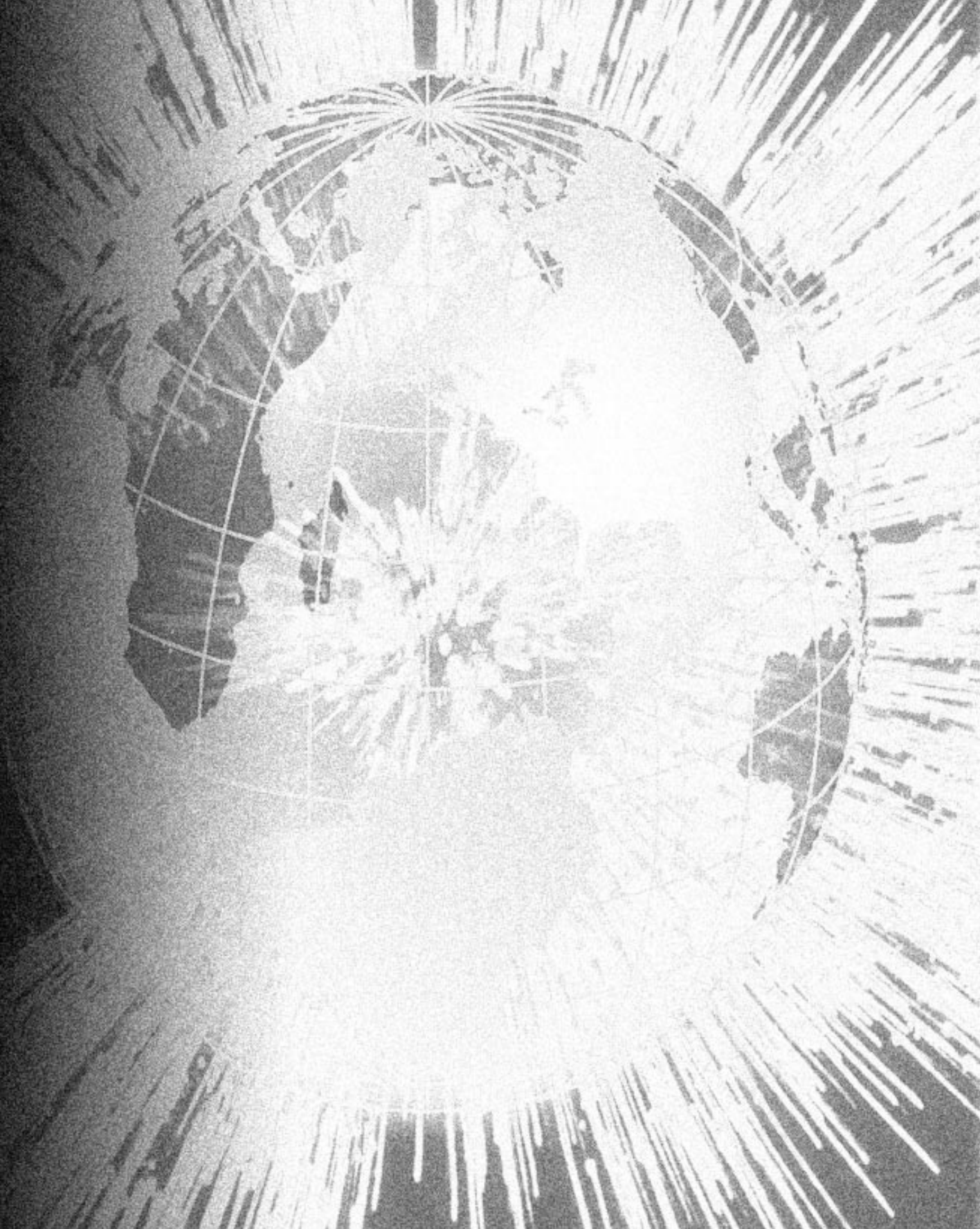
You've got the book in hand to take you through the Java language and make it stick. So do it!

Or heads will bounce.



1

Introducing Java



Introducing Java



Java has swept the computer industry with its promise to deliver executable content to the vast sea of computers connected to the World Wide Web. Here's a look at why you'll want to start developing with this language as soon as you can.

In just a few months, Java has moved from the R&D labs at Sun Microsystems to the center stage of the World Wide Web. Never before had a simple programming language garnered so much attention and captured the imaginations of so many software developers and computer users alike so quickly. Some cynics think that the best part about Java is its name, which is also the reason they think Java gets so much attention in the press. But most experts who follow Web development think that Java is the most significant thing that has been developed or announced for the Web.

Why has Java taken over so quickly? The short answer is found in its platform independence and potential to turn the Web into a much more dynamic and interactive environment—something that is badly needed. Other reasons are because of Java's similarity to C++ and its support of popular object-oriented programming techniques, making it easier for hundreds of thousands of C and C++ programmers to quickly master Java's powerful features.

Our goal in this chapter is to set the stage for Java, exploring where this language has come from and where it is going. We'll introduce the key features of Java, give you some insight into why Java was developed in the first place, and examine some of the key similarities between Java and C++. We think it is important to spend as much time as possible looking at Java through the lenses of a C++ programmer because the syntax and object-oriented features are very similar.

The World of Java

“Java may be overhyped, but it also may be as significant to the computer world as television was to broadcasting.”

—*U.S. News and World Report*

In the old days of computer languages (less than six months ago), programs were designed to run under a single operating system, more or less, and the name of the game was to create programs that could run as fast as possible. Almost overnight, the World Wide Web and Java have changed this notion of operating system-based language environments to platform independent network-driven languages and systems. Java represents the sea change of distributed programming and application development that is taking place in the computer industry today. Languages like Java are radically shifting the computing burden from local desktop computers to servers that deliver executable content to users.

Programmers have accepted Java very quickly because it provides everything that is needed in a modern day language including:

- Object-Oriented Features
- Multithreading
- Garbage Collection (Automatic Memory Management)
- Networking and Security Features
- Platform Independence (Architecture Neutral)
- Internet/Web Development Features

(If you are unfamiliar with some of this terminology, make sure you read the *Java Jargon Survival Guide* included in this chapter.) Popular languages like Smalltalk and C++ have been moving programmers away from top-down structured programming methods and into the more flexible and adaptable object-oriented paradigm for years. Java greatly contributes to this evolution and even enhances some of the shortcomings found in object-oriented languages like C++.

What is most remarkable about Java is that it is the first language that allows developers to create robust cross-platform networked software for the Internet. Once you start using Java, you can throw away the notion that all software must first be developed on a specific platform to be run on the same platform and then ported if it needs to run on other systems.

If you have seen Java in action by using your Web browser to view Web pages that contain Java applets, you already know some of the types of programs you can write. But Java applets are only half of the story. The Java language can be used to write both applets and standalone applications. Applets are incorporated into Web pages using a special `<APPLET>` HTML tag and they are downloaded and launched automatically when their pages are displayed by a Java-enabled Web browser. This process is similar to the way in which a Web browser might process and display other elements such as images and hyperlinked text. The big difference with an applet is that the browser processes *dynamic executable content* instead of static data.

A Java application, on the other hand, looks suspiciously like a C++ program. It can run on its own and perform a myriad of tasks from performing calculations to specialized file I/O operations. The only problem with writing Java applications at the moment is that Java is an interpreted language, and thus programs written in Java require the Java Virtual Machine in order to execute. Fortunately, work is underway to develop compilers that enable Java applications to run quicker and more independently.

Java Jargon Survival Guide

Architecture-Natural This is a term language designers use to describe languages like Java that are truly portable across different operating systems. Programs written in architecture-natural languages typically run under bytecode interpreters that are capable of running on any type of computer.

Bytecodes The entire language design of Java is based on the notion of bytecode interpreters, which can efficiently run ("interpret") programs at runtime and perform operations like error handling and memory management. To create and use bytecodes, a compiler must first translate a program into a series of codes which are then executed by an interpreter which converts the general bytecode instructions into local actions.

Classes Java utilizes the basic object technology found in C++, which in turn was heavily influenced by Smalltalk. Object-oriented features are implemented in these languages using basic building blocks called *classes*. Essentially, a class is a structure

6 Chapter 1

that allows you to combine data and the code that must operate on the data under one roof. Once classes have been defined, they can easily be used to derive other classes.

Distributed Programming This emerging field of software development involves the techniques of writing programs that can be executed across networks like the Internet. In traditional programming, applications run on a single computer and only data is distributed across networks. In distributed programming, programs can be downloaded from networks and executed at the same time. As you might guess, this approach opens the door wide for ways in which software can be shared and distributed.

Garbage Collection This is the memory management technique that Java programs use to handle dynamic memory allocation. In traditional compiled languages like C and C++, memory for dynamic variables and objects (pointers), must be allocated and deallocated manually by the programmer. In Java, memory allocation is handled by the Java runtime environment, thus eliminating the need for explicit memory allocation and pointers. Garbage collection is one key feature that many programmers claim make "Java a better and safer C++."

HotJava This is Sun's Web browser written in the Java language. It contains the Java Virtual Machine and can download and play any Java applet. Originally, HotJava was the rage on the Web but since Netscape has licensed and incorporated the Java Virtual Machine into Netscape Navigator, most Web users have forgotten about HotJava.

Java Applet These are small programs written in Java that can execute remotely over networks. They are downloaded from the Web via a Java-enabled Web browser and then they are executed from within the shell of the browser. An applet can be anything from animations to search engines to games. In the short time that Java has been available, hundreds of applets have appeared on the Web written by programmers from all around the world.

Java Virtual Machine This is the code that serves as the engine or interpreter for Java programs. Any application that is capable of running Java programs requires the use of the Java

Virtual Machine (VM). The process for running a Java applet involves compiling the applet into a sequence of bytecodes which are then interpreted by the Java VM at runtime. Sun has aggressively licensed the Java VM to many companies, such as Netscape, Oracle, and Borland, to help expand the developer base for Java.

Java-Enabled A term used to indicate if Internet applications like Web browsers are capable of running Java applications, in particular, Java applets.

JavaScript This is an object-oriented scripting language developed by both Sun and Netscape. The language was designed to be used as a scripting language to customize Netscape browsers and control Java applets running under Netscape. Originally, Netscape called their scripting language “LiveScript” but the name was changed to JavaScript. Rumor has it that Netscape’s stock rose \$20 per share in one day after announcing the name change. You can think of JavaScript as the “glue” between Java applets and Netscape browser features such as plug-ins, HTML, and special events.

Just-in-Time Compiler This is new compiler technology that is being developed for Java so that Java applications can be custom compiled for a particular platform as they are downloaded from networks.

Methods These are the functions (operations) that are included in Java classes to operate on data.

Multithreading To allow Java applications and applets to run efficiently even though they must be executed by a Java interpreter, Java supports a technique called *multithreading* that allows different processes to execute at the same time.

The Java Development Platform

As we’ve mentioned, a Java-enabled browser such as HotJava or Netscape 2 is needed in order to run Java applets. You can also use the *appletviewer* utility which is provided with Sun’s Java Development Kit (JDK). For any Java programming that you wish to do, you’ll need the JDK because it provides the compiler for compiling Java applets and applications (*javac*), an interpreter for

8 Chapter 1

running standalone Java applications (*java*), and a debugger (*jdbg*). But don't be surprised to find that each of these development tools are somewhat primitive and must be run from the command line. In the near future, we should have much better Java visual development tools. If you don't currently have the JDK, you can visit Sun's Java Web site to download a copy (<http://www.javasoft.com/>). The syntax and command options for using the tools in the JDK are presented with the Java Programming Language reference card included at the end of this book.

All of the tools included in the JDK are designed to support Sun's notion of what the Java language is all about including:

- A compiler for the Java language that generates architecture-neutral bytecodes
- The Java Virtual Machine that interprets bytecodes at runtime.
- A set of class libraries to help Java programmers create applications. Some of these libraries include interface tools, I/O, applet development, networking, and so on.
- A Java runtime environment that supports bytecode verification, multithreading, and garbage collection
- Java development support tools including a debugger, documentation generator, and so on.

The Roots of Java

"Java has become perhaps the first programming language to move the stock market."

—*Application Development Trends*

Whether or not you believe all the hype surrounding Java, no one will deny that Java is going places. Already numerous major software and hardware companies have licensed Java, including Netscape, IBM (Lotus), Borland, Adobe, Fujitsu, Mitsubishi, Symantec, Spyglass, Macromedia, and even Microsoft. But before we look at the key elements of Java and where it is going, let's explore its roots to give you some perspective. Like many great technological creations, Java's development progressed with a number of twists and turns in the road.

The origins of Java began in April 1991 with a small development team at Sun headed by James Gosling. Gosling had developed quite a reputation in the past as a legendary Unix programmer for creating the first C version of a popular

Unix editor named *Emacs*. He also developed the innovative Postscript-based windowing environment for Sun OS called *NEWS*. In the early days, Gosling's team operated as an independent unit from Sun. They went by the name "the Green group" and Sun eventually set this group up as a separate company called *First Person*. (Don't you wish you owned stock in this company?) Their initial charter was to develop software that could be sold to consumer electronics companies. Sun felt that many consumer-driven technologies, such as PDAs, set-top boxes, and so on, were up and coming technologies worth pursuing to enhance their base of software sales.

Soon after the Green group set up shop, they discovered that the success of creating widely-distributed software for consumer products would only come if a platform-independent development environment were created. They began work in this area by trying to extend existing C++ development tools and compilers but this turned out to be a dead end of sorts because of the complexity of C++. C and C++ have always been promoted as highly portable languages, but when it comes right down to it, trying to create general purpose, portable, and robust code in these languages is not so easy.

When you run into a wall in software development, the best thing to do is develop a language that provides the solutions you need. And that's exactly what Gosling and the Green group did. Their new language was originally called *Oak*—named after a tree outside Gosling's office. Because Oak could not be trademarked by Sun, a new name emerged—Java—after a brainstorming session or two.

The overriding goal of Java's developers was simple but ambitious: *Design a programming language that can run on anything connected to a network*. This would include Sun workstations, PCs, Macs, PDAs, and even television sets and toasters. To meet these goals, they borrowed heavily from existing languages such as C++, Objective-C, Smalltalk, Eiffel, and Cedar-Mesa. Java's developers also wanted to make sure their language achieved an entirely new level of robustness, not found in languages like C++ because of all the dangerous features like pointers, operator overloading, and dynamic memory allocation. One writer in a popular computer developer's journal summed this goal up nicely when he wrote that Java's developers wanted to "get rid of all the complicated, dangerous, and/or stupid crud in C++." The end result is that you won't find features like the following in Java:

- Header files
- #defines

10 Chapter 1

- typedefs or structs
- Pointer arithmetic
- Multiple inheritance of classes
- General functions (Only methods are supported)

On May 23, 1995, Sun introduced the Java language and its corresponding browser *HotJava*. The language and its associated development tools only took about four years to create, but as the Green group proved, four years was ample time to create a new standard for the rest of the world to follow. Soon after the announcement of Java, alpha versions of the language started appearing across the World Wide Web. Another noteworthy date to mark on your calendar is December 7, 1995—the date Microsoft agreed to license Java, an endorsement that has helped to accelerate Java’s popularity. (If you owned any of Sun’s stock, you probably noticed that their stock increased by \$336 million on that day!)

The Power of Distributed Software

“I have seen the future of the World Wide Web, and it is executable content.”
—Ray Valdes, *Dr Dobb’s Developer Update*

Before Java was unleashed on the world, most Web development and interactivity was accomplished using Common Gateway Interface (CGI) scripting. For the past year, the key scripting contender has been Perl. The processing model for CGI is completely client-server based. For example, a user on a client computer running a software program like Netscape fills out a form on a Web page and the data is sent to a server. Then, the server reads and processes the data and sends a response back to the client. The disadvantage of this approach is that the client operates almost like a dumb terminal; most of the key processing tasks are performed by a central computer—the server. And if the server is busy (which happens a lot in the Web world), the client must wait *and wait and wait*.

In the world of distributed software, networks are used to send executable code, often called *executable content*, to client computers which are capable of running the software locally. For many software developers and users, this is a dream come true. In fact over the past year, many leading software development companies have been trying to create standards for delivering executable content. Some of the more noteworthy attempts include Macromedia’s Shockwave, NEXT’s WebObjects, and Microsoft’s new ActiveX controls.

The visual benefits of running distributed software like Java applets are only the tip of the iceberg. Of course, it is impressive to see a well-designed animated applet dance across a Web page as its code is being downloaded across a network and executed locally, but Java programmers are already looking forward to the day in the not so distant future when they can develop major applications that can run across networks. This dynamic flexibility will open up new possibilities for both updatable entertainment and business-related software. And the best part is that programmers will no longer have to write applications that they have to port to multiple platforms. The same program written in Java can run on any type of computer that can connect to a network and run the Java Virtual Machine.

But the best part is that you can use Java today in its current form and take advantage of some of the key benefits the distributed software paradigm has over the CGI client-server approach:

Develop Interactive Web Interfaces With Java you can create much more interactive interfaces to the Web than you can with CGI languages like Perl. Applets that you customize for Web pages can allow users to move objects on the screen, view animations, and control multimedia features, such as sound and music.

Utilize Local Resources With the CGI model, a server is limited to processing the data it has on hand. With Java, on the other hand, you can write applications that truly take advantage of resources available on a user's local system. For example, a Java program might use local hardware features in a way that a CGI program never could. The Java approach allows the local computer to take full control over how and where code is executed.

Greater Internet/Web Access One of the biggest problems with the Internet and the Web is that content is scattered all over the place in a somewhat chaotic fashion. Using Java, you can write better front-ends to the Web, such as agents and search engines, to better access the Web.

Reduce the Cost of Distributing Software The software industry has rapidly turned into a "hits" based business, which means that computer software outlets typically only carry the major blockbuster products. One of the reasons this has occurred is that the cost and risks of selling and distributing software have greatly increased over the past five years. With distributed software, users can purchase and download the software they need instead of having to order from a direct mail catalog or buying it in a store. This approach of getting software from a publisher or developer to a user also is ideal for updating software. If you need a new version of your favorite tax program to get your taxes done by April 15, you can simply point to the right place on the Web and quickly access the software

you need.

The Challenges of Security

Along with the promises and opportunities of distributed software, come the risks of security. Think about it. The programs you run on your desktop computer are ones that you've decided to buy or download. Once you install them, you can check them for viruses and remove them if they cause problems on your system. Distributed programs such as Java applets, on the other hand, reside on someone else's computer. When you run them you are essentially downloading executable code from another computer, of which you have no control over.

Fortunately, the underlying philosophy behind Java's design is security. Bytecodes that are downloaded from a network are passed to a bytecode verifier which attempts to weed out bad code that could damage a local computer. Because Java has no pointers or programmer-driven memory allocation routines, Java code is less likely to go off track and crash a local computer due to illegal memory access operations. The absence of pointers also keeps troublesome hackers from writing code that accesses a local computer's system memory to get unauthorized privileges.

By design, the Java Virtual Machine assumes that code downloaded from a network should be trusted less than code that is resident on a local computer. To enhance security, Java provides built-in classes that check for security related conflicts. As a final measure, Java allows its user *layer of security* to be configurable. For example, a user can specify exactly which directories applets can read from and write to. Applets can also be limited to accessing sockets on other machines.

Java and C++

If you haven't noticed already, Java is very similar to C++. If you are an accomplished C++ programmer, moving to Java will be easy for you. However, there are a few important things you should know that we will present in this chapter. If you are new to both C++ and Java, you may have a little more catching up to do to understand the object-oriented nature of the Java language.

Object-Oriented Quick Tour

Let's start by looking at some key object-oriented programming issues. First of all, C++ is not a "true" object-oriented (OO) language but Java is much closer. Why? Because *everything* in Java is a class and *all* method calls are done via a

method invocation of a Java class object. In C++ you can use stand-alone functions, header files, global variables, and so on. This is an extremely important point, so don't gloss over it. The only thing in Java not placed in a class is interfaces, although they are used like classes but without implementations.

This strict OO nature means that you won't be able to port C++ code as easily. You will need to change the basic structure of your C++ applications, although you should be able to keep the logic as long as you are not using any of the features that have been removed.

What's Missing?

As we've mentioned, one of big goals for the developers of Java was to look at all the other programming languages and pull the best features of each and dump the rest. Since C/C++ has such a large installed base of programmers, it is obvious why they chose to mimic so much of its syntax and structure. There are, however, several features that C++ has that Java does not implement. Many of these subtractions were made for security reasons, since Java was designed as a Web language. Other features were left out because the Java creators thought they were too difficult to use or just plain useless. Let's look at some of the important subtractions.

Gone: Pointers

Pointer arithmetic is the bane of everyone who hates C++. For the few programmers who have mastered pointers, we salute you. For the rest of us, good riddance. The major reason pointers are not used with Java is security. If a Java applet had the ability to access memory directly, it could easily cause some real problems. By forcing the Java interpreter to handle memory allocation and garbage collection, it relieves you of a big burden and lessens the chance that anyone can do bad things to your computer through a Java program.

There are a few areas where pointers seem necessary for performing certain operations. But since we don't have them in Java, we need to find a way around them. In Java, objects are passed as arguments directly instead of passing a pointer to an object's memory space. You must also use indices to manipulate arrays rather than accessing the values directly.

Gone: Header Files

To C++ users, header files are a mainstay of programming life. However, if you look closely at how most programmers use header files, you'll find that the big-

gest use is for prototyping and documentation. To examine the interface to a certain member function, you can read a header file and find the function. By just looking at the header files from a C++ class, you can figure out a lot about what that class does—without ever seeing any of the implementation.

In Java there is no way to do this since the implementation for classes and methods must reside in the same place. The implementation always follows the declaration in Java. Of course, you can add all the comments you want to aid in understanding your code, but your code may run on for pages and pages. It is not always easy to look at a Java class and understand how it can be used.

So, why doesn't Java use header files? There are two reasons. First, it is not possible to use a library that declares a method but does not implement it. Second, it is more difficult to program using files that are out of synchronization with the implementation.

Gone: Multiple Inheritance

Very often in object-oriented programming, an object needs to inherit the functionality of more than one class. In C++ this is accomplished using multiple inheritance, a technique allowing a single class to subclass as many classes as it needs. Multiple inheritance can get extremely complicated and is one of the leading causes of C++ bugs (and programmer suicide). Java's answer to multiple inheritance is *interfaces*. Interfaces are the only item in Java that are not a class. They are simply templates for a class to follow. They list method declarations with no implementation (no guts). Any class that implements an interface *must* use the methods declared in the interface.

Interfaces work well, but they do have some limitations. The big drawback is that you must write the code for each method that an interface declares. Multiple inheritance in C++ allows you to override the methods that you chose and then you can just use the parent's implementation of the methods for the others.

