

GLOBAL
EDITION



Starting Out with Java

From Control Structures through Objects

SIXTH EDITION

Tony Gaddis

ALWAYS LEARNING

PEARSON

STARTING OUT WITH

JAVATM

A decorative graphic consisting of a grid of small, light blue squares arranged in a 5x5 pattern, located on the left side of the blue banner.

From Control Structures
through Objects

This page intentionally left blank

STARTING OUT WITH

JAVA™

From Control Structures through Objects

SIXTH EDITION
GLOBAL EDITION

Tony Gaddis

Haywood Community College

PEARSON

Boston Columbus Indianapolis New York San Francisco Hoboken
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montréal Toronto
Delhi Mexico City São Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editor in Chief: Marcia Horton
Acquisitions Editor: Matt Goldstein
Editorial Assistant: Kelsey Loanes
Assistant Acquisitions Editor, Global Edition:
Murchana Borthakur
Associate Project Editor, Global Edition: Binita Roy
VP of Marketing: Christy Lesko
Director of Field Marketing: Tim Galligan
Product Marketing Manager: Bram van Kempen
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Director of Product Management: Erin Gregg
Team Lead Product Management: Scott Disanno
Program Manager: Carole Snyder

Production Project Manager: Camille Trentacoste
Procurement Manager: Mary Fischer
Senior Specialist, Program Planning and Support:
Maura Zaldivar-Garcia
Senior Manufacturing Controller, Production, Global Edition: Trudy Kimber
Cover Designer: Lumina Datamatics
Cover Image: © javarman/Shutterstock
Manager, Rights Management: Rachel Youdelman
Associate Project Manager, Rights Management:
William J. Opaluch
Full-Service Project Management: Kailash Jadli,
iEnergizer Aptara®, Ltd.

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsonglobaleditions.com

© Pearson Education Limited 2016

The rights of Tony Gaddis to be identified as the author of this work have been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Starting Out with Java: From Control Structures through Objects, 6th edition, ISBN 978-0-13-395705-1, by Tony Gaddis, published by Pearson Education © 2016.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided “as is” without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability. Whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time partial screen shots may be viewed in full within the software version specified.

Microsoft® Windows® and Microsoft Office® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

ISBN 10: 1-292-11065-1

ISBN 13: 978-1-292-11065-3

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

Typeset in 10 Sabon LT Std by iEnergizer Aptara®, Ltd.

Printed and bound by Courier Kendallville in the United States of America.

Contents in Brief

	Preface	23	
Chapter 1	Introduction to Computers and Java	37	
Chapter 2	Java Fundamentals	63	
Chapter 3	Decision Structures	147	
Chapter 4	Loops and Files	225	
Chapter 5	Methods	305	
Chapter 6	A First Look at Classes	355	
Chapter 7	Arrays and the <code>ArrayList</code> Class	441	
Chapter 8	A Second Look at Classes and Objects	531	
Chapter 9	Text Processing and More about Wrapper Classes	595	
Chapter 10	Inheritance	649	
Chapter 11	Exceptions and Advanced File I/O	739	
Chapter 12	A First Look at GUI Applications	797	
Chapter 13	Advanced GUI Applications	885	
Chapter 14	Applets and More	953	
Chapter 15	Creating GUI Applications with JavaFX and Scene Builder	1027	
Chapter 16	Recursion	1083	
Chapter 17	Databases	1111	
	Index	1207	
	Appendixes A–M		Companion Website
	Case Studies 1–7		Companion Website

This page intentionally left blank

Contents

Preface 23

Chapter 1 Introduction to Computers and Java 37

1.1	Introduction	37
1.2	Why Program?	37
1.3	Computer Systems: Hardware and Software.	38
	<i>Hardware</i>	38
	<i>Software</i>	41
1.4	Programming Languages	42
	<i>What Is a Program?</i>	42
	<i>A History of Java</i>	44
	<i>Java Applications and Applets</i>	44
1.5	What Is a Program Made Of?	45
	<i>Language Elements</i>	45
	<i>Lines and Statements</i>	47
	<i>Variables</i>	47
	<i>The Compiler and the Java Virtual Machine</i>	48
	<i>Java Software Editions</i>	50
	<i>Compiling and Running a Java Program.</i>	50
1.6	The Programming Process	52
	<i>Software Engineering.</i>	54
1.7	Object-Oriented Programming.	55
	<i>Review Questions and Exercises</i> 57	
	<i>Programming Challenge</i> 61	