Interfaces are much easier to understand and master than multiple inheritance. Check Chapter 6 for an in-depth look at interfaces. With the right programming strategy, you can get almost all of the functionality of multiple inheritance. And hopefully, you won't have all the problems.

What's New?

Since Java is supposed to be the next step in the evolution of programming languages you would expect some advancements. Most of the new features focus

on security and making programming easier.

Garbage Collection

When you finish using a resource in a C++ program, you must explicitly tell the computer when to release the memory it was using. This is accomplished with pointers. Since Java does not use pointers for security reasons, it needs a way to clean up resources when they are not needed any more. That's where garbage collection comes in.

Garbage collection is a threaded run-time environment that keeps track of all the parts of your program and automatically de-allocates the memory used when the memory is no longer needed. For example, when you declare a variable, memory is allocated to store its value. The garbage collection engine looks at what scope of the program is seen by this variable. When the program leaves that scope, the memory is cleared.

Lets look at a specific example. Here is a simple **for** loop:

```
for (int x; x < 10; ++x) System.out.println(x);
```

The integer we are using to count to ten is actually declared within the declaration of the loop. As soon as the expression is met and the loop ends, the **x** variable's memory space is cleared and put back into the shared memory pool. This same idea works at all levels of the Java environment.

Security

Security was an issue that the creators of C++ did not have to deal with—they left that up to individual programmers. However, since Java is designed to be a distributed programming language, security is a prime concern. Java includes many features that aid in preventing security problems. The omission of pointers is a key issue that reduces security risks. The functionality you lose is made up for in the robustness of your applications and applets. Now, it's just up to browser creators to develop programs that can't be hacked!

Exceptions

Exceptions are not really new—they were used in C++. However, using them was difficult at best. In Java, exceptions are used heavily. Exceptions are error conditions that are not expected during the course of a standard program run. Situa-

tions like missing files will result in exceptions in Java.

In Java, exceptions are actually part of the language; they have their own classes, interfaces, and so on. In C++, exceptions were more of an add-on that was never fully implemented. Look at Chapter 7 for a detailed look at exceptions.

Strings versus Character Arrays

In C++, strings are simply arrays of characters. In Java, strings can be handled as strings. They are not officially a primitive type but are in fact a class which is instantiated as an object whenever you use strings. So, whenever you handle strings, you are actually handling a **String** object that has its own methods. Instead of calling methods that act upon your string (C++), you are actually calling methods *of* your string object that act upon itself.

If you choose to, you could still use an array of **chars** to act like a string, but you would lose much of the easy functionality built in to the **String** class.

The Super Class

If you have used C++ much, you are familiar with the **this** keyword that is used to reference the current object. Java implements the **this** operator to, but also adds the **super** operator, which tells Java to find the class that our current class extended. You can use **super** to make explicit calls to the superclass of the current class.

New Modifiers

In C++, modifiers are used quite heavily. Java takes many of the C++ modifiers and adds new ones. Most of the new modifiers are needed to help support security issues. Table 1.1 provides a list of the new modifiers:

Table 1.1 Some of the New Java Modifiers

Modifier	Descriptions
abstract	Used to define classes and methods that <i>must</i> be subclassed or overridden to be useful.
synchronized	Tells Java that a method can only be called by one object at a time.
native	Used to create calls to languages like C.
final	Tells Java that a class cannot be subclassed.

The instanceof Operator

The **instanceof** operator is an extremely handy operator to use when you are dealing with objects and you are not sure of their type. You will probably find yourself using this operator most often in conjunction with the Abstract Windows Toolkit (AWT).

Helper Programs

The Java Developer's Kit (JDK) ships with two helpful programs: *javadoc* and *javap*. *javadoc* is an automatic documentation program that creates HTML files automatically to list your classes methods, interfaces, variables, and so on. We discuss this program in greater detail in Chapter 3 so we won't repeat it here. The entire API documentation that shipped with the 1.0 JDK was created using this program. *Javadoc* can only be used with source files.

Another useful utility is *javap*, a disassembler program that prints class signatures. *javap* is used with the compiled class files. When *javap* is run, it outputs a simple listing of a classes public methods and variables. Here is an example:

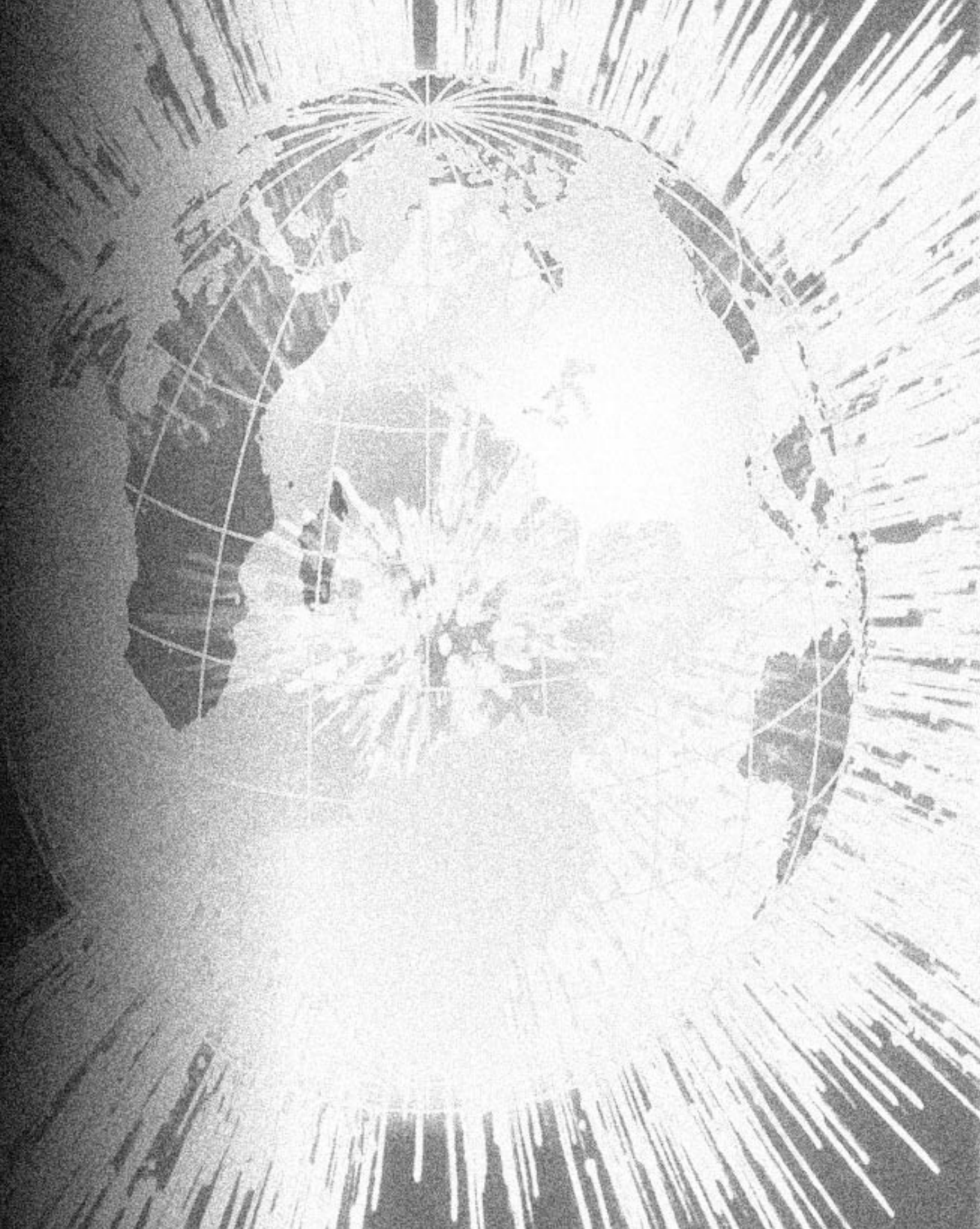
```
Compiled from /home/weisblat/C.java
private class C extends java/lang/Object {
    static int a;
    static int b;
    public static void main(java/lang/String []);
    public C();
    static void ();
}
```

As you can see, this program can be very useful when trying to figure out how to use a class that has little documentation or that you do not have the source for.



2

Writing Your First Java Applet



Writing Your First Java Applet

2

The best way to learn the elements of the Java language is to dive in head first and create a real-world applet.

Before we get into the down and dirty details behind the Java language, let's create a simple program that will introduce you to many of the basic concepts of Java.

Once we decided to put a tutorial program at the beginning of this book, we tried to find one that included many of the major programming elements that you will encounter while coding your own programs. We had to decide if the program would be an application or an applet. Applications and applets are very different things. Creating one over the other is not as simple as changing a couple of lines of code and recompiling. Java applications are free-standing programs; therefore, they must create their own "space" to work within. Java applets, on the other hand, are run from within another program, usually a Web browser. Applets have many parts of their code already written for them and ready to go. For example, if you wanted to display a graphic in a stand-alone Java application, you would first have to create a window to run the program in, a frame, then the graphics you might need. With an applet, most of that work is done for you. A simple call to a graphic method can load an image into your applet's space.

Does that mean that applets are better? Not necessarily, they are simply "different." Both applets and applications have their place in the programming world. The one you use depends on your needs and the needs of the people who will use your programs.

22 Chapter 2

We decided to use an applet as a tutorial mostly because of their emerging popularity. Java applets are popping up on the Web faster than Trekkies show up at a Star Trek convention. The concepts and programming techniques we present as we discuss our applet can be used in any program you create. We'll start out by showing you the code for the applet and then we'll take it apart, piece-by-piece. As we dissect it, you'll begin to see how straightforward the Java language really is.

Introducing the TickerTape Applet

Our first applet is called TickerTape. It scrolls a custom message across the applet space (like a ticker tape). You will be able to specify a couple of parameters, including the text that is displayed and the speed at which the text moves across the screen.

The best part about this applet is that it will introduce you to a number of Java programming language features all at once, including:

- Comments
- Packages
- Classes
- Class Inheritance
- Variables
- Parameters
- Constructors
- Threads
- Overriding Methods
- Graphic Double-Buffering
- Basic Operators
- Interfaces
- Exceptions

And, here is the moment we've been waiting for—the applet itself:

```
// TickerTape Applet

import java.applet.*;
import java.awt.*;
```

```

// TickerTape Class
public class TickerTape extends Applet implements Runnable {
    // Declare Variables
    String inputText;
    String animSpeedString;
    Color color = new Color(255, 255, 255);
    int xpos;
    int fontLength;
    int fontHeight;
    int animSpeed;
    Font font;
    Thread ttapeThread = null;
    Image im;
    Graphics osGraphics;
    boolean suspended = false;

    // Initialize Applet
    public void init(){
        inputText = getParameter("TEXT");
        animSpeedString = getParameter("SPEED");
        animSpeed = Integer.parseInt(animSpeedString);
        im=createImage(size().width, size().height);
        osGraphics = im.getGraphics();
        xpos = size().width;
        fontHeight = 4 * size().height / 5;
        font = new Font("Helvetica", 1, fontHeight);
    }

    // Override Applet Class' paint method
    public void paint(Graphics g){
        paintText(osGraphics);
        g.drawImage(im, 0, 0, null);
    }

    // Draw background and text on buffer image
    public void paintText(Graphics g){
        g.setColor(Color.black);
        g.fillRect(0, 0, size().width, size().height);
        g.clipRect(0, 0, size().width, size().height);
        g.setFont(font);
        g.setColor(color);
        FontMetrics fmetrics = g.getFontMetrics();
        fontLength = fmetrics.stringWidth(inputText);
        fontHeight = fmetrics.getHeight();
        g.drawString(inputText, xpos, size().height - fontHeight / 4);
    }
}

```

24 Chapter 2

```
// Start Applet as thread
public void start(){
    if(ttapeThread == null){
        ttapeThread = new Thread(this);
        ttapeThread.start();
    }
}

// Animate coordinates for drawing text
public void setcoord(){
    xpos = xpos - animSpeed;
    if(xpos <- fontLength){
        xpos = size().width;
    }
}

// Change coordinates and repaint
public void run(){
    while(ttapeThread != null){
        try {Thread.sleep(50);} catch (InterruptedException e){}
        setcoord();
        repaint();
    }
}

// Re-paint when buffer is updated
public void update(Graphics g) {
    paint(g);
}

// Handle mouse clicks
public boolean handleEvent(Event evt) {
    if (evt.id == Event.MOUSE_DOWN) {
        if (suspended) {
            ttapeThread.resume();
        } else {
            ttapeThread.suspend();
        }
        suspended = !suspended;
    }
    return true;
}

// Stop thread then clean up before close
public void stop(){
    if(ttapeThread != null)
```

```

        ttapeThread.stop();
        ttapeThread = null;
    }

} // End TickerTape

```

Before we discuss how the program works, why don't you try it out and see how it looks. You can type the code in for yourself and compile it or head up to the Coriolis Group Web site at <http://coriolis.com> and download this applet as well as all the other code examples listed in this book (look in the "What's Free" section). You can also view this entire chapter on-line by going into the Coriolis books section on the site and searching for this book. There you can choose the "sample chapter" option where you will see this chapter.

Running the Applet

Because our program is set up to be an applet, it cannot run on its own. It needs the help of a Web browser. To run the program from your personal system, you'll need to follow these steps:

1. Use a text editor to type in the applet code. Save the file as TickerTape.java.
2. Compile the program. This will create the file TickerTape.class. To compile the program, you'll need access to the Java Developer's Kit.
3. Move the file into the same directory where you store your HTML files.
4. Create an HTML (HyperText Markup Language) file or edit an existing one, and add the following instructions:

```

<APPLET CODE=TickerTape.class WIDTH=600 HEIGHT=50>
<PARAM NAME=TEXT VALUE="The Java TickerTape Applet...">
<PARAM NAME=SPEED VALUE="4">
</APPLET>

```

Recall that HTML is the language used to create Web pages. Each statement in the language specifies one specific formatting, file processing or hypertext (linking) operation, such as loading and displaying a graphic image, defining a hypertext link, displaying a word or sentence in bold, or loading and playing a Java applet. (We'll look at the HTML instructions for playing our applet in much more detail in a moment.) If you are creating a new HTML file, you might want to name it TTAPE.HTML.

26 Chapter 2

5. Start a Web browser like Netscape 2 that is capable of running Java applets. Then, load in the HTML file you just created. Keep in mind that not all Web browsers can run Java applets. If nothing happens after you load in the HTML file, first check to make sure that you entered the HTML instructions carefully. Then, check to make sure that you are using a Java-playable browser.

After you open the HTML file that includes the required instructions, you should see a screen similar to one shown in Figure 2.1. Once you get the applet to run, you can experiment with it by changing the text and speed parameters. Just replace the strings listed after each **VALUE** statement. For example, if you changed the third HTML statement to be:

```
<PARAM NAME=TEXT VALUE="Buy stock in Java">
```

You would see the string “Buy stock in Java” scrolled across the screen. For the **SPEED** parameter, lower numbers produce slower but smoother animation and higher numbers create faster but sometimes jerky animation.

Even if you have created Web pages using HTML instructions, you might be unfamiliar with the Java applet-specific HTML tags. (If you need to brush up on the general techniques of creating Web pages with HTML, we suggest you get a copy of a good tutorial book such as The Coriolis Group’s *Netscape and*



Figure 2.1

The TickerTape applet in action.

HTML Explorer.) The `<APPLET> ... </APPLET>` tag pair tells a Java-enabled Web browser, such as Netscape 2, that a specified applet should be loaded and played. Notice that in our example, one HTML line does this work for us:

```
<APPLET CODE=TickerTape.class WIDTH=600 HEIGHT=50>
```

The **CODE** parameter specifies the name of the applet—in this case it is the name of our file, *TickerTape.class*. If you used a different filename, you would need to change this instruction. The **WIDTH** and **HEIGHT** parameters specify the width and height of the window or “space” that will be used to play the applet. The dimensions are specified in units of screen pixels. Since our applet needs to simulate a ticker tape-like device, we’ve defined it to be very wide but short. The other HTML instructions, `<PARAM>`, are used to specify the parameters for our applet:

```
<PARAM NAME=TEXT VALUE="The Java TickerTape Applet...">
<PARAM NAME=SPEED VALUE="4">
```

Notice that each parameter has a name and a value. The name must correspond with the name of the parameter used in the Java applet. The **VALUE** clause handles the work of assigning the parameter a default value. Later in this chapter we’ll show you how parameters are processed using special Java functions. If you change the values for either of these parameters and reload the HTML file, you’ll see the effect of the changes immediately.

Where’s the Main Program?

After taking a quick look at our applet, the first question you might have is where the heck is the main program? That is, which code is executed first? If you have experience programming with a language like C/C++ or Pascal, you’re probably looking for a program entry point like this:

```
main() // The starting point for a C++ program
{
    inputText = getParameter("TEXT");
    animSpeedString = getParameter("SPEED");
    animSpeed = Integer.parseInt(animSpeedString);
    im=createImage(size().width, size().height);
    ...
}
```

28 Chapter 2

Don't look too hard because you won't find such a "main function" in a Java applet. Instead, Java applets really are designed to run "inside" another application—in our case the Netscape browser. You can think of an applet as if it were a plug-in or component. This means that the routines and methods in a Java applet are executed by the controlling program (the browser). Let's step through our applet to better understand how this process works.

The TickerTape applet contains a number of functions, which are actually called methods in Java. (This terminology is borrowed directly from C++.) Take a moment to look over the applet and you'll find methods like **init()**, **paintText()**, **start()**, **run()**, **stop()**, and so on. Some of these are standard Java applet methods (they have names and perform operations that are pre-defined); and others are user-defined (we made them up). When the applet runs, the browser running the applet knows which methods are used and in which order to call them. Figure 2.2 shows the order of how the methods are called. Notice, first that the browser calls the **init()** method. Each statement in **init()** executes until the method

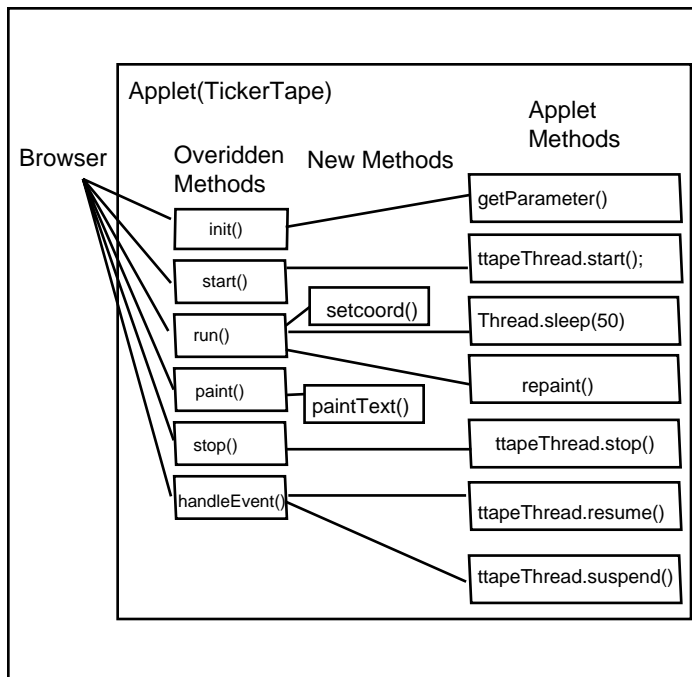


Figure 2.2

How the methods are controlled in the Java applet.

finishes. If you look closely, you'll see that **init()** doesn't call any of the other **TickerTape** class methods. What gives?

To understand what happens next, keep in mind that the browser is the controlling program—not the applet itself. After **init()** is called, the **start()** method is then called by the Web browser. **start()** sets up **TickerTape** as a thread. You'll learn more about threads later on, but for now you can think of a thread as a mechanism for handing off control. If the applet were not set up as a thread, it would run inefficiently and the browser performance would suffer as well. Once the **TickerTape** applet is set up as a thread, the browser will know what to do when events occur such as the mouse button being clicked, a window being moved, and so on.

After **start()** has done its job of setting up the thread, typically the **run()** method would be called next by the browser. This method acts as a loop that keeps the program in motion—in other words, it drives the operations to make our text move across the screen. Table 2.1 provides a summary of each of the key methods used in the **TickerTape** applet. The methods like **paint()**, which are standard Java applet methods, can be redefined to perform different operations. For ex-

Table 2.1 Methods Used in the **TickerTape** Applet

Method	Description
init()	A standard Java applet method that is used to initialize variables and objects defined for the TickerTape applet.
paint()	A standard Java applet method that is called whenever a browser has recognized that something has changed, such as a window being moved or resized, text being drawn on a window, and so on.
paintText()	A user defined method that refreshes text for the ticker tape in a buffer.
start()	A standard Java applet method that turns the applet into a thread.
setcoord()	A user defined method that is called by the run() method at every program cycle to update the position of the text.
run()	A standard Java applet method that serves as the heart of the program. Without the run() method, the applet would not perform any actions.
update()	A standard Java applet method that calls the paint() method to update the screen.
handleEvent()	A standard Java applet method that process mouse activity.
stop()	A standard Java applet method that stops the thread, which in turn terminates the execution of the applet.

30 Chapter 2

ample, in our TickerTape applet we redefine **paint()** to process updates to the screen in a more efficient way. We could add additional features (code) to this method to handle other tasks such as adding a border around the ticker tape, adding a frame counter, and so on.

Introducing Java Comments

Like any good programming language, Java allows you to include comments along with your programming statements. You can see Java's connection to C/C++ right off, since it supports both the C and C++ comment styles. For example:

```
int i; /* This is a C-style comment */
```

```
int i; // This is a C++-style comment
```

Java also provides a new type of comment syntax that can be used to automatically generate formatted documentation. This new syntax looks like this:

```
/** Documentation comment. Comments listed between these symbols will be  
used to automatically create documentation. */
```

Back to our program, the first line of the code is actually just a comment that tells us about the program:

```
// TickerTape Applet
```

Although Java allows us to use any or all of these styles within a single program, we will only use the `//` notation in our TickerTape program. After all, we don't want to make things more complicated than they need to be.

What's in a Package?

The next two lines of our applet are used to reference *packages* that contain *classes* that contain the *methods* we wish to use. This may sound like a mouthful, but the concepts involved are actually quite simple. Here's the code in question:

```
import java.applet.*;  
import java.awt.*;
```

Packages are used to group related classes for use in other programs. This is a direct extension of the object-oriented techniques that languages like C++ provide for programmers.

There are several class packages that come with Java. Table 2.2 provides the current set of them.

By default, every Java application imports the classes contained within the `java.lang` package, so you do not have to manually import this package.



The Import Statement for C Users

Using the **import** statement to include packages is similar in concept to using the **include** statement in C to include header files.

If you look closely at our **import** statements in the TickerTape program, you'll notice that the `*` character is used. This character tells the Java byte-code compiler to use all the classes stored within the package. You could also specify which classes to use; but since you usually need many classes within a single package, it is easier to simply use the asterisk. Also, the Java compiler is smart enough to figure out which classes are used and which ones aren't so that using the asterisk does not eat up any additional memory.

In our program we import the **applet** package because we are creating an applet. We also need to import the **awt** package because we want to use its graphics

Table 2.2 Standard Java Classes

Java Class	Description
<code>java.lang</code>	Contains essential Java classes.
<code>java.io</code>	Contains classes used to perform input/output to different sources.
<code>java.util</code>	Contains utility classes for items such as tables and vectors.
<code>java.net</code>	Contains classes that aid in connecting over networks. These can be used in conjunction with <code>java.io</code> to read and write information to files over a network.
<code>java.awt</code>	Contains classes that let you write platform-independent graphic applications. It includes classes for creating buttons, panels, text boxes, and so on.
<code>java.applet</code>	Contains classes that let you create Java applets that will run within Java-enabled browsers.

32 Chapter 2

capabilities. This package contains the classes we need so that we can display our ticker tape-like graphics.

The AWT package was developed to aid in creating windowed applications and applets. It does for Java what Visual C++ does for C. Instead of having to manually define graphical user elements like buttons, windows, and menus, and then manually having to write code to handle mouse events, the AWT package takes care of it for you.

The Mystery of the AWT Package

Now that you know a bit about the Java programming language, it's time for a little quiz. What does AWT stand for?

- A. Another Window Toolkit
- B. Abstract Window Types
- C. Abstract Window Toolkit
- D. Advanced Window Toolkit
- E. Abstract Windowing Toolkit

The answer is C. (If you guessed right, you may have a future in Java programming after all.) According to the official AWT tutorial, this acronym stands for the *Abstract Window Toolkit*. (Another Window Toolkit came in a close second.) We think all the extra names came about because the name got passed on from programmer to programmer without the aid of any official documentation. As people referred to the package, using different names, no one knew who was correct anymore.

Classes, Inheritance, and Interfaces

If you have done any programming in C++, you already know how important classes are. Java is no exception. Most of the Java programming work you will be doing involves writing classes from scratch and deriving more powerful classes from your existing classes.

To see how classes are defined in Java, let's return to our TickerTape program. After the two key packages have been included, we define our first class:

```
// TickerTape Class
public class TickerTape extends Applet implements Runnable {
    // Declare Variables
    String inputText;
    String animSpeedString;
    Color color = new Color(255, 255, 255);
    int xpos;
    ...
}
```

We're not showing the complete class here but it contains the following components:

- Definition
- Variables
- Methods

The first line of code that actually sets up the class definition is illustrated in Figure 2.3. Let's take a close look at each section.

Class Modifier A class modifier tells the Java compiler how and where a class can be used. The two main types of classes are called *public* and *private*. A **public** class can be accessed from other packages, either directly or by using an **import** statement. If you omit the **public** modifier at the beginning of the class definition, the class would become private and use of the class would be limited to the package in which it is declared.

The two other modifiers that can be used to define classes are **abstract** and **final**. We'll cover these class modifiers in detail in Chapter 4.

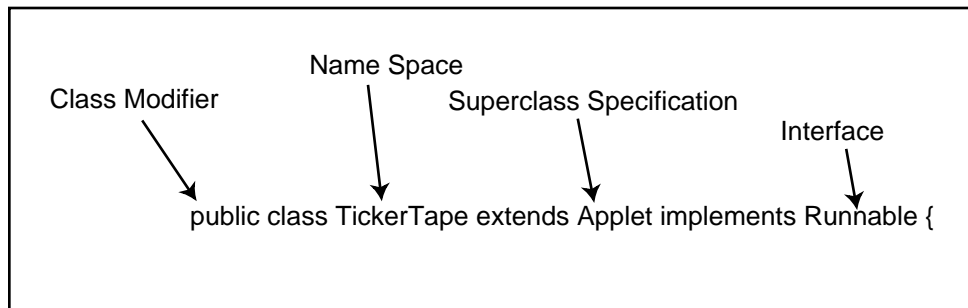


Figure 2.3

Setting up a class definition in Java.

34 Chapter 2

Name Space The name space in a class declaration is simply the name of the class. In our case the name space is “TickerTape.”

Superclass The keyword **extends** indicates that we are inheriting all of the methods, variables, and field declarations from the **Applet** class. **Applet** becomes the superclass of our TickerTape class. This means that we can use any of the methods and variables from the **Applet** class.

If we did not include the superclass specifier, the program would derive itself from the **Object** class by default.



Applet Package vs. Applet Class

Don't get confused by the **applet** *package* and the **Applet** *class*. At the beginning of the program we imported the **applet** package with the **import** command. This gave us access to all the classes within the package, which in turn means that we can then subclass the **Applet** class.

The **applet** package has other classes in it that help in creating applets. The **Applet** class has methods in it that we *must* use in all applets.

Interface An *interface* is a collection of method declarations, but without implementations. An interface simply sets up a template that all classes that use it must follow. For instance, if we set up an interface that has two methods, **start** and **stop**, then any class that implements that interface must have **start** and **stop** methods within it.

The interface **Runnable** is used here via the **implements** keyword. Interfaces solves some of the same problems that are solved by multiple-inheritance in C++. You can implement many interfaces if you want. Here's an example:

```
TickerTape extends Applet implements Runnable, Stoppable, Pausable
```

Once again, you *must* implement every method in the interface you are using. Interface **Runnable** contains only the method **run()**.

Types, Objects, and Constructors

Now that we have declared the class for our TickerTape applet we need to set up a few variables that we will need to store various strings, numbers, dimensions, and so on. Table 2.3 lists the basic types supported by the Java language.

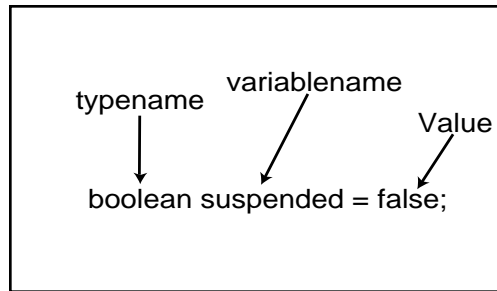
Let's now use a few of these data types to declare our variables. These variables will be the ones that are used throughout our class, so we place them directly after the class declaration as shown here:

```
public class TickerTape extends Applet implements Runnable {
    String inputText;
    String animSpeedString;
    Color color = new Color(255, 255, 255);
    int xpos;
    int fontLength;
    int fontHeight;
    int animSpeed;
    Font font;
    Thread ttapeThread = null;
    Image im;
    Graphics osGraphics;
    boolean suspended = false;
```

Let's take a few of these variable declarations and break them down into their components. As shown in Figure 2.4, a standard variable declaration consists of a data type, variable name, and optional value.

Table 2.3 The Basic Data Types Supported by Java

Data Type	Description
boolean	A true or false value. You cannot convert between booleans and any other basic types.
byte	8-bit signed value
short	16-bit signed value
char	16-bit unicode character
int	32-bit signed value
float	32-bit IEEE754 floating-point
double	64-bit IEEE754 floating-point
long	64-bit signed value

**Figure 2.4**

Standard Java variable declaration.

The *typename* in a standard declaration needs to be one of the basic Java types listed in Table 2.3. One potential problem to watch out for here is capitalization. Spelling `boolean` with a capital B can really cause some errors. With the current state of Java debuggers, don't expect them to help much.

The *variableName* can be any ASCII string. You can even make the *variablename* the same as the class name. Be careful though; this can get very confusing. Variable names are also case-sensitive. This means the variable `String1` is different from the variable `string1`. You also cannot include spaces or any other whitespace characters in your variable names.

When you declare a variable, you can also assign it a value at the same time. This is a very useful feature that was introduced with the C language, which Java heavily steals from. You can even perform a calculation or call a method to obtain values while you are declaring a variable. However, the value sections of any declaration do not need to be set when they are declared—they can be given values later in your program. Here are some examples:

```
// declare a variable and assign a value
int xpos = 10;

// use a calculation in a declaration
int fontLength = 30 + xpos;

// call a method to obtain a value during a declaration
int fontHeight = getValue();
```

Variables can also use modifiers like classes do. However, there are several more modifiers for variables, including **static**, **final**, **transient**, and **volatile**. We'll cover all of the variable modifiers in detail in Chapter 4.

VARIABLE DECLARATIONS USING A CONSTRUCTOR

If you look closely at the variable declarations in the TickerTape class, you'll see that most of them are quite simple. But there is one that looks a little different:

```
Color color = new Color(255, 255, 255);
```

In this declaration, a device called a constructor is used. The constructor actually serves as a special type of method that is responsible for initializing new objects. Constructors are used to create custom and/or complex objects other than the basic types. In this case, the **Color** constructor (a method in the AWT package) is passed the three integers (255,255,255). Then a value "java.awt.Color[r=255,g=255,b=255]" is returned to the variable **color**. The actual components used in this type of declaration are illustrated in Figure 2.5.

This constructor initializes an instance of the variable **color** that represents a color by its RGB values.

Thank Goodness for Garbage Collection

One of the biggest headaches in creating large programs with languages like C or C++ is keeping track of resources and disposing of them when they are not needed. An unruly C program that does not properly clean up after itself can quickly eat up a lot of memory (and a lot of your time trying to debug it). For example, if you write a C++ program that allocates a big block of memory for a dynamic data structure, but you forget to release the memory after the data structure is no longer used, your program could make a mess of things. Or if you use pointers in a program and you don't allocate, access, manage, or release them properly, you could end up spending a number of late nights debugging your code. The trouble with these type of resource allocation problems is that they

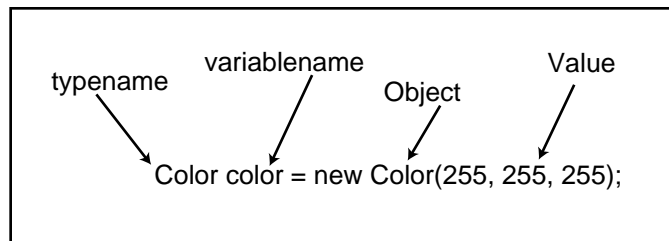


Figure 2.5

Using a constructor in a variable declaration.

are very difficult to detect. Anyone who is active in the software development industry is keenly aware of intermittent errors in their software that can cause their release dates to slip. Every year millions of dollars are spent and lost because of extensive software testing that is required to find the errors caused by troublesome pointers and mismanaged memory allocation.

So is there is a solution to this crippling problem? The answer is *garbage collection*. Garbage collection is not a new invention, recently created for Java programmers. It has been around for years; in fact, programmers who have used languages like Lisp, Smalltalk, and Prolog, have been writing amazing programs that really push this technology to the limit. The basic idea behind garbage collection is to offload the work of managing memory and other resources to the program itself. If memory is needed for a new data structure or object, the program automatically takes care of this task and automatically releases the resources when they are no longer needed. This frees the programmer from a number of complex tasks, such as declaring pointers to access memory, passing pointers as arguments to functions, setting up memory buffers to swap the contents of data structures, and so on.

Of course there is a price to pay for all of this convenience. Programming languages that use garbage collection tend to be bigger and run slower than programs written in “the programmer does it all by hand” languages such as C and C++.

In developing Java, the language designers at Sun wanted to make it as flexible yet robust as possible. They reasoned that a smart compiler, even for a language that has a syntax like C++, could figure out for itself which program elements require memory allocation and perform memory management operations automatically. In Java, memory taken up by objects, methods, and variables is allocated and then cleared when those items are no longer needed. This garbage collection was designed not only to make life easier for programmers but because it is required for creating programs that can run on many different platforms and allocate memory in the same way.

Using Methods

Now we’re ready to get into the meat of our program by exploring the implementation section of the applet:

```
public void init(){  
    inputText = getParameter("TEXT");
```

```

    animSpeedString = getParameter("SPEED");
    animSpeed = Integer.parseInt(animSpeedString);
    im=createImage(size().width, size().height);
    osGraphics = im.getGraphics();
    xpos = size().width;
    fontHeight = 4 * size().height / 5;
    font = new Font("Helvetica", 1, fontHeight);
}

```

We learned earlier that this code actually shows the definition of what is called a *method*. (We'll be looking at methods in more detail a little later.) The method defined here is named **init()**. As we discussed earlier, it is the starting point for all applets. *Remember that it is called by the browser whenever the applet is first loaded.* You can think of this method as serving a similar role to the one carried out by a **main()** function in a C program. When the applet is first loaded into a runtime environment like a Web browser, the program execution will begin with the first statement in the **init()** method.

The **public** modifier in front of this method tells us that this method may be called from any object, and the **void** modifier states that the applet will not return any values.

Init() performs a number of tasks. First, it loads in the two parameters used by the applet. Then, it calculates the animation speed for the ticker tape using the **SPEED** parameter. The remaining statements in this method are needed to set up variables to support the graphics and text fonts used in the applet. Let's look at all of this code in a little more detail.

PROCESSING PARAMETERS

Many times when you create applets you will want the user to be able to specify options such as font size or animation speed. If we were creating an application, instead of an applet, these values could be passed as command line arguments. Applets do not have command line arguments, but they do have parameters. As we introduced earlier, parameters are embedded in HTML tags that reside between the opening and closing **<APPLET>** tags like this:

```

<APPLET CODE=TickerTape.class Width=600 Height=50>
<PARAM NAME=TEXT VALUE="The Java TickerTape Applet...">
<PARAM NAME=SPEED VALUE="4">
</APPLET>

```

40 Chapter 2

The idea behind parameters is very similar to arguments, but parameters allow you a little more flexibility. Since parameters have a name associated with them, they can be put in any order. It is also easier to determine if a parameter has been left out and if so, which one.

When a `getParameter()` method executes in an applet, the method searches a parameter list to locate a parameter with the corresponding `NAME` field. In our applet, we need to read in the text that will be displayed in the applet and a speed setting that effects how fast the text scrolls across the screen. Thus, notice that the `init()` method contains two calls to `getParameter()`:

```
inputText = getParameter("TEXT");
animSpeedString = getParameter("SPEED");
```

After these methods are called, the variable `inputText` will contain the string “The Java TickerTape Applet...” and `animSpeedString` will contain the value “4.” But there is one catch: *All parameters must be read in as strings*. Since the variable we declared for animation speed `animSpeed` in the `TickerTape` class is an integer

```
int animSpeed;
```

we need to convert the speed parameter from a string to its integer representation. To perform the conversion, we use the `parseInt()` method of the `Integer` class that resides in the `java.lang` package. (Make sure that you use the correct capitalization here, or you will get errors when you try and compile the program.) The code that performs this operation looks like this:

```
animSpeed = Integer.parseInt(animSpeedString);
```

Notice that the syntax for calling this method involves using the name of the `Integer` class. The `parseInt()` method takes a string as its argument and returns a 32-bit signed integer. For larger numbers you could use the `parseLong()` method that can return a 64-bit signed integer. The value of the converted string, 4, is stored in the variable `animSpeed`—right where we want it.

COMPLETING THE INITIALIZATIONS

To code our applet so that it provides smooth animation, we use a popular programming trick called *buffering*. The idea is that we don’t want to write every pixel on the screen as changes occur. With buffering, we send data to a hold-

ing area that is constructed until we are ready to update the screen. Then, we blast the entire contents of the image buffer to the screen at once. To understand how this process works in more detail, see the sidebar *Double Buffering*.

Fortunately, an image buffer is easy to set in a Java applet. In fact, we only need a few lines of code. The following two lines in `init()` perform the work of setting up the image buffer component that will store all the changes we make to the graphics until we want to blast it onto the screen:

```
im=createImage(size().width, size().height);
osGraphics = im.getGraphics();
```

This code creates a graphics object called **im** with a width and height equal to the size of the applet. We use the `size().width` and `size().height` objects to return information about the applet's client space. In this case, `size().width` and `size().height` will always be equal to the width and height values we used when we called the applet in our HTML file:

```
<APPLET CODE=TickerTape.class Width=600 Height=50>
```

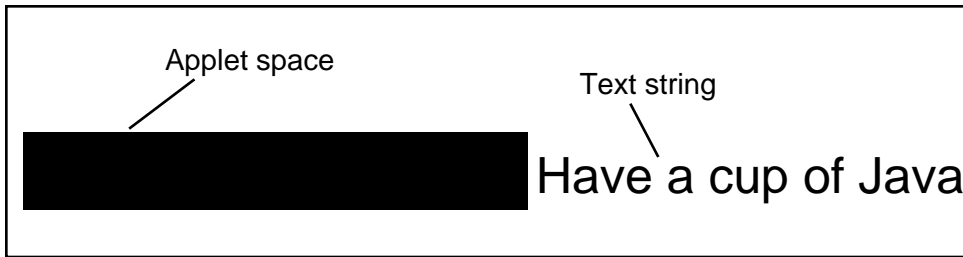
Our final bit of business for creating the buffer is to use the `getGraphics()` method to initialize the graphics and clear the buffer.

The next two lines of code in `init()` are used to set the values of the **xpos** and **fontHeight** variables.

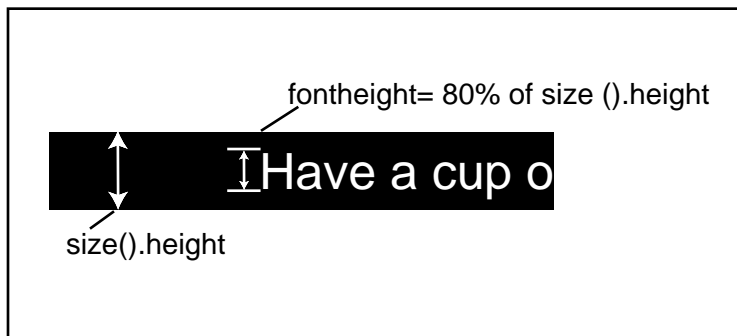
```
xpos = size().width;
fontHeight = 4 * size().height / 5;
```

The **xpos** variable will be used to track the current position of the left side of the text. We start the applet with **xpos** equal to the width of the applet (`size().width`) so that as it starts, the text begins scrolling from off of the applet. Figure 2.6 shows how this process works for scrolling the text to the left.

The **fontHeight** variable is used to store the height of the text. In this case, we are setting the height of the text to be 80% of the height of the applet. By doing this, we allow the HTML programmer to change the size of the applet and have the size of the text reflect that change. You could also use another parameter to input the size of the text, but our method is simpler and saves a step or two. Figure 2.7 shows you how all the applet and text sizes correspond with each other.

**Figure 2.6**

Starting the text off of the visible part of the applet.

**Figure 2.7**

Coordinates in our applet.

The final line of code in `init()` is used to initialize our **font** object:

```
font = new Font("Helvetica", 1, fontHeight);
```

Recall that we already created this variable in our declaration section at the beginning of the applet (the **TickerTape** class) but we gave it no value. Here we are setting up the name of the font (Helvetica), the style (sum of constants **PLAIN**, **BOLD**, and/or **ITALIC**), and the size (points or pixels).

Notice also that we are using a constructor again. Recall that the **new** statement tells Java that we want a “new” object, in this case with all the characteristics of the standard **Font** class.



Using Fonts with Java Applets

Here is something you should keep in mind when using different fonts with your Java applets: *Not all fonts will work with Java*. Be careful when you choose a font name that it is a standard font. You may have strange fonts on your system that are not available to everyone else, so choose them wisely. If your applet requests a font that is not present on the user's system, it will use a default font.

Methods and Method Overriding

As we've seen, methods are the object-oriented equivalent to functions in C. They are much more powerful, however, because they allow you to access the internal components of an object. For example, in our TickerTape applet, we use methods like `init()`, `paint()`, and `paintText()` to access some of the key data elements like `animSpeed` (the animation speed for a ticker tape), `font` (the font used to display text), and `xpos` (the position of the ticker tape text). Because of the object-oriented nature of the Java language, we are able to organize our programs in such a way that details can be hidden away and protected from being accessed by only those parts of the program that need to have access. In the world of object-oriented programming, this technique is called *encapsulation*.

But hiding details and controlling access is only part of the story. The real power of object-oriented programming comes from the ability to take existing code and derive new code from it. Most programmers spend way too much time writing the same functions or routines over and over again, changing them ever so slightly so that they can be used in different applications.

To take advantage of the potential of object-oriented programming, Java, like its C++ counterpart, supports a concept called *method overriding*. The idea behind method overriding is that you can take an existing method and derive a new one from it. The new method would have the same name as the original but its behavior—the actions it performs—could be entirely different.

As you've learned already, Java applets provide a number of methods that are predefined for you. Some of these include `init()`, `paint()`, `start()`, and `stop()`. When you create an applet, it becomes your job to decide which methods to override and how to make them do what you need.

Method Overriding with Classes

The technique of method overriding has quite an impact on how you use classes in Java to derive other classes. Whenever you “subclass” a class, you also have access to all the original class’ methods. Many times though, the whole point of subclassing is to change how the class interacts or responds to certain events. For example, let’s say you created a class to process mouse events. In particular, you could define a class that responds to a mouse click by performing the action of playing a sound, among other things. Now, maybe you want another class that will perform another action, such as displaying a graphic, when a mouse button is clicked. You could either create a new class and duplicate your work or use the sound class as a base class and simply use method overriding to create a new method to respond to the user input.

In the new class, all you have to do is create a method with the same name as the corresponding method in the superclass. Of course, you must change its behavior (how it reacts to the user input—displaying a graphic instead of playing sound).

When working with applets, you will be doing a lot of method overriding because the **Applet** class that you extend already has many methods built into it. Many of these methods perform little, if anything; but they need to be there to capture all the possible calls the browser may send.

INTRODUCING THE `PAINT()` METHOD

Now that you know a little about method overriding, we are ready to dig in and look at the `paint()` method. This method is called by the browser whenever the browser thinks something needs to be repainted. Events that might trigger the `paint()` method include displaying text or graphics, re-sizing a component, and so on. Since we don’t “write” directly to our applet, it will never initiate the `paint()` method on its own; so we will call it later. We use method overriding to create our own `paint()` method that will handle all the painting chores that would usually be handled automatically by Java.