Chapter 2 Java Fundamentals 63

2.1	The Parts of a Java Program	63
2.2	The <code>print</code> and <code>println</code> Methods, and the Java API	69
2.3	Variables and Literals	75
	<i>Displaying Multiple Items with the + Operator.</i>	76
	<i>Be Careful with Quotation Marks.</i>	77
	<i>More about Literals</i>	78

	<i>Identifiers</i>	78
	<i>Class Names</i>	80
2.4	Primitive Data Types.	80
	<i>The Integer Data Types</i>	82
	<i>Floating-Point Data Types</i>	83
	<i>The boolean Data Type</i>	86
	<i>The char Data Type</i>	86
	<i>Variable Assignment and Initialization</i>	88
	<i>Variables Hold Only One Value at a Time</i>	89
2.5	Arithmetic Operators	90
	<i>Integer Division</i>	93
	<i>Operator Precedence</i>	93
	<i>Grouping with Parentheses</i>	95
	<i>The Math Class</i>	98
2.6	Combined Assignment Operators	99
2.7	Conversion between Primitive Data Types	101
	<i>Mixed Integer Operations</i>	103
	<i>Other Mixed Mathematical Expressions</i>	104
2.8	Creating Named Constants with <code>final</code>	105
2.9	The String Class	106
	<i>Objects Are Created from Classes</i>	106
	<i>The String Class</i>	107
	<i>Primitive Type Variables and Class Type Variables</i>	107
	<i>Creating a String Object</i>	108
2.10	Scope.	111
2.11	Comments.	113
2.12	Programming Style	118
2.13	Reading Keyboard Input.	120
	<i>Reading a Character</i>	124
	<i>Mixing Calls to <code>nextLine</code> with Calls to Other Scanner Methods</i>	124
2.14	Dialog Boxes	128
	<i>Displaying Message Dialogs</i>	128
	<i>Displaying Input Dialogs</i>	129
	<i>An Example Program</i>	129
	<i>Converting String Input to Numbers</i>	131
2.15	Common Errors to Avoid	135
	<i>Review Questions and Exercises</i>	136
	<i>Programming Challenges</i>	141

Chapter 3 **Decision Structures** 147

3.1	The <code>if</code> Statement.	147
	<i>Using Relational Operators to Form Conditions</i>	149
	<i>Putting It All Together</i>	150
	<i>Programming Style and the <code>if</code> Statement</i>	154
	<i>Be Careful with Semicolons</i>	155

	<i>Having Multiple Conditionally Executed Statements</i>	155
	<i>Flags</i>	156
	<i>Comparing Characters</i>	156
3.2	The <code>if-else</code> Statement	157
3.3	Nested <code>if</code> Statements	160
3.4	The <code>if-else-if</code> Statement	167
3.5	Logical Operators	173
	<i>The Precedence of Logical Operators</i>	179
	<i>Checking Numeric Ranges with Logical Operators</i>	180
3.6	Comparing String Objects	181
	<i>Ignoring Case in String Comparisons</i>	186
3.7	More about Variable Declaration and Scope	187
3.8	The Conditional Operator (Optional)	188
3.9	The <code>switch</code> Statement	190
3.10	Displaying Formatted Output with <code>System.out.printf</code> and <code>String.format</code>	200
	<i>Format Specifier Syntax</i>	203
	<i>Precision</i>	203
	<i>Specifying a Minimum Field Width</i>	204
	<i>Flags</i>	206
	<i>Formatting String Arguments</i>	210
	<i>The <code>String.format</code> Method</i>	211
3.11	Common Errors to Avoid	214
	<i>Review Questions and Exercises</i>	215
	<i>Programming Challenges</i>	220

Chapter 4 **Loops and Files** 225

4.1	The Increment and Decrement Operators	225
	<i>The Difference between Postfix and Prefix Modes</i>	228
4.2	The <code>while</code> Loop	229
	<i>The <code>while</code> Loop Is a Pretest Loop</i>	232
	<i>Infinite Loops</i>	232
	<i>Don't Forget the Braces with a Block of Statements</i>	233
	<i>Programming Style and the <code>while</code> Loop</i>	234
4.3	Using the <code>while</code> Loop for Input Validation	236
4.4	The <code>do-while</code> Loop	240
4.5	The <code>for</code> Loop	243
	<i>The <code>for</code> Loop Is a Pretest Loop</i>	246
	<i>Avoid Modifying the Control Variable in the Body of the <code>for</code> Loop</i>	247
	<i>Other Forms of the Update Expression</i>	247
	<i>Declaring a Variable in the <code>for</code> Loop's Initialization Expression</i>	247
	<i>Creating a User Controlled <code>for</code> Loop</i>	248
	<i>Using Multiple Statements in the Initialization and Update Expressions</i>	249

4.6	Running Totals and Sentinel Values.	252
	<i>Using a Sentinel Value</i>	255
4.7	Nested Loops.	257
4.8	The <code>break</code> and <code>continue</code> Statements (Optional)	265
4.9	Deciding Which Loop to Use	265
4.10	Introduction to File Input and Output	266
	<i>Using the <code>PrintWriter</code> Class to Write Data to a File</i>	266
	<i>Appending Data to a File</i>	272
	<i>Specifying the File Location</i>	273
	<i>Reading Data from a File</i>	273
	<i>Reading Lines from a File with the <code>nextLine</code> Method</i>	274
	<i>Adding a <code>throws</code> Clause to the Method Header</i>	277
	<i>Checking for a File's Existence</i>	281
4.11	Generating Random Numbers with the <code>Random</code> Class.	285
4.12	Common Errors to Avoid	291
	<i>Review Questions and Exercises</i>	292
	<i>Programming Challenges</i>	298

Chapter 5 **Methods** 305

5.1	Introduction to Methods	305
	<i>void Methods and Value-Returning Methods</i>	306
	<i>Defining a void Method</i>	307
	<i>Calling a Method</i>	308
	<i>Hierarchical Method Calls</i>	313
	<i>Using Documentation Comments with Methods</i>	314
5.2	Passing Arguments to a Method.	315
	<i>Argument and Parameter Data Type Compatibility</i>	317
	<i>Parameter Variable Scope</i>	318
	<i>Passing Multiple Arguments</i>	318
	<i>Arguments Are Passed by Value</i>	320
	<i>Passing Object References to a Method</i>	321
	<i>Using the <code>@param</code> Tag in Documentation Comments</i>	324
5.3	More about Local Variables	327
	<i>Local Variable Lifetime</i>	328
	<i>Initializing Local Variables with Parameter Values</i>	328
5.4	Returning a Value from a Method.	329
	<i>Defining a Value-Returning Method</i>	329
	<i>Calling a Value-Returning Method</i>	331
	<i>Using the <code>@return</code> Tag in Documentation Comments</i>	332
	<i>Returning a boolean Value</i>	336
	<i>Returning a Reference to an Object</i>	336
5.5	Problem Solving with Methods	338
	<i>Calling Methods That Throw Exceptions</i>	342
5.6	Common Errors to Avoid	342
	<i>Review Questions and Exercises</i>	343
	<i>Programming Challenges</i>	348