For our applet we only need two calls in the `paint()` method:

```
public void paint(Graphics g){
    paintText(osGraphics);
    g.drawImage(im, 0, 0, null);
}
```

The declaration of the argument **Graphics g** may look a little strange. It sets up the applet background to be printed to. (Java will set up the device context of the applet itself as **g**.) The first line of this method calls another method, **paintText()**, which prints the text onto the buffer image. The **paintText()** method is a *user-defined* method that takes a single argument. In this case that argument is the graphics object we want the text drawn onto—the **osGraphics** object.

Finally, we call the **drawImage()** method that copies the buffer data (**im**) onto the applet space on the browser (**g**) for the user to see.

Double Buffering

Double buffering is an extremely powerful concept that has helped make some of today's flicker-free animation and game play possible. It reduces flicker by performing all the graphics functions on a hidden image that resides in memory instead of drawing directly to the screen. Then, the entire image is displayed all at once instead of having to update the screen every time a new bit of graphics is drawn.

If we removed the double buffering technique from our applet, you would see the screen flicker every time the text moved even a little bit. (As an experiment, you might want to try changing the applet code so that the double buffering gets disabled.) If we were drawing any extra graphics or additional text, you would also see flickers for each of those events. These flashes and flickers occur because the screen is updated multiple times during the drawing of the object and the Applet class' **paint()** method clears the screen before it redraws it. With text, the screen can sometimes be redrawn for each letter! This causes the flicker and can actually slow things down if you are drawing many items.

Double buffering improves performance because the graphics can be drawn into memory faster than they can be drawn onto the screen. Even with the extra step of blasting the graphics to the screen, double buffering is still much faster. Figure 2.8 illus-

trates how double-buffering speeds display by reducing calls to the **paint()** method.

The double buffering technique we used in our TickerTape applet is very general, and you can apply it to many of the applets that you write that need to perform flicker free animation. We suggest you experiment with these concepts—you may come up some of your own for writing optimized applets.

Graphic Methods

Now that we are aware of the basics involved in overriding methods, let's return to our applet and explore the other methods used to perform all of the graphics drawing operations. Our next step is to see how the applet will print our text onto the image buffer. This work is accomplished by the user-defined `paintText()` method:

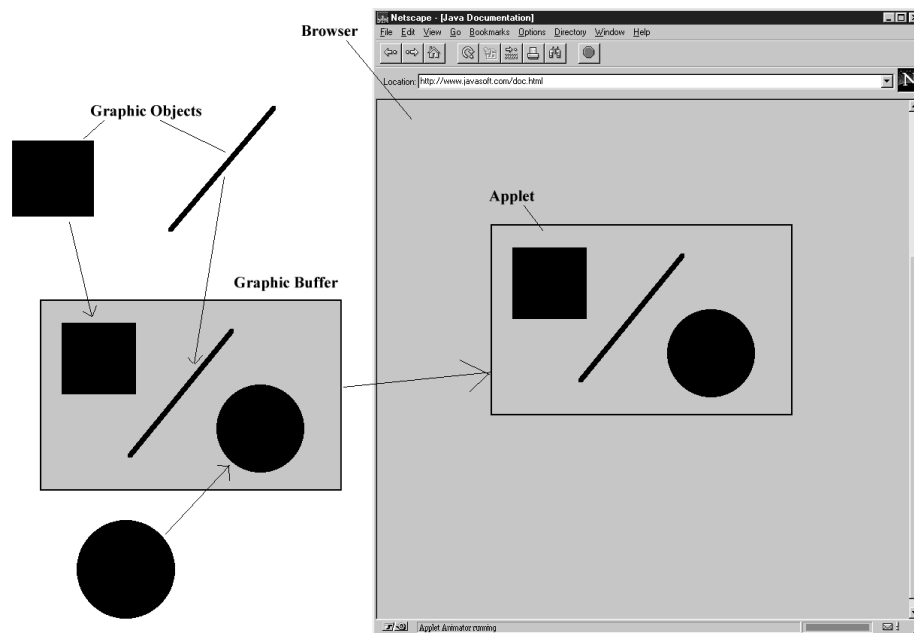


Figure 2.8

Double buffering the applet display to reduce flicker.

```

public void paintText(Graphics g){
    g.setColor(Color.black);
    g.fillRect(0, 0, size().width, size().height);
    g.clipRect(0, 0, size().width, size().height);
    g.setFont(font);
    g.setColor(color);
    FontMetrics fmetrics = g.getFontMetrics();
    fontLength = fmetrics.stringWidth(inputText);
    fontHeight = fmetrics.getHeight();
    g.drawString(inputText, xpos, size().height - fontHeight / 4);
}

```

First, keep in mind that **paintText()** is a method that we created from scratch. In other words, we did not create this method by overriding one that already exists with Java.

This set of method calls found in **paintText()** start by setting the current pen color to black by using the **setColor()** method. Then, we call the **fillRect()** method to draw a filled rectangle that fills the applet. Next, comes the **clipRect()** method that tells Java to clip any data or graphics that are written outside the given boundaries, which in this case is the same as the size of the black rectangle. This set of initializations is illustrated in Figure 2.9.

Next we set the font of the graphics buffer equal to the font we set up in the **init()** method. We also need to change the pen color to something other than black so that our text shows up. For this task, we use the **color** object we set up earlier.

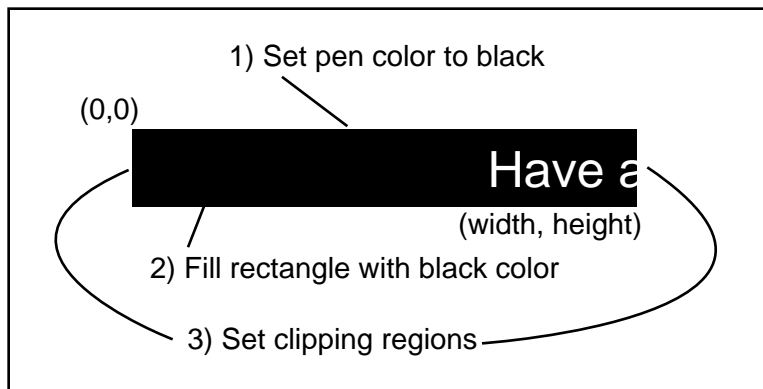


Figure 2.9

Setting up the rectangle for displaying the ticker tape text.

48 Chapter 2

```
g.setFont(font);  
g.setColor(color);
```

Now that we are set to print, we need to determine where to print. Recall that we previously initialized a variable named **xpos** to keep track of the text position. Thus, we can use this variable to tell us where to begin printing. Now we need to find out how tall and long the text we want to print is. We will use this data to center the text vertically and to tell us how long the text is so we can reset the location of the text when it is done scrolling.

To accomplish these tasks, we use the **FontMetrics** constructor. Using font metrics is an easy way to gather information about the physical characteristics of text as it is related to certain components at certain font sizes, types, and so on:

```
FontMetrics fmetrics = g.getFontMetrics();  
fontLength = fmetrics.stringWidth(inputText);  
fontHeight = fmetrics.getHeight();
```

To actually print the text onto the graphics buffer, we use the **drawString()** method. This method takes several arguments as shown:

```
g.drawString(inputText, xpos, size().height - fontHeight / 4);
```

First, we must tell **drawString()** what string we want to print. In this case, **inputText**. Next, we tell it where to start printing along the x-coordinate. Finally we send it the vertical component to tell it where the bottom of the text should be. Figure 2.10 illustrates how **drawString()** sets up the required components. Here, we want the height to be a quarter of the difference of the height of the applet and the height of the text. This does a nice job of centering at any applet size.

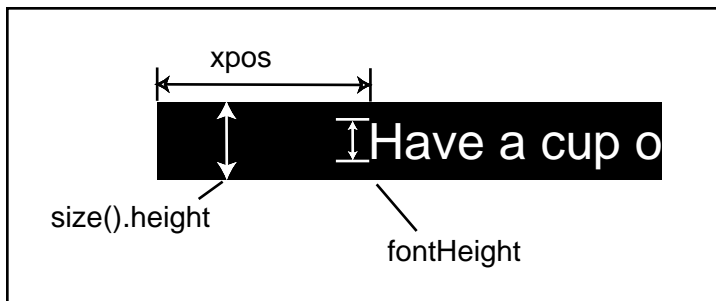


Figure 2.10

Placing text in the buffer with **drawString()**.

In looking over the call to the `drawString()` method, don't get confused by the use of `g` as the name of our graphic object. In the `paint()` method we used the `g` variable to reference the applet's device context. Here you should notice that `g` is referencing the `osGraphics` object we created to act as an image buffer. This is an example of how scoping works with variables in Java. Scoping is basically the same as it is in other languages but we wanted to make sure you were aware of what is going on here. Since the `g` variable is not declared anywhere outside the methods, but simply declared within the method declaration as an argument, it is available only to the method where it was created.

Working with Threads

Now we get to a complex but powerful part of Java—*threads*. As we mentioned earlier, a thread is a special type of process that is used by a Java interpreter to control how a particular applet is executed. Fortunately, our TickerTape applet only needs a single thread to run smoothly, so our job is easy here. In Chapter 8 we will cover threads in much more detail, but for now we need to cover some basics so that you can follow the magic that is occurring in our TickerTape applet.

Here are the three methods that are responsible for handling our thread:

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

public void run(){
    while(ttapeThread != null){
        try {Thread.sleep(50);} catch (InterruptedException e){}
        setcoord();
        repaint();
    }
}

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

50 Chapter 2

As you might have guessed, each of these methods is predefined in Java and they have been overridden by our applet. The **start()** method is called after the **init()** method finishes; **start()** actually belongs to the applet. It is called whenever the applet is started, such as when the Web page the applet is assigned to is first loaded and every time the page is referenced again from a Web browser.

The **start()** method checks to see if the thread for the applet has been created by asking if the thread is equal to a null value. (A null value indicates that it has not been created.) If the thread has not been created, Java creates a new thread using the **this** keyword to tell Java to turn the current applet class we are running into a thread. Then, **start()** calls the **start()** method of the thread to initiate its execution. We know this sounds weird! But make sure you understand that this **start()** method is different than the **start()** method of the applet. Otherwise, you might think this call will create an infinite loop.

The **run()** thread method is called repeatedly as the thread runs. It is the only method that we were required to have because of the use of the **Runnable** interface:

```
public void run(){
    while(ttapeThread != null){
        try {Thread.sleep(50);} catch (InterruptedException e){}
        setcoord();
        repaint();
    }
}
```

We use the **sleep()** method to pause the thread for a few milliseconds to slow it down a little. This gives us a refresh rate of ten frames per second maximum. It also gives the thread time to check for mouse clicks.

We then increment the **xpos** object to facilitate animation by calling the user-defined **setcoord()** method. Here we also need to stop and check to see if the text has gone completely off the left edge of the applet. We do this by checking to see if **xpos** is less than negative **fontLength**:

```
public void setcoord(){
    xpos = xpos - animSpeed;
    if(xpos < -fontLength){
        xpos = size().width;
    }
}
```

Finally, we come to the **stop()** thread method, which is called whenever someone leaves the current Web page the applet is assigned to or otherwise closes the applet.

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

In **stop()**, we check again to see if the thread is equal to **null**. If the thread is active, we stop it using the thread's **stop()** method. Then, we kill the thread by setting it equal to **null**. You may think that garbage collection would take care of killing the thread for us, but this is one case where you would not want to rely on garbage collection. Why? There may be situations where you want to keep a thread active as people move from page to page in the browser (of course, closing the browser will kill all applet threads).

At this point, the applet is ready to run. However, it would be nice to be able to start and stop the TickerTape with a mouse click, so let's add support for user input.

Threading Java Applets

Trying to understand, not to mention program, multitasking and threaded applications is tricky until you get a good grasp of the basics. One good place to start is to try to understand why threading is required in the first place.

Making your applets use threads is extremely important. If you don't use them and your applets are not very "friendly," the browser's performance can suffer dramatically. What causes this? Current browsers like Netscape 2.0 give applets as much of the system resources it can. So, if you create an applet that uses a loop to control its operations, the applet would suck up all the processing time the browser can give it. The net effect is that the browser becomes very unresponsive.

If you are creating a sample applet that only performs a single operation and requires little or no interaction, you may not need to use threading. For example, assume you are creating an applet that simply plays a sound when you click on something. When

this applet is not playing a sound, it is not doing anything other than waiting. Thus, it is not using resources and it does not need to be a thread. However, if the same program had a loop that checked to see when the sound stopped playing, then displayed a message, the loop used to wait for the end of the sound could use considerable resources no matter how simple it is. Instead of using a controlling loop, you would want to set up the applet as a thread—just as we’ve done with the TickerTape applet.

In setting up threads the method you will use most is **run()**. This method is called by the browser every cycle. If you have multiple applets or multiple classes, each with its own **run()** method, they will all be called at the same time.

How does the browser know to call the **run()** method? That’s where interfaces come in. If you want the **run()** method to be called by the browser then you *must* implement the **Runnable** interface, as we’ve done with our TickerTape applet. By doing this, the browser can query the applet to see if it has implemented the **Runnable** interface. If it has, the browser knows it can call the **run()** method of the applet.

Processing User Input

Allowing people to gain control over a program is a very powerful means of interacting with users. Simply being able to start and stop the TickerTape applet is enough to give people the feeling of interactivity rather than passive viewing.

Here is the code that adds user interaction to our applet:

```
public boolean handleEvent(Event evt) {
    if (evt.id == Event.MOUSE_DOWN) {
        if (suspended) {
            ttapeThread.resume();
        } else {
            ttapeThread.suspend();
        }
        suspended = !suspended;
    }
    return true;
}
```

This `handleEvent()` method can be used for everything from mouse clicks to key presses, to drag and drop functions. It is automatically called by the browser whenever it senses that the user is trying to interact with the applet. The argument `evt` is the crucial part of this method. It tells us what event has occurred and allows us to react accordingly.

Since we are filtering for all mouse clicks, we simply use an `if..then` statement to check and see if the `evt` argument ever equals `Event.MOUSE_DOWN`. When it does, we know that a button has been pressed. Since Java is a cross-platform language, we do not have the ability to determine which button was pressed, just that one was indeed pressed. If you are a Windows programmer only, it may be possible to create code in C that detects the other mouse button clicks and then passes that on to a Java program, but that's more than we can get into here.

When we receive the word that a button has been pressed, we check to see if the `TickerTape` is in motion (`suspended` is `False` or `True`). If it is not in motion, we start it by calling the thread's `suspend()` method. If it is already suspended, we use the thread's `resume()` method to start it back up again. Finally, we switch the `suspended` Boolean object to be the opposite of what it was before the mouse button was pressed.

One Last Thing

If this is your first time working with Java, we need to fill you in on how to compile your applet. The process of compiling takes your source code (`.java` file) and turns it into bytecodes (`.class` file). The bytecodes are an interim form of the code that can be read by many different operating systems. The bytecodes will then be used by the Java Virtual Machine (VM) built in to the Web browser that actually interprets the code and runs the program.

The Java compiler is activated by executing the `JAVAC` program. You also need to supply a few arguments including the name of the file you are compiling and whether or not you want to use the debugger.

Here are a few different compile commands that will all work for the `TickerTape` applet:

```
javac TickerTape      // Standard compile
javac TickerTape -g   // Compile with debugging information on
javac TickerTape -d c:\java // Compiles the file to the c:\java directory
                        // Overrides your default classpath
javac TickerTape -classpath .;c:\java\classes
```


If you do not have the Java Development Kit (JDK), check the Javasoft site (<http://www.javasoft.com>). Read the online instructions to learn how to install the JDK. Check out the resource guide from Appendix A for more information.

That's It—Run It

Go ahead and compile the complete program, then run it. How does it look? Try changing the parameters and the applet size. Does the text scale to the height of the applet? If something does not work correctly go back and verify that all your code is correct, recompile, and run the applet again. Make sure that when you recompile an applet that you restart your browser. Many browsers, including Netscape 2.0, do not reload Java applets when you hit the reload button. They *do* however reread the HTML file. So, if you only made changes to the parameters or size of the applet then you do not need to restart the browser.

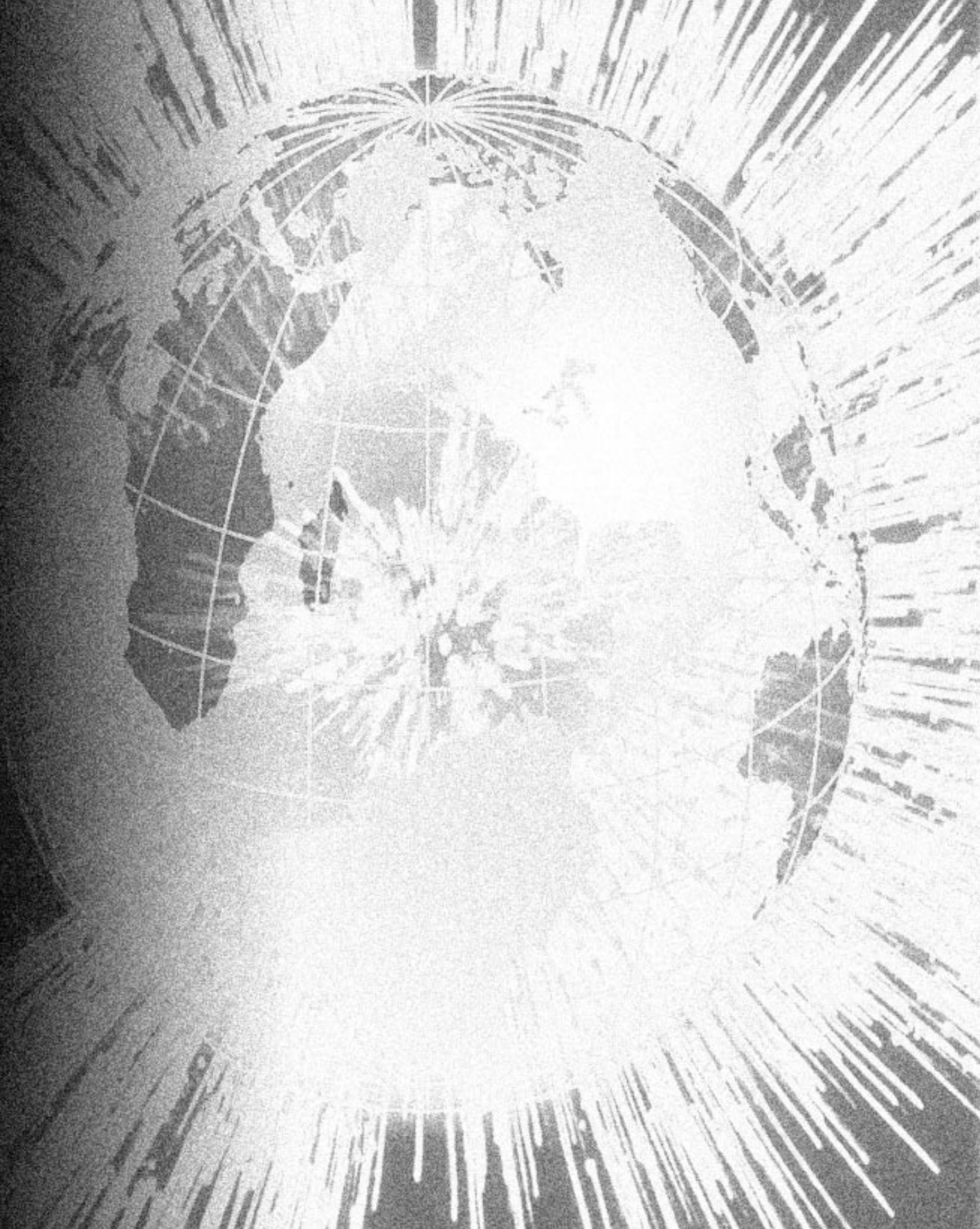
Well, what do you think? Is Java going to rule the world? We can't answer that, but in just a few pages we have shown you how to create a fairly useful applet that can be put to use immediately.

Now that we have hit many of the basics of Java programming, let's look at the details of the language. Over the next several chapters we will delve into the Java language and explore the details of its structure and syntax.



3

Java Language Fundamentals



Java Language Fundamentals

The language building blocks of Java are similar to those found in C++, but keep a close eye out because there are some subtle differences.

After following the ticker tape adventure in the previous chapter, you should now have a basic understanding of the Java language and its grammar—at least you'll know how to write a simple applet that can scroll text across the screen! Unfortunately, we covered a number of Java programming features very quickly, and we didn't get a chance to explain the main language features in sufficient detail. In this chapter and the ones that follow, we'll slow down the pace a little and uncover the key Java language features that you'll need to know to write useful Java programs. In particular, we'll explain the basic Java language components in this chapter—everything from comments to variable declarations. Then we'll move ahead and cover operators, expressions, and control structures in Chapter 4.

For those of you who are already familiar with programming, especially C or C++ programming, this chapter and Chapter 4 should serve as a good hands-on review. As we discuss Java, we'll point out the areas in which Java differs from other languages. If you don't have much experience using structured programming languages, this chapter will give you a good overview of the basic components required to make programming languages like Java come alive.

The actual language components featured in this chapter include:

- Comments
- Identifiers
- Keywords
- Data types
- Variable declarations

What Makes a Java Program?

Before we get into the details of each Java language component, let's stand back ten steps and look at how many of the key language components are used in the context of a Java program. Figure 3.1 (shown later) presents a complete visual guide. Here we've highlighted components such as variable declarations, Java keywords, operators, literals, expressions, and control structures. As we work our way through the next two chapters, you'll learn how these components are defined and used.

In case you're wondering, the output for this program looks like this:

```
Hello John my name is Anthony
That's not my name!
Let's count to ten....
1 2 3 4 5 6 7 8 9 10
Now down to zero by two.
10 8 6 4 2 0
Finally, some arithmetic:
10 * 3.09 = 30.9
10 * 3.09 = 30 (integer cast)
10 / 3.09 = 3.23625
10 / 3.09 = 3 (integer cast)
```

Lexical Structure

The lexical structure of a language refers to the elements of code that make the code easy for us to understand, but have no effect on the compiled code. For example, all the comments you place in a program to help you understand how it works are ignored by the Java compiler. You could have a thousand lines of comments for a twenty line program and the compiled *bytecodes* for the program would be the same size if you took out all the comments. This does not mean that *all* lexical structures are optional. It simply means that they do not effect the bytecodes.

The lexical structures will discuss include:

- Comments
- Identifiers
- Keywords
- Separators

Comments

Comments make your code easy to understand, modify, and use. But adding comments to an application only after it is finished is not a good practice. More often than not, you won't remember what the code you write actually does after you get away from it for a while. Unfortunately, many programmers follow this time-honored tradition. We suggest you try to get in the habit of adding comments as you write your code.

Java supports three different types of comment styles. The first two are taken directly from C and C++. The third type of comment is a new one that can be used to automatically create class and method documentation.

COMMENT STYLE #1

```
/* Comments here... */
```

This style of commenting comes to us directly from C. Everything between the initial slash-asterisk and ending asterisk-slash is ignored by the Java compiler. This style of commenting can be used anywhere in a program, even in the middle of code (not a good idea). This style of commenting is useful when you have multiple lines of comments because your comment lines can wrap from one line to the next, and you only need to use one set of the `/*` and `*/` symbols. Examples:

```
/*
This program was written by Joe Smith.
It is the greatest program ever written!
*/

while (i <= /* comments can be placed here */ maxnum)
{
    total += i;
    i++;
}
```

In the second example, the comment line is embedded within the program statement. The compiler skips over the comment text, and thus the actual line of code would be processed as:

```
while (i <= maxnum)
...
```

```

/**
 * Sample Java Application
 * @author Anthony Potts
 * @version 1.0
 */
class Test extends Object { // Begin Test class
    // Define class variables
    static int i = 10;
    static final double d = 3.09;

    /*
     The main() method is automatically called when
     the program is run. Any words typed after the program
     name when it is run are placed in the args[] variable
     which is an array of strings.
     For this program to work properly, atleast one word must
     be typed after the program name or else an error will occur.
     */
    public static void main(String args[]) {
        Test thisTest = new Test(); // Create instance (object) of class
        String myName = "Anthony";
        boolean returnValue;

        System.out.println("Hello " + args[0] + " my name is " + myName);

        if(thisTest.sameName(args[0], myName)) {
            System.out.println("Your name is the same as mine!");
        } else {
            System.out.println("That's not my name!");
        }

        System.out.println("Let's count to ten...");

        for (int x = 1; x < 11; x++) {
            System.out.print(x + " ");
        }
    }
}

```

unique Java style comment

superclass

standard C++ style comment

variable declarations

standard data type

variable

literal

declaration and assignment

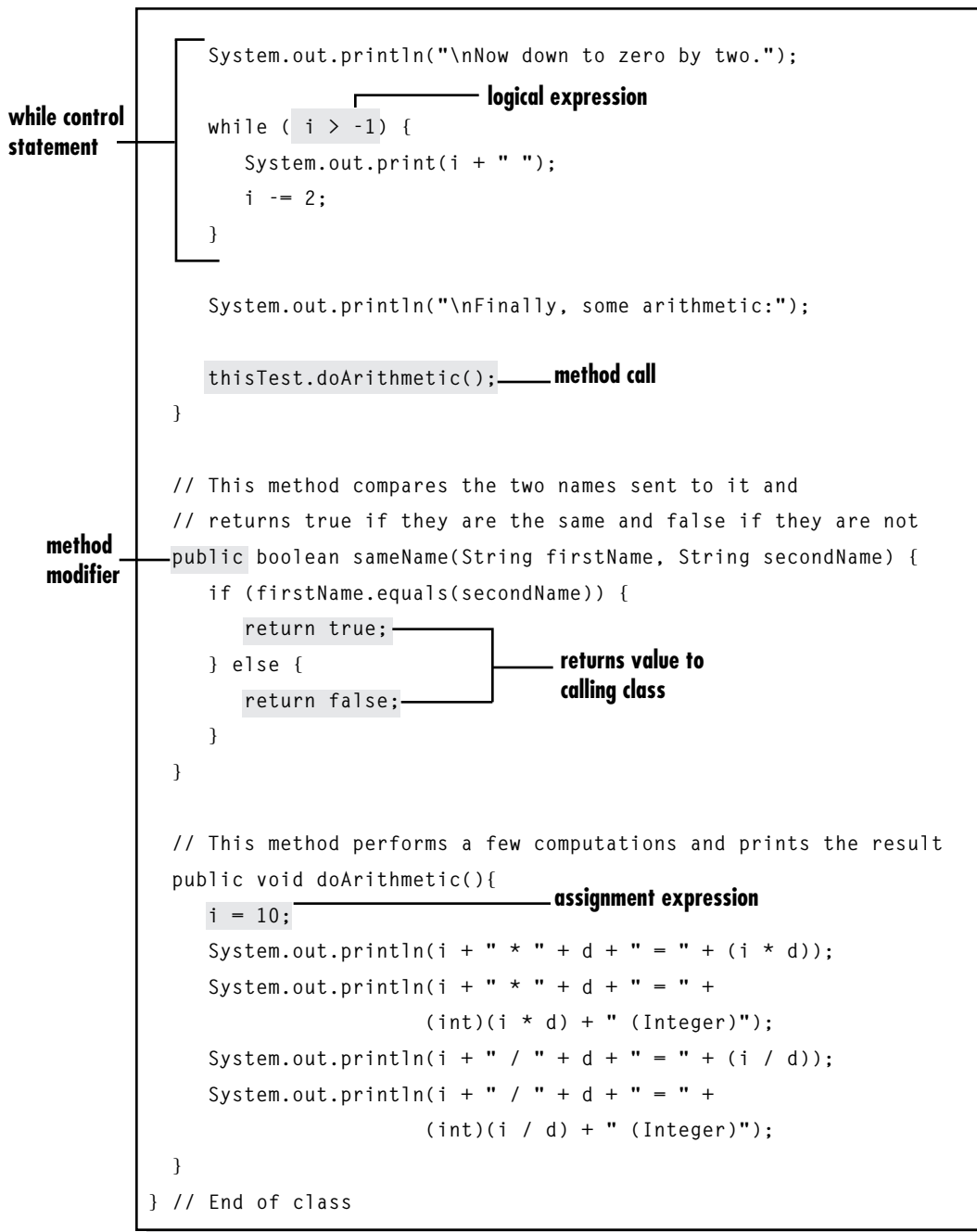
assignment operator

string data type

if-then-else control structure

increment operator

expression

**Figure 3.1**

A visual guide to the key Java language components.

62 Chapter 3

Programmers occasionally use this style of commenting while they are testing and debugging code. For example, you could comment out part of an equation or expression:

```
sum = i /* + (base - 10) */ + factor;
```

COMMENT STYLE #2

```
// Comment here...
```

This style of commenting is borrowed from C++. Everything after the double slash marks is ignored by the Java compiler. The comment is terminated by a line return, so you can't use multiple comment lines unless you start each line with the double-slash. Examples:

```
// This program was written by Joe Smith.  
// It is the greatest program ever written!  
  
while (i <= // this won't work maxnum)  
{  
    total += i;  
    i++;  
}  
  
base = 20;  
// This comment example also won't work because the Java  
    compiler will treat this second line as a line of code  
value = 50;
```

The comment used in the second example won't work like you might intend because the remainder of the line of code would be commented out (everything after `i <=`). In the third example, the second comment line is missing the starting `//` symbols, and the Java compiler will get confused because it will try to process the comment line as if it were a line of code. Believe it or not, this type of commenting mistake occurs often—so watch out for it!

COMMENT STYLE #3

```
/** Doc Comment here... */
```

This comment structure may look very similar to the C style of commenting, but that extra asterisk at the beginning makes a huge difference. Of course, remember that only one asterisk must be used as the comment terminator. The

Java compiler still ignores the comment; but another program called JAVADOC.EXE that ships with the Java Development Kit uses these comments to construct HTML documentation files that describe your packages, classes, and methods as well as all the variables they use.

Let's look at the third style of commenting in more detail. If implemented correctly and consistently, this style of commenting can provide you with numerous benefits. Figure 3.2 shows what the output of the JAVADOC program looks like when run on a typical Java source file.



Figure 3.2
Sample output from the JAVADOC program.

64 Chapter 3

If you have ever looked at the Java API documentation on Sun's Web site, Figure 3.2 should look familiar to you. In fact, the entire API documentation was created this way.

JAVADOC will work if you have created comments or not. Figure 3.3 shows the output from this simple application:

```
class HelloWorld {  
    public static void main(String args[]) {  
        System.out.println("Hello World");  
    }  
}
```

To add a little more information to our documentation, all we have to do is add this third style of comments. If we change the little HelloWorld application and add a few key comments, the code will look like this:



Figure 3.3

Simple output from the JAVADOC program.

```

/**
 * Welcome to HelloWorld
 * @author Anthony Potts
 * @version 1.1
 * @see java.lang.System
 */
class helloworld {
    /**
     * Main method of helloworld
     */
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}

```

If you now run JAVADOC, the browser will display what you see in Figure 3.4. As you can see, this gives us much more information. This system is great for producing documentation for public distribution. Just like all comments though, it is up to you to make sure that the comments are accurate and plentiful enough to be helpful. Table 3.1 lists the tags you can use in your class comments.

Identifiers

Identifiers are the names used for variables, classes, methods, packages, and interfaces to distinguish them to the compiler. In the sample program from Chapter 2 the identifier for the applet's class was **TickerTape**. We also used identifiers like **fontHeight** and **fontWidth** to name some of the variables.

Identifiers in the Java language should always begin with a letter of the alphabet, either upper or lower case. The only exceptions to this rule are the underscore symbol (`_`) and the dollar sign (`$`), which may also be used. If you try to use any other symbol or a numeral as the initial character, you will receive an error.

After the initial character you are allowed to use numbers, but not all symbols. You can also use almost all of the characters from the Unicode character set. If you are not familiar with the Unicode character set or you get errors, we suggest that you stick with the standard alphabetic characters.

The length of an identifier is basically unlimited. We managed to get up to a few thousand characters before we got bored. It's doubtful you will ever need nearly that many characters, but it is nice to know that the Java compiler won't limit

66 Chapter 3

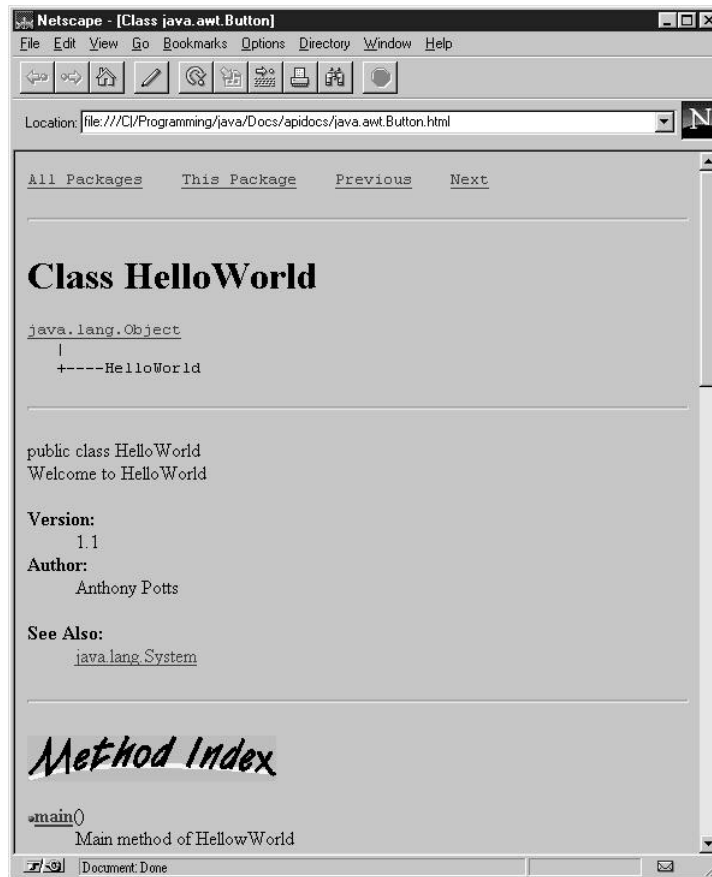


Figure 3.4

The new JAVADOC output.

you if you want to create long descriptive names. The only limit you may encounter involves creating class names. Since class names are also used as file names, you need to create names that will not cause problems with your operating system or anyone who will be using your program.

You must also be careful not to use any of the special Java keywords listed in the next section. Here are some examples of valid identifiers:

HelloWorld	\$Money	TickerTape
_ME2	Chapter3	ABC123

Table 3.1 Tags Used in Class Comments

Tag	Description
<code>@see <i>classname</i></code>	Adds a hyperlinked “See Also” to your class. The <i>classname</i> can be any other class.
<code>@see <i>fully-qualified-classname</i></code>	Also adds a “See Also” to the class, but this time you need to use a fully qualified class name like “ <code>java.awt.window</code> .”
<code>@see <i>fully-qualified-classname#methodname</i></code>	Also adds a “See Also” to the class, but now you are pointing to a specific method within that class.
<code>@version <i>version-text</i></code>	Adds a version number that you provide. The version number can be numbers or text.
<code>@author <i>author-name</i> -</code>	Adds an author entry. You can use multiple author tags. The tags you can use in your method comments include all of the “@see” tags as well as the following:
<code>@param <i>paramter-name description...</i></code>	Used to show which parameters the method accepts. Multiple “@param” tags are acceptable.
<code>@return <i>description...</i></code>	Used to describe what the method returns.
<code>@exception <i>fully-qualified-classname description...</i></code>	Used to add a “throw” entry that describes what type of exceptions this method can throw. Multiple “@exception” tags are acceptable. (Don’t worry about exceptions and throws too much yet. We will discuss these in detail in Chapter 7.)

And here are some examples of invalid identifiers:

```
3rdChapter      #Hello      -Main
```

COMMON ERRORS WITH USING IDENTIFIERS

As you are defining and using identifiers in your Java programs, you are bound to encounter some errors from time-to-time. Let’s look at some of the more common error messages that the Java compiler displays. Notice that we’ve included the part of the code that is responsible for generating the error, the error message, as well as a description of the message so that you can make sense of it.

Code Example:

```
public class ltest {
}
```

68 Chapter 3

Error Message:

D:\java\lib\test.java:1: Identifier expected.

Description:

An invalid character has been used in the class identifier. You will see this error when the first character is invalid (especially when it is a number).

Code Example:

```
public class te?st {  
}
```

Error Message:

D:\java\lib\test.java:1: '{' Expected

Description:

This is a common error that occurs when you have an invalid character in the middle of an identifier. In this case, the question mark is invalid, so the compiler gets confused where the class definition ends and its implementation begins.

Code Example:

```
public class #test {  
}
```

Error Message:

D:\java\lib\test.java:1: Invalid character in input.

Description:

Here, the error stems from the fact that the initial character is invalid.

Code Example:

```
public class catch {  
}
```

Error Message:

D:\java\lib\test.java:1: Identifier expected.

Description:

This error shows up when you use a protected keyword as an identifier.

Keywords

In Java, like other languages, there are certain *keywords* or “tokens” that are reserved for system use. These keywords can’t be used as names for your classes, variables, packages, or anything else. The keywords are used for a number of tasks such as defining control structures (*if*, *while*, and *for*) and declaring data types (*int*, *char*, and *float*). Table 3.2 provides the complete list of the Java keywords.

Table 3.2 Java Language Keywords

Keyword	Description
abstract	Class modifier
boolean	Used to define a boolean data type
break	Used to break out of loops
byte	Used to define a byte data type
byvalue *	Not implemented yet
cast	Used to translate from type to type
catch	Used with error handling
char	Used to define a character data type (16-bit)
class	Used to define a class structure
const *	Not implemented yet
continue	Used to continue an operation
default	Used with the switch statement
do	Used to create a do loop control structure
Double	Used to define a floating-point data type (64-bit)
else	Used to create an else clause for an if statement
extends	Used to subclass
final	Used to tell Java that this class can not be subclassed
finally that	Used with exceptions to determine the last option before exiting. It guarantees code gets called if an exception does or does not happen.
float	Used to define a floating-point data type (32-bit)
for	Used to create a for loop control structure
future *	Not implemented yet
generic *	Not implemented yet
goto *	Not implemented yet
if	Used to create an if-then decision-making control structure
implements	Used to define which interfaces to use
import	Used to reference external Java packages
inner	Used to create control blocks
instanceof	Used to determine if an object is of a certain type
int	Used to define an integer data type (32-bit values)
interface	Used to tell Java that the code that follows is an interface

continued

Table 3.2 Java Language Keywords (Continued)

long	Used to define an integer data type (64-bit values)
native	Used when calling external code
new	Operator used when creating an instance of a class (an object)
null	Reference to a non-existent value
operator *	Not implemented yet
outer	Used to create control blocks
package	Used to tell Java what package the following code belongs to
private	Modifier for classes, methods, and variables
protected	Modifier for classes, methods, and variables
public	Modifier for classes, methods, and variables
rest *	Not implemented yet
return	Used to set the return value of a class or method
short	Used to define an integer data type (16-bit values)
static	Modifier for classes, methods, and variables
super	Used to reference the current class' parent class
switch	Block statement used to pick from a group of choices
synchronized	Modifier that tells Java that only one instance of a method can be run at one time. It keeps Java from running the method a second time before the first is finished. It is especially useful when dealing with files to avoid conflicts.
this	Used to reference the current object
throw	Statement that tells Java what exception to pass on an errors
transient	Modifier that can access future Java code
try	Operator that is used to test for exceptions in code
var *	Not implemented yet
void	Modifier for setting the return value of a class or method to nothing
volatile	Variable modifier
while	Used to create a while loop control structure.

The words marked with an asterisk (*) are not currently used in the Java language, but you still can't use them to create your own identifiers. More than likely they will be used as keywords in future versions of the Java language.

Literals

Literals are the values that you assign when entering explicit values. For example, in an assignment statement like this:

```
i = 10;
```

the value 10 is a literal. But do not get literals confused with types. Even though they usually go hand in hand, literals and types are not the same.

Types are used to define what type of data a variable can hold, while literals are the values that are actually assigned to those variables.

Literals come in three flavors: numeric, character, and boolean. Boolean literals are simply True and False

NUMERIC LITERALS

Numeric literals are just what they sound like—numbers. We can subdivide the numeric literals further into *integers* and *floating-point* literals.

Integer literals are usually represented in *decimal* format although you can use the *hexadecimal* and octal format in Java. If you want to use the hexadecimal format, your numbers need to begin with an 0x or 0X. Octal integers simply begin with a zero (0).

Integer literals are stored differently depending on their size. The **int** data type is used to store 32-bit integer values ranging from -2,147,483,648 to 2,147,483,648 (decimal). If you need to use even larger numbers, Java switches over to the **long** data type, which can store 64 bits of information for a range of - 9.223372036855e+18 to 9.223372036855e+18. This would give you a number a little larger than 9 million trillion—enough to take care of the national debt! To specify a **long** integer, you will need to place an “l” or “L” at the end of the number. Don’t get confused by our use of the terms **int** and **long**. There are many other integer data types used by Java, but they all use **int** or **long** literals to assign values. Table 3.3 provides a summary of the two integer literals.

Table 3.3 Summary of Integer Literals

Integer Literals Ranges	Negative Minimum	Positive Maximum
int data type	-2,147,483,648	2,147,483,648
long data type	-9.223372036855e+18	9.223372036855e+18

72 Chapter 3

Here are some examples of how integer literals can be used to assign values in Java statements:

```
int i;
i = 1; // All of these literals are of the integer type
i = -9;
i = 1203131;

i = 0xA11; // Using a hexadecimal literal
i = 07543; // Using an octal literal

i = 4.5; // This would be illegal because a floating-point
        // literal can't be assigned to an integer type

long lg;
lg = 1L; // All of these literals are of the long
        // integer type

lg = -9e15;
lg = 7e12;
```

The other type of numeric literal is the floating-point literal. Floating-point values are any numbers that have anything to the right of the decimal place. Similar to integers, floating-point values have 32-bit and 64-bit representations. Both data types conform to IEEE standards. Table 3.4 provides a summary of the two floating-point literals.

Here are some examples of how floating-point literals can be used to assign values in Java statements:

```
float f;
f = 1.3; // All of these literals are of the floating-point
        // type float (32-bit)

f = -9.0;
f = 1203131.1241234;

double d;
d = 1.0D; // All of these literals are of the floating-
        // point type double(32-bit)

d = -9.3645e235;
d = 7.0001e52D;
```

Table 3.4 Summary of Floating-Point Literals

Floating-Point Ranges	Negative Minimum	Positive Maximum
float data type	1.40239846e-45	3.40282347e38
double data type	4.94065645841246544e-324	1.79769313486231570e308

CHARACTER LITERALS

The second type of literal that you need to know about is the *character literal*. Character literals include single characters and strings. Single character literals are enclosed in single quotation marks while string literals are enclosed in double quotes.

Single characters can be any one character from the Unicode character set. There are also a few special two-character combinations that are non-printing characters but perform important functions. Table 3.5 shows these special combinations.

The string character literal are any number of characters enclosed in The character combinations from Table 3.5 also apply to strings. Here are some examples of how character and string literals can be used in Java statements:

Table 3.5 Special Character Combinations in Java

Character Combination	Standard Designation	Description
\	<newline>	Continuation
\n	NL or LF	New Line
\b	BS	Backspace
\r	CR	Carriage Return
\f	FF	Form Feed
\t	HT	Horizontal Tab
\\	\	Backslash
\'	'	Single Quote
\"	"	Double Quote
\xdd	0xdd	Hex Bit Pattern
\ddd	0ddd	Octal Bit Pattern
\uddd	0xddd	Unicode Character

74 Chapter 3

```
char ch;  
ch = 'a';    // All of these literals are characters  
ch = '\n';   // Assign the newline character  
ch = '\'';   // Assign a single quote  
ch = '\x30'; // Assign a hexadecimal character code  
  
String str;  
str = "Java string";
```

Operators

Operators are used to perform computations on one or more variables or objects. You use operators to add values, comparing the size of two numbers, assigning a value to a variable, incrementing the value of a variable, and so on. Table 3.6 lists the operators used in Java. Later in this chapter, we'll explain in detail how each operator works; and we'll also explain operator precedence.

Table 3.6 Operators Used in Java	
Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
++	Increment
—	Decrement
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
!	Logical NOT
&&	Logical AND
	Logical OR
&	Bitwise AND

continued

Table 3.6 Operators Used in Java (Continued)

<code>^</code>	Bitwise exclusive OR
<code> </code>	Bitwise OR
<code>~</code>	Bitwise complement
<code><<</code>	Left shift
<code>>></code>	Right shift
<code>>>></code>	Zero fill right shift
<code>=</code>	Assignment
<code>+=</code>	Assignment with addition
<code>-=</code>	Assignment with subtraction
<code>*=</code>	Assignment with multiplication
<code>/=</code>	Assignment with division
<code>%=</code>	Assignment with modulo
<code>&=</code>	Assignment with bitwise AND
<code> =</code>	Assignment with bitwise OR
<code>^=</code>	Assignment with bitwise exclusive OR
<code><<=</code>	Assignment with left shift
<code>>>=</code>	Assignment with right shift
<code>>>>=</code>	Assignment with zero fill right shift

Separators

Separators are used in Java to delineate blocks of code. For example, you use curly brackets to enclose a method's implementation, and you use parentheses to enclose arguments being sent to a method. Table 3.7 lists the separators used in Java.