Chapter 6 **A First Look at Classes** 355

6.1	Objects and Classes	355
	<i>Classes: Where Objects Come From</i>	356
	<i>Classes in the Java API</i>	357
	<i>Primitive Variables vs. Objects</i>	359
6.2	Writing a Simple Class, Step by Step	362
	<i>Accessor and Mutator Methods</i>	376
	<i>The Importance of Data Hiding</i>	376
	<i>Avoiding Stale Data</i>	377
	<i>Showing Access Specification in UML Diagrams</i>	377
	<i>Data Type and Parameter Notation in UML Diagrams</i>	377
	<i>Layout of Class Members</i>	378
6.3	Instance Fields and Methods	379
6.4	Constructors	384
	<i>Showing Constructors in a UML Diagram</i>	386
	<i>Uninitialized Local Reference Variables</i>	386
	<i>The Default Constructor</i>	386
	<i>Writing Your Own No-Arg Constructor</i>	387
	<i>The String Class Constructor</i>	388
6.5	Passing Objects as Arguments	396
6.6	Overloading Methods and Constructors	408
	<i>The BankAccount Class</i>	410
	<i>Overloaded Methods Make Classes More Useful</i>	416
6.7	Scope of Instance Fields	416
	<i>Shadowing</i>	417
6.8	Packages and <i>import</i> Statements	418
	<i>Explicit and Wildcard import Statements</i>	418
	<i>The java.lang Package</i>	419
	<i>Other API Packages</i>	419
6.9	Focus on Object-Oriented Design: Finding the Classes and Their Responsibilities	420
	<i>Finding the Classes</i>	420
	<i>Identifying a Class's Responsibilities</i>	423
	<i>This Is Only the Beginning</i>	426
6.10	Common Errors to Avoid	426
	<i>Review Questions and Exercises</i>	427
	<i>Programming Challenges</i>	432

Chapter 7 **Arrays and the ArrayList Class** 441

7.1	Introduction to Arrays	441
	<i>Accessing Array Elements</i>	443
	<i>Inputting and Outputting Array Contents</i>	444
	<i>Java Performs Bounds Checking</i>	447
	<i>Watch Out for Off-by-One Errors</i>	448
	<i>Array Initialization</i>	449
	<i>Alternate Array Declaration Notation</i>	450

7.2	Processing Array Elements	451
	<i>Array Length</i>	453
	<i>The Enhanced for Loop</i>	454
	<i>Letting the User Specify an Array's Size</i>	455
	<i>Reassigning Array Reference Variables</i>	457
	<i>Copying Arrays</i>	458
7.3	Passing Arrays as Arguments to Methods	460
7.4	Some Useful Array Algorithms and Operations	464
	<i>Comparing Arrays</i>	464
	<i>Summing the Values in a Numeric Array</i>	465
	<i>Getting the Average of the Values in a Numeric Array</i>	466
	<i>Finding the Highest and Lowest Values in a Numeric Array</i>	466
	<i>The SalesData Class</i>	467
	<i>Partially Filled Arrays</i>	475
	<i>Working with Arrays and Files</i>	476
7.5	Returning Arrays from Methods	477
7.6	String Arrays	479
	<i>Calling String Methods from an Array Element</i>	481
7.7	Arrays of Objects	482
7.8	The Sequential Search Algorithm	485
7.9	Two-Dimensional Arrays	488
	<i>Initializing a Two-Dimensional Array</i>	492
	<i>The length Field in a Two-Dimensional Array</i>	493
	<i>Displaying All the Elements of a Two-Dimensional Array</i>	495
	<i>Summing All the Elements of a Two-Dimensional Array</i>	495
	<i>Summing the Rows of a Two-Dimensional Array</i>	496
	<i>Summing the Columns of a Two-Dimensional Array</i>	496
	<i>Passing Two-Dimensional Arrays to Methods</i>	497
	<i>Ragged Arrays</i>	499
7.10	Arrays with Three or More Dimensions	500
7.11	The Selection Sort and the Binary Search Algorithms	501
	<i>The Selection Sort Algorithm</i>	501
	<i>The Binary Search Algorithm</i>	504
7.12	Command-Line Arguments and Variable-Length Argument Lists	506
	<i>Command-Line Arguments</i>	507
	<i>Variable-Length Argument Lists</i>	508
7.13	The ArrayList Class	510
	<i>Creating and Using an ArrayList Object</i>	511
	<i>Using the Enhanced for Loop with an ArrayList</i>	512
	<i>The ArrayList Class's toString method</i>	513
	<i>Removing an Item from an ArrayList</i>	514
	<i>Inserting an Item</i>	515
	<i>Replacing an Item</i>	516
	<i>Capacity</i>	517
	<i>Using the Diamond Operator for Type Inference (Java 7)</i>	518
7.14	Common Errors to Avoid	519

Review Questions and Exercises 519

Programming Challenges 524

Chapter 8 **A Second Look at Classes and Objects** 531

8.1	Static Class Members	531
	<i>A Quick Review of Instance Fields and Instance Methods</i>	531
	<i>Static Members</i>	532
	<i>Static Fields</i>	532
	<i>Static Methods</i>	535
8.2	Passing Objects as Arguments to Methods	538
8.3	Returning Objects from Methods	541
8.4	The <code>toString</code> Method	543
8.5	Writing an <code>equals</code> Method	547
8.6	Methods That Copy Objects	550
	<i>Copy Constructors</i>	552
8.7	Aggregation	553
	<i>Aggregation in UML Diagrams</i>	561
	<i>Security Issues with Aggregate Classes</i>	561
	<i>Avoid Using <code>null</code> References</i>	563
8.8	The <code>this</code> Reference Variable	566
	<i>Using <code>this</code> to Overcome Shadowing</i>	567
	<i>Using <code>this</code> to Call an Overloaded Constructor</i>	568
8.9	Enumerated Types	569
	<i>Enumerated Types Are Specialized Classes</i>	570
	<i>Switching On an Enumerated Type</i>	576
8.10	Garbage Collection	578
	<i>The <code>finalize</code> Method</i>	580
8.11	Focus on Object-Oriented Design: Class Collaboration	580
	<i>Determining Class Collaborations with CRC Cards</i>	583
8.12	Common Errors to Avoid	584
	<i>Review Questions and Exercises</i> 585	
	<i>Programming Challenges</i> 589	

Chapter 9 **Text Processing and More about Wrapper Classes** 595

9.1	Introduction to Wrapper Classes	595
9.2	Character Testing and Conversion with the <code>Character</code> Class	596
	<i>Character Case Conversion</i>	601
9.3	More <code>String</code> Methods	604
	<i>Searching for Substrings</i>	604
	<i>Extracting Substrings</i>	611
	<i>Methods That Return a Modified <code>String</code></i>	615
	<i>The Static <code>valueOf</code> Methods</i>	616

9.4	The <code>StringBuilder</code> Class	618
	<i>The <code>StringBuilder</code> Constructors</i>	619
	<i>Other <code>StringBuilder</code> Methods</i>	620
	<i>The <code>toString</code> Method</i>	623
9.5	Tokenizing Strings	629
9.6	Wrapper Classes for the Numeric Data Types	633
	<i>The Static <code>toString</code> Methods</i>	634
	<i>The <code>toBinaryString</code>, <code>toHexString</code>, and <code>toOctalString</code> Methods</i>	634
	<i>The <code>MIN_VALUE</code> and <code>MAX_VALUE</code> Constants</i>	634
	<i>Autoboxing and Unboxing</i>	634
9.7	Focus on Problem Solving: The <code>TestScoreReader</code> Class	636
9.8	Common Errors to Avoid	640
	<i>Review Questions and Exercises</i>	641
	<i>Programming Challenges</i>	644

Chapter 10 **Inheritance** 649

10.1	What Is Inheritance?	649
	<i>Generalization and Specialization</i>	649
	<i>Inheritance and the “Is a” Relationship</i>	650
	<i>Inheritance in UML Diagrams</i>	658
	<i>The Superclass’s Constructor</i>	659
	<i>Inheritance Does Not Work in Reverse</i>	661
10.2	Calling the Superclass Constructor	662
	<i>When the Superclass Has No Default</i>	
	<i>or No-Arg Constructors</i>	668
	<i>Summary of Constructor Issues in Inheritance</i>	669
10.3	Overriding Superclass Methods	670
	<i>Overloading versus Overriding</i>	675
	<i>Preventing a Method from Being Overridden</i>	678
10.4	Protected Members	679
	<i>Package Access</i>	684
10.5	Chains of Inheritance	685
	<i>Class Hierarchies</i>	691
10.6	The <code>Object</code> Class	691
10.7	Polymorphism	693
	<i>Polymorphism and Dynamic Binding</i>	694
	<i>The “Is-a” Relationship Does Not Work in Reverse</i>	696
	<i>The <code>instanceof</code> Operator</i>	697
10.8	Abstract Classes and Abstract Methods	698
	<i>Abstract Classes in UML</i>	704
10.9	Interfaces	705
	<i>An Interface is a Contract</i>	707
	<i>Fields in Interfaces</i>	711
	<i>Implementing Multiple Interfaces</i>	711
	<i>Interfaces in UML</i>	711