Table 3.7 Separators Used in Java

Separator	Description
<code>()</code>	Used to define blocks of arguments
<code>[]</code>	Used to define arrays
<code>{}</code>	Used to hold blocks of code
<code>,</code>	Used to separate arguments or variables in a declaration
<code>;</code>	Used to terminate lines of contiguous code

Types and Variables

Many people confuse the terms *types* and *variables*, and use them synonymously. They are, however, not the same. Variables are basically buckets that *hold information*, while types *describe what type of information* is in the bucket.

A variable must have both a type and an identifier. Later in this chapter we will cover the process of declaring variables. For now, we just want to guide you through the details of how you decide which types to use and how to use them properly.

Similar to literals, types can be split into several different categories including the numeric types—**byte**, **short**, **int**, **long**, **float**, and **double**—and the **char** and **boolean** types. We will also discuss the string type. Technically, the string type is not a type—it is a class. However, it is used so commonly that we decided to include it here.

All of the integer numeric types use signed two's-complement integers for storing data. Table 3.8 provides a summary of the ranges for each of the key Java data types.

byte

The **byte** type can be used for variables whose value falls between -256 and 255. **byte** types have an 8-bit length. Here are some examples of byte values:

-7 5 238

short

The **short** numeric type can store values ranging from -32768 to 32767. It has a 16-bit depth. Here are some examples:

-7 256 -29524

Table 3.8 Summary of the Java Data Types

Data Type	Negative Minimal	Positive Maximal
byte	-256	255
short	-32768	32767
int	-2147483648	2147483647
long	-9223372036854775808	9223372036854775807
float	1.40239846e-45	3.40282347e38
double	4.94065645841246544e-324	1.79769313486231570e308
boolean	False	True

int

The **int** data type takes the **short** type to the next level. It uses a 32-bit signed integer value that takes our minimal and maximal value up to over 2 billion. Because of this tremendous range, it is one of the most often used data types for integers.

Often, unskilled programmers will use the **int** data type even though they don't need the full resolution that this data type provides. If you are using smaller integers, you should consider using the **short** data type. The rule of thumb to follow is *if you know exactly the range of values a certain variable will store, use the smallest data type possible*. This will let your program use less memory and therefore run faster, especially on slower machines or machines with limited RAM.

Here are some examples of **int** values:

```
-7      256      -29523234      1321412422
```

long

The **long** data type is the mother of all integer types. It uses a full 64-bit data path to store values that reach up to over 9 million trillion. But be extremely careful when using variables of the **long** type. If you start using many of them or God forbid, an array of **longs**, you can quickly eat up a ton of resources.



The Danger of Using long

Java provides useful garbage collection tools, so when you are done with these large data types, they will be disposed of and their resources reclaimed. But if you are creating large arrays of **long** integers you could really be asking for trouble. For example, if you created a two-dimensional array of **long** integers that had a 100x100 grid, you would be using up about 100 kilobytes of memory.

Here are some examples of **long** values:

```
-7      256      -29523234      1.835412e15      -3e18
```


float

The **float** data type is one of two types used to store floating-point values. The **float** type is compliant with the IEEE 754 conventions. The floating-point types of Java can store gargantuan numbers. We do not have enough room on the page to physically show you the minimal and maximal values the **float** data type can store, so we will use a little bit of tricky sounding lingo taken from the Java manual.

“The finite nonzero values of type **float** are of the form $s * m * 2^e$, where s is +1 or -1, m is a positive integer less than 2^{24} and e is an integer between -149 and 104, inclusive.”

Whew, that’s a mouthful. Here are a few examples to show you what the **float** type might look like in actual use:

```
-7F      256.0   -23e34      23e100
```

double

As if the **float** type could not hold enough, the **double** data type gives you even bigger storage space. Let’s look again at Sun’s definition of the possible values for a **double**.

“The finite nonzero values of type **float** are of the form $s * m * 2^e$, where s is +1 or -1, m is a positive integer less than 2^{53} and e is an integer between -1045 and 1000, inclusive.”

Again, you can have some truly monstrous numbers here. But when you start dealing with hard core programming, this type of number becomes necessary from time to time, so it is wise to understand its ranges. Here are a few examples:

```
-7.00D   256.00D  -23e424 23e1000
```

boolean

In other languages, the **boolean** data type has been represented by an integer with a nonzero or zero value to represent True and False, respectively. This method works well because it gives the user the ability to check for all kinds of values and perform expression like this:

```
x=2;
if x then...
```

This can be handy when performing parsing operations or checking string lengths. In Java, however, the **boolean** data type has its own `True` and `False` literals that do not correspond to other values. In fact, as you will learn later in this chapter, Java does not even allow you to perform casts between the **boolean** data type and any others. There are ways around this limitation that we will discuss in a few pages when we talk about conversion methods.

char

The **char** data type is used to store single characters. Since Java uses the Unicode character set, the **char** type needs to be able to store the thousands of characters, so it uses a 16-bit signed integer. The **char** data type has the ability to be cast or converted to almost all of the others, as we will show you in the next section.

string

The **string** type is actually not a primitive data type; it is a class all its own. We decided to talk about it a little here because it is used so commonly that it might as well be considered a primitive. In C and C++, strings are stored in arrays of chars. Java does not use the **char** type for this but instead has created its own class that handles strings. In Chapter 5, when we get into the details of declaring variables within classes, you will see the difference between declaring a primitive variable and declaring an instance of a class type.

One big advantage to using a class instead of an array of **char** types is that we are more or less unlimited in the amount of information we want to place in a string variable. In C++, the array of chars was limited, but now that limitation is taken care of within the class, where we do not care how it is handled.

Variable Declarations

Declaring variables in Java is very similar to declaring variables in C/C++ as long as you are using the primitive data types. As we said before, almost everything in Java is a class—except the primitive data types. We will show you how to instantiate custom data types (including strings) in Chapter 5. For now, let's look at how primitive data types are declared.

Here is what a standard declaration for a primitive variable might look like:

```
int i;
```

80 Chapter 3

We have just declared a variable “i” to be an integer. Here are a few more examples:

```
byte i, j;
int a=7, b = a;
float f = 1.06;
String name = "Tony";
```

These examples illustrate some of the things you can do while declaring variables. Let’s look at each one individually.

```
int i;
```

This is the most basic declaration, with the data type followed by the variable you are declaring.

```
byte i, j;
```

In this example, we are declaring two byte variables at one time. There is no limit to the number of variables you can declare this way. All you have to do is add a comma between each variable you wish to declare of the given type, and Java takes care of it for you.

You also have the ability to assign values to variables as you declare them. You can even use a variable you are declaring as part of an expression for the declaration of another variable in the same line. Before we confuse you more, here is an example:

```
int i = 1;
int j = i, k= i + j;
```

Here we have first declared a variable **i** as **int** and assigned it a value of 1. In the next line, we start by declaring a variable **j** to be equal to **i**. This is perfectly legal. Next, on the same line, we declare a variable **k** to be equal to **i** plus **j**. Once again, Java handles this without a problem. We could even shorten these two statements to one line like this:

```
int i = 1, j = i, k= i + j;
```

One thing to watch out for is using variables *before* they have been declared. Here’s an example:

```
int j = i, k = i + j; // i is not defined yet
int i = 1;
```

This would cause an “undefined variable” error because Java does not know to look ahead for future declarations. Let’s look at another example:

```
float f = 1.06;
```

Does this look correct? Yes, but it’s not. This is a tricky one. By default, Java assumes that numbers with decimal points are of type **double**. So, when you try and declare a **float** to be equal to this number, you receive the following error:

```
Incompatible type for declaration. Explicit cast needed to convert double
to float.
```

Sounds complicated, but all this error message means is that you need to explicitly tell Java that the literal value 1.06 is a **float** and not a **double**. There are two ways to accomplish this. First, you can *cast* the value to a **float** like this:

```
float f = (float)1.06;
```

This works fine, but can get confusing. Java also follows the convention used by other languages of placing an “f” at the end of the literal value to indicate explicitly that it is a float. This also works for the double data type, except that you would use a “d.” (By the way, capitalization of the f and d does not make a difference.)

```
float f = 1.06f;
double d = 1.06d;
```

You should realize that the “d” is not needed in the **double** declaration because Java assumes it. However, it is better to label all of your variables when possible, especially if you are not sure.

We will cover variables and declarations in more detail in Chapter 5, but you should have enough knowledge now to be able to run a few basic programs and will delve deeper into the Java fundamentals and look at operators, expressions, and control statements.

Using Arrays

It's difficult to imagine creating any large application or applet without having an array or two. Java uses arrays in a much different manner than other languages. Instead of being a structure that holds variables, arrays in Java are actually objects that can be treated just like any other Java object.

The powerful thing to realize here is that because arrays are objects that are derived from a class, they have methods you can call to retrieve information about the array or to manipulate the array. The current version of the Java language only supports the **length** method, but you can expect that more methods will be added as the language evolves.

One of the drawbacks to the way Java implements arrays is that they are only one dimensional. In most other languages, you can create a two-dimensional array by just adding a comma and a second array size. In Java, this does not work. The way around this limitation is to create an array of arrays. Because this is easy to do, the lack of built-in support for multi-dimensional arrays shouldn't hold you back.

Declaring Arrays

Since arrays are actually instances of classes (objects), we need to use constructors to create our arrays much like we did with strings. First, we need to pick a variable name and declare it as an array object and also specify which data type the array will hold. Note that an array can only hold a single data type—you can't mix strings and integers within a single array. Here are a few examples of how array variables are declared:

```
int intArray[];  
String Names[];
```

As you can see, these look very similar to standard variable declarations, except for the brackets after the variable name. You could also put the brackets after the data type if you think this approach makes your declarations more readable:

```
int[] intArray;  
String[] Names;
```

Sizing Arrays

There are three ways to set the size of arrays. Two of them require the use of the **new** operator. Using the **new** operator initializes all of the array elements to a default value. The third method involves filling in the array elements with values as you declare it.

The first method involves taking a previously declared variable and setting the size of the array. Here are a few examples:

```
int intArray[];           // Declare the arrays
String Names[];

intArray[] = new int[10]; // Size each array
Names[] = new String[100];
```

Or, you can size the array object when you declare it:

```
int intArray[] = new int[10];
String Names[] = new String[100];
```

Finally, you can fill in the array with values at declaration time:

```
String Names[] = {"Tony", "Dave", "Jon", "Ricardo"};
int[] intArray = {1, 2, 3, 4, 5};
```

Accessing Array Elements

Now that you know how to initialize arrays, you'll need to learn how to fill them with data and then access the array elements to retrieve the data. We showed you a very simple way to add data to arrays when you initialize them, but often this just is not flexible enough for real-world programming tasks. To access an array value, you simply need to know its location. The indexing system used to access array elements is zero-based, which means that the first value is always located at position 0. Let's look at a little program that first fills in an array then prints it out:

```
public class powersOf2 {

    public static void main(String args[]) {
        int intArray[] = new int[20];
        for (int i = 0; i < intArray.length; i++) {
```

84 Chapter 3

```
        intArray[i] = 1;
        for(int p = 0; p < i; p++) intArray[i] *= 2 ;
    }
    for (int i = 0; i < intArray.length; i++)
        System.out.println("2 to the power of " + i + " is " +
            intArray[i]);
    }
```

The output of this program looks like this:

```
2 to the power of 0 is 1
2 to the power of 1 is 2
2 to the power of 2 is 4
2 to the power of 3 is 8
2 to the power of 4 is 16
2 to the power of 5 is 32
2 to the power of 6 is 64
2 to the power of 7 is 128
2 to the power of 8 is 256
2 to the power of 9 is 512
2 to the power of 10 is 1024
2 to the power of 11 is 2048
2 to the power of 12 is 4096
2 to the power of 13 is 8192
2 to the power of 14 is 16384
2 to the power of 15 is 32768
2 to the power of 16 is 65536
2 to the power of 17 is 131072
2 to the power of 18 is 262144
2 to the power of 19 is 524288
```

So, how does the program work? We first create our array of integer values and assign it to the **intArray** variable. Next, we begin a loop that goes from zero to **intArray.length**. By calling the **length** method of our array, we find the number of indexes in the array. Then, we start another loop that does the calculation and stores the result in the index specified by the **i** variable from our initial loop.

Now that we have filled in all the values for our array, we need to step back through them and print out the result. We could have just put the **print** statement in the initial loop, but the approach we used gives us a chance to use another loop that references our array.

Here is the structure of an index call:

```
arrayName[index];
```

Pretty simple. If you try and use an index that is outside the boundaries of the array, a run-time error occurs. If we change the program to count to an index of 21 instead of the actual array length of 20, we would end up getting an error message like this:

```
java.lang.ArrayIndexOutOfBoundsException: 20
    at powersOf2.main(powersOf2.java:10)
```

This is a pretty common error in any programming language. You need to use some form of exception handling to watch for this problem unless you are positive you can create code that never does this (in your dreams). See Chapter 7 for additional information on exception handling.

Multidimensional Arrays

Multidimensional arrays are created in Java in using arrays of arrays. Here are a few examples of how you can implement multidimensional arrays:

```
int intArray[][];
String Names[][];
```

We can even do the same things we did with a single dimension array. We can set the array sizes and even fill in values while we declare the arrays:

```
int intArray[][] = new int[10][5];
String Names[][] = new String[25][3];

int intArray[][] = {{2, 3, 4} {1, 2, 3}};
String Names[][] = {"Jon", "Smith"} {"Tony", "Potts"} {"Dave", "Friedel"};
```

We can also create arrays that are not “rectangular” in nature. That is, each array within the main array can have a different number of elements. Here are a few examples:

```
int intArray[][] = {{1, 2} {1, 2, 3} {1, 2, 3, 4}};
String Names[][] = {"Jon", "Smith"} {"Tony", "A", "Potts"} {"Dave", "H",
    "Friedel", "Jr."};
```


86 Chapter 3

Accessing the data in a multidimensional array is not much more difficult than accessing data in a single-dimensional array. You just need to track the values for each index. Be careful though, as you add dimensions, it becomes increasingly easy to create out of bounds errors. Here are a few examples of how you can declare multidimensional arrays, assign values, and access array elements:

```
int intArray[][] = new int[10][5];           // Declare the arrays
String Names[][] = new String[25][3];

intArray[0][0] = 5;           // Assign values
intArray[7][2] = 37;
intArray[7][9] = 37;         // This will cause an out of bounds error!
Names[0][0] = "Bill Gates";
// Access an array element in a Java statement
System.out.println(Names[0][0]);
```

We will cover variables and declarations in more detail in Chapter 5, but you should have enough knowledge now to be able to run a few basic programs and get the feel for Java programming.

Using Command-Line Arguments

Programming with command-line arguments is not a topic you'd typically expect to see in a chapter on basic data types and variable declarations. However, because we've been using command-line arguments with some of the sample programs we've been introducing, we thought it would be important to discuss how this feature works in a little more detail.

Command-line arguments are only used with Java applications. They provide a mechanism so that the user of an application can pass in information to be used by the program. Java applets, on the other hand, read in parameters using HTML tags as we learned in Chapter 2. Command-line arguments are common with languages like C and C++, which were originally designed to work with command-line operating systems like Unix.

The advantage of using command-line arguments is that they are passed to a program when the program *first* starts, which keeps the program from having to query the user for more information. Command-line arguments are great for passing custom initialization data.

Passing Arguments

The syntax for passing arguments themselves to a program is extremely simple. Just start your programs as you usually would and add any number of arguments to the end of the line with each one separated by a space. Here is a sample call to a program named “myApp”:

```
Java myApp open 640 480
```

In this case, we are calling the Java run-time interpreter and telling it to run the class file “myApp.” We then are passing in three arguments: “open,” “640,” and “480.”

If you wanted to pass in a longer string with spaces as an argument, you could. In this case, you enclose the string in quotation marks and Java will treat it as a single argument. Here is an example:

```
Java myApp "Nice program!" "640x480"
```

Once again the name of the program is “myApp.” However, this time we are only sending it two arguments: “Nice program!” and “640x480.” Note that the quotes themselves are not passed, just the string between the quotes.

Reading in Arguments

Now that we know how to pass arguments, where are they stored? How can we see them in our application? If you’ll recall, all applications have a **main()** method. You should also notice that this method has an interesting argument structure:

```
public static void main(String args[]) {  
    ...  
}
```

Here, **main()** indicates that it takes an array named **args[]** of type **String**. Java takes any command-line arguments and puts them into the **args[]** string array. The array is dynamically resized to hold just the number of arguments passed, or zero if none are passed. Note that the use of the **args** identifier is completely arbitrary. You can use any word you want as long as it conforms to the Java naming rules. You can even get a little more descriptive, like this:

88 Chapter 3

```
public static void main(String commandLineArgumentsArray[]) { ...
```

That may be a bit much, but you will never get confused as to what is in the array!

Accessing Arguments

Once we've passed in the arguments to an application and we know where they are stored, how do we get to them? Since the arguments are stored in an array, we can access them just like we would access strings in any other array. Let's look at a simple application that takes two arguments and prints them out:

```
class testArgs {  
    public static void main(String args[]) {  
        System.out.println(args[0]);  
        System.out.println(args[1]);  
    }  
}
```

If we use this command line statement to run the application

```
java testArgs hello world
```

we'd get this output:

```
hello  
world
```

Now, try this command line:

```
java testArgs onearg
```

Here is the result:

```
onearg  
java.lang.ArrayIndexOutOfBoundsException: 1  
    at testArgs.main(testArgs.java:4)
```

What happened? Since we only were passing a single argument, the reference to **args[1]** is illegal and produces an error.

So, how do we stop from getting an error? Instead of calling each argument in line, we can use a **for** loop to step through each argument. We can check the **args.length** variable to see if we have reached the last item. Our new code will also recognize if no arguments have been passed and will not try and access the array at all. Enough talking, here is the code:

```
class testArgs {
    public static void main(String args[]) {
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

Now, no matter how many arguments are passed (or none) the application can handle it.



Indexing Command-Line Arguments

Don't forget that Java arrays are zero based, so the first command-line argument is stored at position 0 not position 1. This is different than some other languages like C where the first argument would be at position 1. In C, position 0 would store the name of the program.

Dealing with Numeric Arguments

One more thing we should cover here is how to deal with numeric arguments. If you remember, all arguments are passed into an array of strings so we need to convert those values into numbers.

This is actually very simple. Each data type has an associated class that provides methods for dealing with that data type. Each of these classes has a method that creates a variable of that type from a string. Table 3.9 presents a list of those methods.

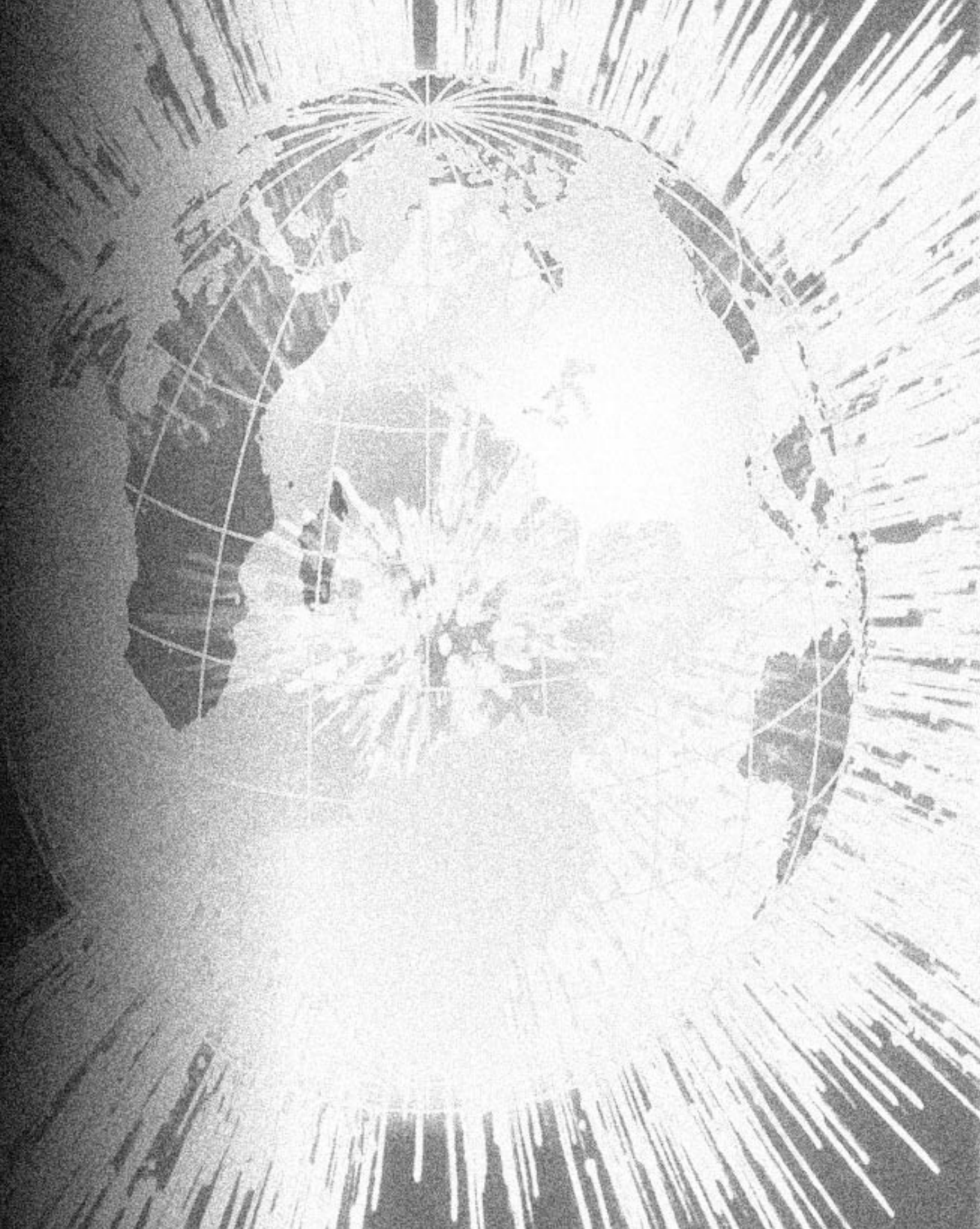
Make sure you understand the difference between the **parse*()** methods and the **valueOf()** methods. The parsing methods return just a value that can be plugged into a variable or used as part of an expression. The **valueOf()** methods return an *object* of the specified type that has an initial value equal to the value of the string.

Table 3.9 Classes and Their Associated Methods for Handling Data Types

Class	Method	Return
Integer	parseInt(String)	An integer value
Integer	valueOf(String)	An Integer object initialized to the value represented by the specified String
Long	parseLong(String)	A long value
Long	valueOf(String)	A Long object initialized to the value represented by the specified String
Double	valueOf(String)	A Double object initialized to the value represented by the specified String
Float	valueOf(String)	A Float object initialized to the value represented by the specified String

4

Operators, Expressions, and Control Structures



Operators, Expressions, and Control Structures

4

To build useful Java programs you'll need to master the art of using operators, expressions, and control structures.

Now that you know about the types of data you can use in Java, you need to learn how to manipulate your data. The tools for manipulating data fall into three categories—operators, expressions, and control structures—each playing a more powerful role as you move up the ladder. In this chapter, we'll discuss each of the key Java operators—everything from assignment statements to bitwise operators. Although Java operators are very similar to C/C++ operators, there are a few subtle differences which we'll point out. Next, we'll show you the basics for creating expressions with Java. Finally, in the last part of the chapter, we'll investigate the world of Java control structures.

Using Java Operators

Operators allow you to perform tasks such as addition, subtraction, multiplication, and assignment. Operators can be divided into three main categories: *assignment*, *integer*, and *boolean operators*. We'll explore each Java operator in detail by examining each of the three categories. But first, let's cover operator precedence.

Operator Precedence

As you are writing your code, you need to keep in mind which operators have precedence over the others—the order in which operators take effect. If you are

94 Chapter 4

an experienced programmer or you can remember some of the stuff you learned in your high school algebra classes, you shouldn't have any problem with understanding the principles of operator precedence. The basic idea is that the outcome or result of an expression like this