	<i>Default Methods</i>	712
	<i>Polymorphism and Interfaces</i>	714
10.10	Anonymous Inner Classes	719
10.11	Functional Interfaces and Lambda Expressions	722
10.12	Common Errors to Avoid	727
	<i>Review Questions and Exercises</i>	728
	<i>Programming Challenges</i>	734

Chapter 11 **Exceptions and Advanced File I/O** 739

11.1	Handling Exceptions	739
	<i>Exception Classes</i>	740
	<i>Handling an Exception</i>	741
	<i>Retrieving the Default Error Message</i>	745
	<i>Polymorphic References to Exceptions</i>	748
	<i>Using Multiple catch Clauses to Handle Multiple Exceptions</i>	748
	<i>The finally Clause</i>	756
	<i>The Stack Trace</i>	758
	<i>Handling Multiple Exceptions with One catch Clause (Java 7)</i>	759
	<i>When an Exception Is Not Caught</i>	761
	<i>Checked and Unchecked Exceptions</i>	762
11.2	Throwing Exceptions	763
	<i>Creating Your Own Exception Classes</i>	766
	<i>Using the @exception Tag in Documentation Comments</i>	769
11.3	Advanced Topics: Binary Files, Random Access Files, and Object Serialization	770
	<i>Binary Files</i>	770
	<i>Random Access Files</i>	777
	<i>Object Serialization</i>	782
	<i>Serializing Aggregate Objects</i>	786
11.4	Common Errors to Avoid	787
	<i>Review Questions and Exercises</i>	787
	<i>Programming Challenges</i>	793

Chapter 12 **A First Look at GUI Applications** 797

12.1	Introduction	797
	<i>The JFC, AWT, and Swing</i>	798
	<i>Event-Driven Programming</i>	800
	<i>The javax.swing and java.awt Packages</i>	800
12.2	Creating Windows	800
	<i>Using Inheritance to Extend the JFrame Class</i>	803
	<i>Equipping GUI Classes with a main Method</i>	805
	<i>Adding Components to a Window</i>	807
	<i>Handling Events with Action Listeners</i>	813

	<i>Writing an Event Listener for the <code>KiloConverter</code> Class</i>	815
	<i>Background and Foreground Colors</i>	820
	<i>The <code>ActionEvent</code> Object</i>	824
12.3	Layout Managers	829
	<i>Adding a Layout Manager to a Container</i>	830
	<i>The <code>FlowLayout</code> Manager</i>	830
	<i>The <code>BorderLayout</code> Manager</i>	833
	<i>The <code>GridLayout</code> Manager</i>	840
12.4	Radio Buttons and Check Boxes	846
	<i>Radio Buttons</i>	846
	<i>Check Boxes</i>	852
12.5	Borders	857
12.6	Focus on Problem Solving: Extending Classes from <code>JPanel</code>	860
	<i>The Brandi's Bagel House Application</i>	860
	<i>The <code>GreetingPanel</code> Class</i>	861
	<i>The <code>BagelPanel</code> Class</i>	862
	<i>The <code>ToppingPanel</code> Class</i>	864
	<i>The <code>CoffeePanel</code> Class</i>	866
	<i>Putting It All Together</i>	868
12.7	Splash Screens	872
12.8	Using Console Output to Debug a GUI Application	873
12.9	Common Errors to Avoid	878
	<i>Review Questions and Exercises</i>	878
	<i>Programming Challenges</i>	881