```
x = 5 * (7+4) - 3;
```

is determined by the *order in which the operators are evaluated* by the Java compiler. In general, all operators that have the same precedence are evaluated from left to right. If the above expression were handled in this manner, the result would be 36 (multiply 5 by 7, add 4, and then subtract 3). Because of precedence, we know that some operators, such as `()`, are evaluated before operators such as `*`. Therefore, the real value of this expression would be 52 (add 7 and 4, multiply by 5, and then subtract 3).

The actual rules for operator precedence in Java are nearly identical to those found in C/C++. The only difference is that C/C++ includes a few operators, such as `->`, that are not used in Java. Table 4.1 lists the major operators in order of precedence. Notice that some operator symbols such as `(-)` show up twice.

Table 4.1 Operator Precedence with Java

Operators	Operator Type
<code>() [] .</code>	Expression
<code>++ -- ! ~</code>	Unary
<code>* / %</code>	Multiplicative
<code>+ -</code>	Additive
<code><< >> >>></code>	Shift
<code>< <= > >=</code>	Relational (inequality)
<code>== !=</code>	Relational (equality)
<code>&</code>	Bitwise AND
<code>^</code>	Bitwise XOR
<code> </code>	Bitwise OR
<code>&&</code>	Logical AND
<code> </code>	Logical OR
<code>?:</code>	Conditional
<code>= *= /= %= += -= <<= >>= &= = ^=</code>	Assignment

The reason for this is because the operator has different meanings depending on how it is used in an expression. For example, in an expression like this

```
x = 7 + -3;
```

the (-) operator is used as a unary operator to negate the value 3. In this case, it would have a higher precedence than a standard additive operator (+ or -). In an expression like this, on the other hand,

```
x = 7 - 3 + 5;
```

the (-) operator is used as a binary additive operator, and it shares the same precedence with the (+) operator.



Which Operators Are Missing?

If you are an experienced C/C++ programmer, you're probably wondering what operators used in C/C++ are not available in Java. The ones missing are the four key data access and size operators shown in Table 4.2. These operators are not needed because Java does not support pointers and does not allow you to access memory dynamically. As we learned in Chapter 2, Java uses garbage collection techniques to provide its own internal system of memory management.

Assignment Operators

The most important and most often used operator is the assignment operator (=). This operator does just what it looks like it should do; it takes whatever variable is on the left and sets it equal to the expression on the right:

```
i = 35;
```

Table 4.2 C/C++ Operators Missing from Java

Operator	Description
*	Performs pointer indirection
&	Calculates the memory address of a variable
->	Allows a pointer to select a data structure
sizeof	Determines the size of an allocated data structure

96 Chapter 4

The expression on the right can be any valid Java expression—a literal, an equation with operands and operators, a method call, and so on. When using an assignment operator, you must be careful that the variable you are using to receive the expression is the correct size and type to receive the result of the expression on the right side. For example, statements like the following could cause you a lot of headaches:

```
short count;
// This number is way too big for a short type!
count = 5000000000000;

char ch;
// Oops! We should be assigning a character here
ch = 100;
```

In the first example, the variable **count** is declared as a **short**, which means that the variable can only hold a number as large as 32767. Obviously, the number being assigned to the variable is way too large. In the second example, the variable **ch** expects to receive a character but in reality is assigned something else entirely.

If you look closely at the last line in Table 4.1, you'll see that Java offers a number of variations of the standard assignment statement. They are all borrowed from the C language. An assignment statement like this

```
num *= 5;
```

would be equivalent to this expression:

```
num = num * 5;
```

The combination assignment operators turn out to be very useful for writing expressions inside loops that perform counting operations. Here's an example:

```
While (i <= count)
{
    i += 2;    // Increment the counting variable
    ...
}
```

In this case `i` is used as the loop “counting” control variable, and it is incremented by using a combination assignment statement.

Integer Operators

In the category of integer operators, there are two flavors to choose from: *unary* and *binary*. A unary operator performs a task on a single variable at a time. Binary operators, on the other hand, must work with two variables at a time. Let’s start with the unary operators.

UNARY OPERATORS

There are four integer unary operators: negation, bitwise complement, increment, and decrement. They are used without an assignment operation. They simply perform their operation on a given variable, changing its value appropriately.

NEGATION (-)

Unary negation changes the sign of an integer. You must be careful when reaching the lower limits of integer variables because the negative limit is always one greater than the positive limit. So, if you had a variable of type `byte` with a value of -256 and you performed a unary negation on it, an error will occur because the `byte` data type has a maximum positive value of 255. Here are some examples of how this operator can be used:

```
- k;  
-someInt;  
x = -50 + 10;
```

As we learned earlier, the negation operator is at the top end of the precedence food chain; thus, you can count on operands that use it to be evaluated first.

BITWISE COMPLEMENT (~)

Performing a bitwise complement on a variable flips each bit of the variable—all 1s become 0s and all 0s become 1s. For strict decimal calculations, this operator is not used very often. But if you are working with values that represent bit settings, such as an index into a color palette, this type of operator is invaluable. Here is an example of the unary complement operator in action:

```
// input: byte type variable bitInt = 3 (00000011 in binary)  
~bitInt;  
// Output: bitInt = 252 (11111100 in binary)
```

INCREMENT (++) AND DECREMENT (--)

The increment and decrement operators are very simple operators that simply increase or decrease an integer variable by 1 each time they are used. These operators were created as a shortcut to saying $x=x+1$. As we've already mentioned, they are often used in loops where you want a variable incremented or decremented by one each time a loop is completed. Here is an example of how each operator is used:

```
++intIncrement;  
--intDecrement;
```

BINARY OPERATORS

When you need to perform operations that involve two variables, you will be dealing with binary operators. Simple addition and subtraction are prime examples of binary operators. These operators do not change the value of either of the operands, instead they perform a function between the two operands that is placed into a third. Table 4.3 lists the complete set of the binary integer operators. Let's look at each of these operators in detail.

ADDITION, SUBTRACTION, MULTIPLICATION, AND DIVISION

These operators are the standard binary operators that we have all used since we started programming. We won't explain the theory behind algebra be-

Table 4.3 The Binary Integer Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
<<	Left Shift
>>	Right Shift
>>>	Zero-Fill Right Shift

cause we assume you already know this stuff. We will, however, give you a few examples:

```
// X=12 and Y=4
Z = X + Y; // Answer = 16
Z = X - Y; // Answer = 8
Z = X * Y; // Answer = 48
Z = X / Y; // Answer = 3
```

MODULUS

The modulus operator divides the first operand by the second operand and returns the remainder:

```
// X=11 and Y=4
Z = X % Y; // Answer = 3
```

BITWISE OPERATORS

The bitwise binary operators perform operations at the binary level on integers. They act much like custom *if..then* statements. They compare the respective bits from each of the operands and set the corresponding bit of the return variable to a 1 or 0 depending on which operator is used. The AND operator works as follows: “if both bits are 1 then return a 1, otherwise return a 0.” The OR operators works like this: “if either bit is a 1 then return a 1, otherwise return a 0.” Finally, the XOR operator works like this: “if the bits are different return a 1, if they are the same return a 0.” Table 4.4 provides a set of examples that illustrate how each bitwise operator works.

And here are some code examples to show you how to incorporate bitwise operators into your Java statements:

```
// X=3 (00000011)
// Y=2 (00000010)
Z = X & Y; // Answer: Z = 2X 00000011
//           Y 00000010
//           Z 00000010

Z = X | Y; // Answer: Z = 3X 00000011
//           Y 00000010
//           Z 00000011

Z = X ^ Y; // Answer: Z = 1X 00000011
//           Y 00000010
//           Z 00000001
```

Table 4.4 Using the Java Bitwise Operators

Operand 1	Operand 2	Bitwise Operator	Return
1	1	AND	True
1	0	AND	False
0	1	AND	False
0	0	AND	False
1	1	OR	True
1	0	OR	True
0	1	OR	True
0	0	OR	False
1	1	XOR	False
1	0	XOR	True
0	1	XOR	True
0	0	XOR	False

Boolean Operators

The boolean data type adds several new operators to the mix. All of the operators that can be used on boolean values are listed in Table 4.5.

BOOLEAN NEGATION (!)

Negation of a boolean variable simply returns the opposite of the boolean value. As you might have guessed, boolean negation is a unary operation. Here's an example:

```
// Bool1 = True
!Bool1; // Answer: Bool1 = False
```

LOGICAL AND (&), OR (|), & XOR (^)

The AND, OR, and XOR operators work identically to the way they do with integer values. However, they only have a single bit to worry about:

```
Bool2 = true;
Bool3 = true;
```

Table 4.5 Java Boolean Operators

Operator	Operation
!	Negation
&	Logical AND
	Logical OR
^	Logical XOR
&&	Evaluation AND
	Evaluation OR
==	Equal to
!=	Not Equal to
&=	And Assignment
=	OR Assignment
^=	XOR Assignment
?:	Ternary (Conditional)

```

Bool4 = False;
Bool5 = False;
Bool1 = Bool2 & Bool3; // Answer: Bool1 = True
Bool1 = Bool2 & Bool4; // Answer: Bool1 = False
Bool1 = Bool2 | Bool3; // Answer: Bool1 = False
Bool1 = Bool2 | Bool4; // Answer: Bool1 = True
Bool1 = Bool3 ^ Bool4; // Answer: Bool1 = False
Bool1 = Bool4 ^ Bool5; // Answer: Bool1 = True

```

EVALUATION AND (&&) AND OR (||)

The evaluation AND and OR are a little different than the logical versions. Using these operators causes Java to avoid evaluation of the righthand operands if it is not needed. In other words, if the answer can be derived by only reading the first operand, Java will not bother to read the second. Here are some examples:

```

// op1 = True op2 = False
result = op1 && op2; // result=False-both ops are evaluated
result = op2 && op1; // result=False-only first op is evaluated

result = op1 || op2; // result=True-only first op is evaluated
result = op2 || op1; // result=True-both ops are evaluated

```


EQUAL TO (==) AND NOT EQUAL TO (!=)

These operators are used to simply transfer a boolean value or transfer the opposite of a boolean value. Here are a few examples:

```
op1 = True;
if (result == op1); // Answer: result = true
if (result != op1); // Answer: result = false
```

ASSIGNMENT BOOLEAN OPERATORS (&=), (|=), (^=)

Boolean assignment operators are a lot like the assignment operators for integers. Here is an example of an assignment being used on both an integer and a boolean so that you can compare the two:

```
i    += 5;      // Same as int = int + 5
bool &= true;   // Same as bool = bool & true
bool |= true;   // Same as bool = bool | true
bool ^= false;  // Same as bool = bool ^ false
```

TERNARY OPERATOR

This powerful little operator acts like an extremely condensed *if..then* statement. If you look at the example below you will see that if the operand is True, the expression before the colon is evaluated. If the operand is False, the expression after the colon is evaluated. This type of coding may look a little strange at first. But once you understand the logic, you'll begin to see just how useful this operator can be. In the following example, the parentheses are not actually needed, but when you use more complicated expressions they will make the code much easier to follow:

```
// op1 = True op2 = False
op1 ? (x=1):(x=2); // Answer: x=1
op2 ? (x=1):(x=2); // Answer: x=2
```

Floating-Point Number Operators

Almost all of the integer operators work on floating-point numbers as well, with a few minor changes. Of course, all the standard arithmetic operators (+, -, *, /) work as well as the assignment operators (+=, -=, *=, /=). Modulus (%) also works; however, it only evaluates the integer portion of the operands. The increment and decrement operators work identically by adding or subtracting 1.0 from the integer portion of the numbers. Be careful when using relational operators on floating-point numbers. Do not make assumptions about how the numbers will

behave just because integers behave a certain way. For example, just because an expression like `a==b` may be true for two floating-point values, don't assume that an expression like `a<b || a>b` will be true. This is because floating-point values are not ordered like integers. You also have to deal with the possibility of a floating-point variable being equal to negative or positive infinity, `-Inf` and `Inf`, respectively. You can get a positive or negative `Inf` when you perform an operation that returns an overflow.

Using Casts

In some applications you may need to transfer one type of variable to another. Java provides us with *casting* methods to accomplish this. Casting refers to the process of transforming one variable of a certain type into another data type.

Casting is accomplished by placing the name of the data type you wish to cast a particular variable into in front of that variable in parentheses. Here is an example of how a cast can be set up to convert a `char` into an `int`:

```
int a;
char b;
b = 'z';
a = (int) b;
```

Since the variable `a` is declared as an `int`, it expects to be assigned an `int` value. The variable `b`, on the other hand, is declared as a `char`. To assign the contents of `b` to `a`, the cast is used on the right side of the assignment statement. The contents of `b`, the numeric value of the character 'z' is safely assigned to the variable `a` as an integer. If you wanted to, you could perform the cast in reverse:

```
short a;
char b;
a = 40;
b = (char) a; // Convert value 40 into a character
```

Casting is extremely simple when you are using the primitive data types—`int`, `char`, `short`, `double`, and so on. You can also cast classes and interfaces in Java, which we'll show you how to do in Chapter 5.

The most important thing to remember when using casts is the space each variable has to work with. Java will let you cast a variable of one data type into a

variable of a different data type if the size of the data type of the target variable is smaller than the other data type, but you may not like the result. Does this sound confusing? Let's explain this a little better. If you had a variable of type **long**, you should only cast it into another variable of type **float** or **double** because these data types are the only other two primitives with at least 64-bits of space to handle your number. On the other hand, if you had a variable of type **byte**, then you could cast it into any of the other primitives except boolean because they all have more space than the lowly **byte**. When you are dealing with **double** variables, you are stuck, since no other data type offers as much space as the **double**.

If you have to cast a variable into another variable having less space, Java will do it. However, any information in the extra space will be lost. On the plus side though, if the value of a larger variable is less than the maximum value of the variable you are casting into, no information will be lost.

Writing Expressions and Statements

So far we've been more or less looking at operators, literals, and data types in a vacuum. Although we've used these components to write expressions, we haven't formally defined what Java expressions are. Essentially, expressions are the Java statements that make your code work; they are the guts of your programs. A basic expressions contains *operands* and *operators*. For example, in this expression

```
i = x + 10;
```

the variable **x** and the literal **10** are the operands and **+** is the operator. The evaluation of an expression performs one or more operations that return a result. The data type of the result is always determined by the data types of the operands(s).

When multiple operands are combined, they are referred to as a *compound expression*. The order in which the operators are evaluated is determined by the precedence of the operators that act upon them. We discussed precedence earlier and showed you the relative precedence of each Java operator.

The simplest form of expression is used to calculate a value, which in turn is assigned to a variable in an assignment statement. Here are a few assignment statements that use expressions that should look very familiar to you by now:

```
i = 2;
thisString = "Hello";
```

Here are a few assignment statements that are a little more involved:

```
Bool1 != Bool2;
i += 2;
d *= 1.9
Byte1 ^= Byte2;
```

An assignment expression involves a variable that will accept the result, followed by a single assignment operator, followed by the operand that the assignment operator is using.

The next step up the ladder is to create expressions that use operators like the arithmetic operators we have already discussed:

```
i = i + 2;
thisString = "Hello";
```

Expressions with multiple operands are probably the most common type of expressions. They still have a variable that is assigned the value of the result produced by evaluating the operands and operators to the right of the equal sign. You can also have expressions with many operators and operands like this:

```
i = i + 2 - 3 * 9 / 3;
thisString = "Hello" + "World, my name is " + myName;
```

The art of programming in Java involves using operators and operands to build expressions, which are in turn used to build *statements*. Of course, the assignment statement is just one type of statement that can be constructed. You can also create many types of control statements, such as while and for loops, if-then decision making statements, and so on. (We'll look at all of the control statements that can be written in Java in the last part of this chapter.)

There are essentially two types of statements you can write in Java: *simple* and *compound*. A simple statement performs a single operation. Here are some examples:

```
int i;          // Variable declaration
i = 10 * 5;     // Assignment statement
if (i = 50) x = 200; // if-then decision statement
```

The important thing to remember about simple statements in Java is that they are *always* completed with a semicolon (;). (Some of the others like class declara-

tions and compound if..else statements don't need semicolons, but if you leave it off the end of an expression, you'll get an error.)

Compound statements involve the grouping of simple statements. In this case, the characters ({ }) are used to group the separate statements into one compound statement. Here are a few examples:

```
while (x < 10)
{
    ++x;
    if (sum < x) println();
}

if (x < 10)
{
    i = 20;
    p = getValue(i);
}
```

Notice that the (;) terminating character is not used after the final (}). The braces take care of this for us.

Control Flow Statements

Control flow is what programming is all about. What good are basic data types, variables, and casting if you don't have any code that can make use of them? Java provides several different types of control flow structures. These structures provide your application with direction. They take an input, decide what to do with it and how long to do it, and then let expressions handle the rest.

Let's look at each of these structures in detail. If you have done any programming before, all of these should look familiar. Make sure you study the syntax so that you understand exactly how they work in Java as compared to how they work in other languages.

Table 4.6 lists all of the standard control flow structures, and it shows you what the different parts of their structure represent.

if..else

The if..else control structure is probably used more than all the others combined. How many programs have you written that didn't include one? Not very many, we'll wager.

Table 4.6 Control Flow Structures

Structure	Expression
if..else	if (<i>boolean</i> = true) <i>statement</i> else <i>statement</i> ;
while	while (<i>boolean</i> = true) <i>statement</i> ;
do..while	do <i>statement</i> while (<i>boolean</i> = true);
switch	switch (<i>expression</i>) { case <i>expression</i> : <i>statement</i> ; case <i>expression</i> : <i>statement</i> ; ... default : <i>statement</i> ; }
for	for (<i>expression1</i> ; <i>expression2</i> ; <i>expression3</i>) <i>statement</i> ;
label	label : <i>statement</i> break <i>label</i> ; continue <i>label</i> ;

In its simplest terms, the **if..else** structure performs this operation: if *this* is true then do *that* otherwise do *something else*. Of course, the “otherwise” portion is optional. Since you probably already know what **if..else** statements are used for, we will just show you a few examples so you can see how they work in Java.

Here is the structure labeled with standard terms:

```
if (boolean) statement
else statement;
```

Here is a sample of what an **if..else** statement might look like with actual code:

```
if (isLunchtime) {
    Eat = true;
    Hour = 12;
}
else {
    Eat = False;
    Hour = 0;
}
```

108 Chapter 4

You can also use nested **if..else** statements:

```
if (isLunchtime) {
    Eat = true;
    Hour = 12;
}
else if (isBreakfast) {
    Eat = true;
    Hour = 6;
}
else if (isDinner) {
    Eat = true;
    Hour = 18;
}
else {
    Eat = false;
    Hour = 0;
}
```

The curly braces are used when multiple statements need to take place for each option. If we were only performing a single operation for each part of the **if..else** statement, we would not need the braces. Here is an example of an **if..else** statement that uses curly braces for one part but not the other:

```
if (isLunchtime) {
    Eat = true;
    Hour = 12;
}
else Eat = False;
```

while and do..while

The **while** and **do..while** loops perform the same function. The only difference is that the **while** loop verifies the expression *before* executing the statement, and the **do..while** loop verifies the expression *after* executing the statement. This is a major difference that can be extremely helpful if used properly.

Here are the structures labeled with standard terms:

```
while (boolean) {
    statement;
}
```

```
do {
    statement
} while(boolean);
```

while and **do..while** loops are used if you want to repeat a certain statement or block of statements until a certain expression becomes false. For example, assume you wanted to send e-mail to all of the people at a particular Web site. You could set up a **while** loop that stepped through all the people, one-by-one, sending them e-mail until you reached the last person. When the last person is reached, the loop is terminated and the program control flow moves on to the statement following the loop. Here is what that loop might look like in very simple terms:

```
boolean done = false;

while (!done){
    emailUser();
    goNextuser();
    if (noNewuser) done = true;
}
```

switch

The **switch** control flow structure is useful when you have a single expression with many possible options. The same thing can be done using recursed **if..else** statments, but that can get very confusing when you get past just a few options. The **if..else** structure is also difficult to change when it becomes highly nested.

The **switch** statement is executed by comparing the value of an initial expression or variable with other variables or expressions. Let's look at the labeled structure:

```
switch(expression) {
    case expression: statement;
    case expression: statement;
    case expression: statement;
    default: statement;
}
```

Now let's look at a real piece of code that uses the **switch** structure:

```
char age;
```


110 Chapter 4

```
System.out.print("How many computers do you own? ");
age = System.in.read();
switch(age) {
    case '0':
        System.out.println("\nWhat are you waiting for?");
        break;
    case '1':
        System.out.println("\nIs that enough these days?");
        break;
    case '2':
        System.out.println("\nPerfect!");
        break;
    default:
        System.out.println("\nToo much free time on you hands!");
}
```

The **break** statement is extremely important when dealing with **switch** structures. If the **switch** finds a case that is true, it will execute the statements for that case. When it is finished with that case, it will move on to the next one. This process continues until a match is found or the **default** statement is reached. The **break** statement tells the **switch** “OK, we found a match, let’s move on.”

The **default** clause serves as the “catch-all” statement. If all of the other cases fail, the **default** clause will be executed.

for

for loops are another programming standard that would be tough to live without. The idea behind a **for** loop is that we want to step through a sequence of numbers until a limit is reached. The loop steps through our range in whatever step increment we want, checking at the beginning of each loop to see if we have caused our “quit” expression to become true.

Here is the labeled structure of a **for** loop:

```
for (variable ; expression1 ; expression2);
```

The variable we use can either be one we have previously created, or it can be declared from within the **for** structure. Expression1 from the above example is the expression we need to stay true until the loop is finished. More often than not, this expression is something like $x < 10$ which means that we will step through

the loop until x is equal to 10 at which time the expression ($x < 10$) becomes false and drops us out of the loop.

Here is an example of a **for** loop that actually works:

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

If you put this code into an empty **main** method you should get the following output:

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

For loops are used for many different applications. They are a necessity when dealing with arrays and can really help when creating lookup tables or indexing a database.

labels

Java **labels** provide a means of controlling different kinds of loops. Sometimes, when you create a loop, you need to be able to break out of it before it finishes on its own and satisfies its completion expression. This is where **labels** come in very handy.

The key to **labels** is the **break** statement that you learned to use with the **switch** statement. You can also use the **break** statement to exit out of any loop. It is great for breaking out of **for** loops and **while** loops especially.

However, sometimes you have embedded loops and you need to be able to break out of a certain loop. A great example of this is two embedded **for** loops that are setting values in an array. If an error occurs or you get a strange value, you may want to be able to break out of one loop or another. It gets confusing if you have

112 Chapter 4

all these embedded loops and break statements all over with no apparent link to one loop or another. **labels** rectify this situation.

To use a label, you simply place an identifier followed by a colon at the beginning of the line that initiates a loop. Let's look at an example before we go further:

```
outer: for (int x = 0 ; x < 10 ; x++) {  
    inner: for (int y = 0 ; y < 10 ; y++) {  
        System.out.println(x + y);  
        if (y=9) {  
            break outer;  
        } else {  
            continue outer;  
        }  
    }  
}
```

Labels are probably new to most of you, so you may not see a need for them right away. However, as your programs become more complicated you should think about using labels where appropriate to make your code simple and more readable.

Moving Ahead

We covered a lot of ground in this chapter and the previous one. If you are new to Java programming and have little C or C++ background, make sure you understand these concepts well so that you do not get confused in the upcoming chapters.

Let's now move on and discuss another basic structure of Java programming. In fact, we would have to call it the basic structure of Java programming—the class.

Index

A

- Abstract classes, 33, 121, 158
- Abstract methods, 132
- Abstract Window Toolkit, 32
- Action() method, 294
- Add method, 274
- Addition, 98
- Addressing
 - Internet, 353
- Animation
 - buffering, 40
 - speed, 26
- API documentation, 64
- Applet class
 - hierarchy, 288
 - methods available, 288
 - methods derived, 290
- Applets, 5, 6
 - browser interaction, 294
 - class, 287
 - closing, 51
 - compiling, 53
 - defined, 21
 - drawbacks, 291
 - file access, 292
 - file execution, 292
 - fonts, 43
 - images, 298
 - navigation sample, 293
 - network communication, 292
 - package, 31
 - parameters, 39
 - passing information, 367
 - sounds, 297
 - tags, 39
 - threading, 51
 - ticker tape sample, 22
 - vs. applications, 21
- Appletviewer, 7
- Applications
 - command line arguments, 86
 - defined, 21
 - networking, 353
 - sample menu, 268
 - vs. applets, 21
- Architecture natural, 5
- Arguments
 - accessing, 88
 - indexing, 89
 - numeric, 89
 - passing, 87
 - reading, 87
- Arrays, 16, 82
 - accessing data, 86
 - declaring, 82
 - elements, 83
 - indexing, 85
 - multidimensional, 85
 - sizing, 83
- Assignment boolean operators, 102
- Assignment operators, 93, 95
- Audio clips, 297
- AWT, 31, 227
 - AWTError, 191

404 Index

- class hierarchy, 230
- components, 229
- defined, 32
- importing, 228
- layout manager, 228
- menus, 229

B

- Bandwidth considerations, 360
- Binary, 97
- Binary integer operators, 99
- Binary operators, 98
- Bitwise complement, 97
- Bitwise operators, 99
- Blocking, 217
- Body (class), 128
- Boolean data type, 78
- Boolean operators
 - assignment, 102
 - evaluation, 101
 - logical, 100
 - negation, 100
 - ternary, 102
- BorderLayout class, 274
 - declaration, 275
 - methods, 275
- Break statement, 110
- Browsers
 - HotJava, 6
 - Netscape, 26
- BufferedInput Stream class, 327
- BufferedOutputStream class, 327
- Buffering, 40, 45
- Button class
 - declaration, 243
 - getLabel(), 244
 - hierarchy, 243
 - setLabel(), 244
- Buttons, 32
- Byte streams, 321
- Byte type, 76
- ByteArrayInputStream class, 328

- ByteArrayOutputStream class, 328
- Bytecodes, 5, 53

C

- Canvas class
 - declaration, 245
 - hierarchy, 244
 - paint(), 245
- CardLayout class, 282
 - declaration, 282
 - methods, 282
- Case-sensitivity
 - declarations, 36
 - package names, 171
 - parameters, 40
 - variable names, 36
- Casting
 - interfaces, 165
 - vs. creating, 151
- Casts, 103
- Catch statements, 187
- Catch() method, 185
- Catching errors, 186
- CGI. *See Common Gateway Interface*
- Char data type, 79
- Character arrays, 16
- Character literals, 73
- Checkbox class, 245
 - declaration, 246
 - getCheckboxGroup(), 247
 - getLabel(), 247
 - getState(), 247
 - hierarchy, 246
 - setCheckboxGroup(), 247
 - setLabel(), 247
 - setState(), 247
- Choice class, 247
 - addItem(), 249
 - countItems(), 249
 - declaration, 248, 251
 - getItem(), 249

- getSelectedIndex(), 249
- getSelectedItem(), 249
- hierarchy, 248, 250
- methods, 251
- select(), 250
- Classes, 5
 - abstract, 33, 121, 158
 - advantages, 116
 - applet, 287
 - body, 128
 - bufferedInputStream, 328
 - bufferedOutputStream, 327
 - button, 243
 - byteArrayInputStream, 328
 - byteArrayOutputStream, 328
 - canvas, 244
 - casting, 150
 - checkbox, 245
 - choice, 247
 - component, 290
 - container, 290
 - dataInputStream, 330
 - dataOutputStream, 330
 - declaring, 116
 - defined, 32
 - documenting, 63
 - error, 191
 - event, 304
 - exception, 193
 - extending, 124
 - fileInputStream, 333
 - fileOutputStream, 333
 - filterInputStream, 335
 - filterOutputStream, 335
 - final, 33, 123
 - flowLayout, 271
 - frame, 235
 - fully qualified name, 118
 - hiding, 177
 - identifiers, 124
 - importing packages, 176
 - InetAddress, 354
 - inputStream, 325
 - label, 241
 - lineNumberInputStream, 337
 - list, 250
 - menuItem, 265
 - modifiers, 33, 119
 - name space, 34, 129
 - naming, 124
 - networking, 353
 - object, 34
 - outputStream, 325
 - panel, 238
 - pipedInputStream, 339
 - pipedOutputStream, 339
 - printStream, 340
 - private, 33
 - protocols, 158
 - public, 33, 120
 - pushbackInputStream, 342
 - runtime, 194
 - scrollbar, 258
 - sequenceInputStream, 342
 - socket, 355
 - stringBufferInputStream, 343
 - super(), 142
 - superclass, 34
 - System, 321
 - textArea, 253
 - textField, 253
 - throwable, 182
 - URL, 364
 - variables, 148
 - WriteAFile, 185
- CLASSPATH, 171, 173, 174
- Client, 350
- Client/server technology, 350
- Code parameter, 27
- Color method, 37
- Command line arguments, 86
 - indexing, 89
 - numeric, 89
 - passing arguments, 87
 - reading, 87
- Comments, 30
 - styles, 59
 - tags, 67

406 Index

Common Gateway Interface, 10

Compilers, 7, 53

Component class

 bounds(), 232

 disable(), 232

 enable([Boolean]), 232

 getFontMetrics(), 232

 getGraphics(), 232

 getParent, 232

 handleEvent(Event evt), 232

 hide(), 233

 inside(int x, int y), 233

 isEnabled(), 233

 isShowing(), 233

 isVisible(), 233

 locate(int x, int y), 233

 location(), 233

 move(int x, int y), 233

 repaint(), 233

 resize(), 234

 setFont(), 234

 show([Boolean]), 234

 size(), 235

Components, 60

Compound expressions, 104

Compound statements, 106

Constructors, 37, 138

 body, 146

 calling, 140

 declaring, 140

 FontMetrics, 48

 Java default, 142

 modifiers, 143

 object creation, 148

 protected, 143

 public, 143

 return type, 139

Container class, 290

Control flow, 106

Control structures

 do...while, 108

 for, 110

 if...else, 106

 labels, 111

 list of, 107

 switch, 109

 while, 108

Controls, 229

 buttons, 243

 canvas, 244

 checkbox, 245

 components, 231

 frame, 235

 label, 241

 layout manager, 270

 lists, 250

 menus, 229, 263

 panel, 238

 pop-up menus, 247

 scrollbar, 258

 text areas, 253

 text fields, 253

Converting values

 casting, 150

D

Data types, 35

 boolean, 78

 byte, 76

 casting, 103

 char, 79

 double, 78

 float, 78

 int, 71, 77

 long, 71, 77

 separators, 75

 short, 76

 string, 79

 variables, 76

DataInputStream class, 330

DataOutputStream class, 330

Debugging, 181

Decrement operator, 98

Destroy() method, 222

Developers Kit, 17

Directories

- search path, 174
- Disassembler program, 17
- Distributed programming, 6
- Distributed software, 10
- Division, 98
- Do...while, 108
- Doc comment clauses, 119
- Documenting classes, 63
- Double buffering, 45
- Double data type, 78

E

- Encapsulation, 43
- Equal to operators, 102
- Error handling, 181
- Errors
 - catching, 186
 - checking, 323
 - file not found, 189
 - input/output, 185
 - throws, 133
 - try clauses, 186
- Evaluation operators, 101
- Event class, 304
 - methods, 306
 - variables, 305
- Event handling, 53
- Events
 - hierarchy, 313
 - processing problems, 318
 - system, 315
 - types, 304
- Exceptions, 15, 181, 182
 - class, 193
 - creating, 200
 - error, 191
 - file not found, 189
 - finally statements, 189
 - handler, 182
 - IOException, 185
 - try clauses, 186
 - URL, 364

- Executable content, 10
- Export statement, 228
- Expressions
 - assignment, 105
 - writing, 104
- Extending classes, 124
- Extends keyword, 34

F

- Fatal errors, 191
- File
 - access, 292
 - execution, 292
 - input/output, 321
 - saving, 173
- File Transfer Protocol. *See FTP*
- FileInputStream class, 333
- FileOutputStream class, 333
- FilterInputStream, 335
- FilterOutputStream class, 335
- Final classes, 33
- Final methods, 132
- Finally statement, 189
- Finger protocol, 349
- Float data type, 78
- Floating-point, 72
 - operators, 102
- FlowLayout class, 271
 - declaration, 271
 - methods, 273
- Font metrics, 48
- Fonts, 43
- For loops, 110
- Frame class, 235
 - declaration, 235
 - dispose(), 237
 - getIconImage(), 237
 - getMenuBar, 237
 - getTitle(), 237
 - hierarchy, 235
 - isResizable(), 238
 - remove(), 238

408 Index

- setCursor(), 238
- setIconImage(), 238
- setMenuBar(), 238
- setResizable(), 238
- setTitle(), 238

FTP, 349

G

Garbage collection, 6, 15, 37

Gateways, 355

Graphical User Interface

- button class, 243
- canvas class, 244
- checkbox class, 245
- choice class, 247
- component class, 231
- frame class, 235
- label class, 241
- lists, 250
- menu class, 263
- menu items, 265
- menuBar class, 261
- panel class, 238
- scrollbar class, 258
- text areas, 253
- text fields, 253

Graphics methods, 46

GridBagLayout class, 278

- declaration, 281
- methods, 281
- variables to customize, 278

GridLayout class, 276

- declaration, 277
- methods, 277

H

Header files, 13

Height parameter, 27

Helper programs, 17

Hexadecimal format, 71

History of Java, 8

HotJava, 6, 10

HTML. *See Hyper Text Markup Language*

- applet tags, 39

HTTP, 349

Hyper Text Markup Language, 25

Hyper Text Transfer Protocol. *See HTTP*

I

Identifiers, 65

- class, 118
- classes, 124
- errors, 67

If...else, 106

Image buffer, 41

Images, 298

Implements clause, 126

Implements keywords, 34

Import statements, 31, 228

Including packages, 31

Increment operator, 98

Index, 84

InetAddress class, 354

Init(), 130

Input streams, 321, 324

InputStream class, 325

- methods, 325

InstanceOf operator, 17, 168

Int data type, 77

Integers, 71

- literals, 72
- operators, 93, 97

Interfaces, 34, 158

- casting, 165
- class, 126
- declaring, 161
- design issues, 160
- implementation tips, 167
- implementing, 161
- implements clauses, 126
- keyword, 161

- layout manager, 271
- runnable, 34
- tips on using, 165
- Internet
 - addressing, 353
 - java.net package, 352
 - Request for Comments, 351
- IOException, 324

J

- Java language
 - advantages, 4
 - benefits, 11
 - compared to C++, 9
 - developer's kit, 7, 17
 - history, 8
 - interfaces, 158
 - jargon, 5
 - tools, 8
 - virtual machine, 6
- JAVAC, 7, 53
- JAVADOC.EXE, 63
- Java-enabled, 7
- JAVAP, 17
- JavaScript, 7
- Just-in-Time compiler, 7

K

- Keyboard events, 311
 - keyDown(), 311
 - keyUp(), 311
- Keywords, 68
 - class, 124
 - extends, 34, 124
 - implements, 34, 162
 - interface, 161
 - list of keywords, 69
 - super, 135
 - this, 50, 135

L

- Label class, 241
 - declaration, 241
 - getAlignment(), 242
 - getText(), 242
 - hierarchy, 241
 - setAlignment(), 242
 - setText(), 243
- Labels, 111
- Layout manager, 228, 270
 - borderLayout class, 274
 - cardLayout class, 282
 - flowLayout class, 271
 - gridBagLayout class, 278
 - gridLayout class, 276
- Lexical structures, 58
 - comments, 59
 - identifiers, 65
 - keywords, 68
 - separators, 75
- LineNumberInputStream class, 337
- List class, 250
- Literals, 71
 - character, 73
 - numeric, 71
- Logical operators, 100
- Long data type, 77
- Long integers, 71

M

- Main programs, 27
- Menu class, 263
 - declaration, 264
 - hierarchy, 263
 - methods, 264
- MenuBar class, 262
 - declaration, 262
 - hierarchy, 262
 - methods, 262
- MenuItem class, 265

410 Index

- declaration, 266
 - hierarchy, 266
 - methods, 267
 - Menus, 32
 - creating, 229
 - Methods, 7, 38, 130
 - abstract, 132
 - action(), 294
 - add(), 274
 - applet class, 288
 - body, 134
 - catch(), 185
 - color, 37
 - constructors, 138
 - createImage(), 230
 - declaring, 130
 - defined, 28
 - destroy(), 222
 - disable(), 230
 - documenting, 63
 - drawString(), 48
 - final, 132
 - getGraphics(), 41
 - getMessage, 198
 - getParameter(), 40
 - graphics, 46
 - handleEvent(), 53
 - hide(), 230
 - init(), 28, 130
 - main(), 87
 - modifiers, 131
 - native, 132, 292
 - overloading, 137
 - overriding, 43, 137, 170
 - paint(), 29, 44
 - parameter lists, 133
 - parse(), 89
 - private, 132
 - protected, 131
 - public, 131
 - resume(), 220
 - return type, 133
 - Run(), 29, 214
 - sleep(), 50
 - start(), 29
 - static, 132
 - stop(), 51, 221
 - suspend(), 220
 - synchronized, 132
 - throwing an exception, 194
 - throws, 133
 - valueOf(), 89
 - write(), 184
 - yield, 221
 - Modifiers
 - abstract, 121
 - constructor, 143
 - final, 123, 150
 - method, 131
 - modifiers, 33, 119
 - public, 120
 - transient, 150
 - volatile, 150
 - Modulus operator, 99
 - Mouse events, 304, 307
 - mouseDown(), 307
 - mouseDrag(), 309
 - mouseenter(), 310
 - mouseExit(), 310
 - mousemove(), 309
 - mouseup(), 308
 - Multidimensional arrays, 85
 - Multiple inheritance, 14
 - Multiplication, 98
 - Multithreading, 7, 208
 - grouping, 226
 - synchronizing, 222
- ## N
- Name space, 129
 - Native methods, 132, 292
 - Negation operator, 97
 - Netscape
 - applet, 294
 - Network communication, 292

Network News Transfer Protocol.

See NNTP

Networking, 347

between applets, 367

classes, 353

client/server, 350

concerns, 360

java.net, 352

ports, 350

protocols, 348

sockets, 355

URLs, 364

New lines, 356

NNTP, 349

Not equal to operators, 102

Numeric literals, 71

O

Object-oriented programming, 12

Objects

arrays, 82

class, 34

creation, 148

declaring, 118

Octal integers, 71

Operators, 74, 93

addition, 98

assignment, 95, 102

binary, 98

binary integer, 99

bitwise, 99

bitwise complement, 97

boolean negation, 100

compound expressions, 104

decrement, 98

division, 98

equal to, 102

evaluation, 101

floating-point, 102

increment, 98

instanceof, 17, 168

integer, 97

logical AND, 100

modulus, 99

multiplication, 98

negation, 97

not equal, 102

precedence, 93

subtraction, 98

ternary, 102

Output streams, 324

class, 325

Overloading methods, 137

Overriding methods, 137

P

Packages, 30

applet, 31

awt, 31

case sensitivity, 171

classes, 30

creating, 168

documenting, 63

import keyword, 169

java.io, 322

java.lang, 30

java.net, 352

naming, 170

public classes, 172

standard Java, 177

Paint() method, 44

Panel class, 238

declaration, 240

hierarchy, 240

setLayout(), 241

Parameter lists

constructor, 146

Parameters, 39

code, 27

height, 27

speed, 26

values, 27

width, 27

Parsing, 89

412 Index

Performance issues
 threading, 51
PipedInputStream class, 339
PipedOutputStream class, 339
Pointers, 13
Ports, 350
 Internet, 351
 numbers, 350
Precedence (operators), 93
PrintStream class, 340
Private
 constructors, 143
 methods, 132
Processing parameters, 39
Protected
 constructors, 143
 methods, 131
Protocols
 class, 158
 finger, 349
 FTP, 349
 Internet, 351
 NNTP, 349
 Request for Comments, 351
 SMTP, 348
 TCP/IP, 348
 WhoIs, 349
Public
 classes, 33, 120
 constructors, 143
 keyword, 162
 method, 131
PushbackInputStream class, 342

R

Request for Comments. *See Request for Comments*
Resizing, 239
Resource allocation, 37
Resume() method, 220
Return type, 133
Returns, 356

RFCs. *See Request for Comments*
Run method, 214
Runnable interface, 213
Runtime class, 194

S

Savings files, 173
Scripting language, 7
Scrollbar class, 258
 hierarchy, 260
 methods, 260
Security, 12, 15, 292
Seprators, 75
SequenceInputStream class, 342
Servers, 350
 sample, 361
 setting up, 360
ServerSocket class, 360
Shadowing, 129
Short type, 76
Simple Mail Transfer Protocol.
 See SMTP
Simple statements, 105
Single inheritance, 121
Sleep() method, 50, 219
Socket class, 360
Sockets, 355
Sounds, 297
Source code
 saving, 173
Statements, 105
 catch, 187
 compound, 106
 control flow, 106
 finally, 189
 simple, 105
 switch, 109
 using semi-colons, 106
 writing, 104
Static methods, 132
Status bar, 296
Stop() method, 221

- Streams, 321
 - inputStream, 324
 - outputStream, 324
- String arrays, 16
- String type, 79
- StringBufferInputStream class, 343
- Subclasses, 44
- Subtraction, 98
- Super classes, 16
- Super keyword, 135
- Super(), 142
- Suspend() method
 - suspending execution, 220
- Switch, 109
- Synchronized methods, 132
- System class, 321
 - system.in, 322
- System events, 315
 - action(), 317
 - handleEvent(), 317

T

- Tags, 67
- TCP/IP, 348
- Ternary operators, 102
- TextArea class, 253
 - declaration, 254
 - hierarchy, 254
 - methods, 255
- TextField class, 253
 - declaration, 254
 - hierarchy, 254
 - methods, 255
- This keyword, 50, 135
- ThreadGroup, 226
- Threads, 29, 49, 182, 207, 212
 - blocking, 217
 - creating, 211
 - destroy() method, 222
 - first in first out, 217
 - grouping, 226
 - initializing, 215

- life cycle, 218
- priority, 217
- resuming, 220
- run() method, 219
- runnable interface, 213
- sleep() method, 219
- start() method, 219
- stop() method, 221
- subclassing, 212
- suspending execution, 220
- synchronizing, 222
- when to use, 210
- while loops, 210
- yield() method, 221
- Throws, 133
 - constructor, 146
 - exceptions, 194
- Transient modifiers, 150
- Transmission Control Protocol.
 - See TCP/IP*
- Try clauses, 186
- Types, 76

U

- Unary, 97
- Unicode, 73
- Uniform Resource Locator. *See URLs*
- URLs, 364
- User input, 52
- User interface
 - component class, 231
 - layout manager, 271
 - menus, 229

V

- Variable declarations, 35
- Variables
 - constructors, 37
 - declarations, 79
 - modifiers, 149
 - naming, 36

414 Index

- static, 149
- variables, 148
- vs. types, 76
- Virtual machine, 6, 210
- Volatile modifiers, 150

W

- Web sites
 - Coriolis, 25
 - JavaSoft, 54
- While, 108

- While loops, 210
- WhoIs protocol, 349
- Widening, 151
- Width parameter, 27
- Wild cards
 - hiding classes, 177
- Windows, 32

Y

- Yield() method, 221