Chapter 13 **Advanced GUI Applications** 885

13.1	The Swing and AWT Class Hierarchy	885
13.2	Read-Only Text Fields	886
13.3	Lists	888
	<i>Selection Modes</i>	888
	<i>Responding to List Events</i>	889
	<i>Retrieving the Selected Item</i>	890
	<i>Placing a Border around a List</i>	894
	<i>Adding a Scroll Bar to a List</i>	894
	<i>Adding Items to an Existing <code>JList</code> Component</i>	899
	<i>Multiple Selection Lists</i>	899
13.4	Combo Boxes	904
	<i>Retrieving the Selected Item</i>	905
13.5	Displaying Images in Labels and Buttons	910
13.6	Mnemonics and Tool Tips	916
	<i>Mnemonics</i>	916
	<i>Tool Tips</i>	918
13.7	File Choosers and Color Choosers	918
	<i>File Choosers</i>	919
	<i>Color Choosers</i>	921

13.8	Menus	922
13.9	More about Text Components: Text Areas and Fonts	931
	<i>Text Areas</i>	931
	<i>Fonts</i>	934
13.10	Sliders	935
13.11	Look and Feel	940
13.12	Common Errors to Avoid	942
	<i>Review Questions and Exercises</i>	943
	<i>Programming Challenges</i>	948

Chapter 14 **Applets and More** 953

14.1	Introduction to Applets	953
14.2	A Brief Introduction to HTML	955
	<i>Hypertext</i>	955
	<i>Markup Language</i>	956
	<i>Document Structure Tags</i>	956
	<i>Text Formatting Tags</i>	958
	<i>Creating Breaks in Text</i>	960
	<i>Inserting Links</i>	963
14.3	Creating Applets with Swing	964
	<i>Running an Applet</i>	966
	<i>Handling Events in an Applet</i>	968
14.4	Using AWT for Portability	973
14.5	Drawing Shapes	978
	<i>The XY Coordinate System</i>	978
	<i>Graphics Objects</i>	978
	<i>The repaint Method</i>	992
	<i>Drawing on Panels</i>	993
14.6	Handling Mouse Events	999
	<i>Handling Mouse Events</i>	999
14.7	Timer Objects	1009
14.8	Playing Audio	1013
	<i>Using an AudioClip Object</i>	1014
	<i>Playing Audio in an Application</i>	1017
14.9	Common Errors to Avoid	1018
	<i>Review Questions and Exercises</i>	1018
	<i>Programming Challenges</i>	1024

Chapter 15 **Creating GUI Applications with JavaFX and Scene Builder** 1027

15.1	Introduction	1027
	<i>Event-Driven Programming</i>	1029
15.2	Scene Graphs	1029

15.3	Using Scene Builder to Create JavaFX Applications	1031
	<i>Starting Scene Builder</i>	1032
	<i>The Scene Builder Main Window</i>	1033
15.4	Writing the Application Code	1045
	<i>The Main Application Class</i>	1046
	<i>The Controller Class</i>	1048
	<i>Using the Sample Controller Skeleton</i>	1053
	<i>Summary of Creating a JavaFX Application</i>	1054
15.5	RadioButtons and CheckBoxes	1055
	<i>RadioButtons</i>	1055
	<i>Determining in Code Whether a RadioButton Is Selected</i>	1057
	<i>Responding to RadioButton Events</i>	1060
	<i>CheckBoxes</i>	1063
	<i>Determining in Code Whether a CheckBox Is Selected</i>	1064
	<i>Responding to CheckBox Events</i>	1066
15.6	Displaying Images	1069
	<i>Displaying an Image with Code</i>	1070
15.7	Common Errors to Avoid	1074
	<i>Review Questions and Exercises</i>	1074
	<i>Programming Challenges</i>	1078

Chapter 16 **Recursion** 1083

16.1	Introduction to Recursion	1083
16.2	Solving Problems with Recursion	1086
	<i>Direct and Indirect Recursion</i>	1090
16.3	Examples of Recursive Methods	1091
	<i>Summing a Range of Array Elements with Recursion</i>	1091
	<i>Drawing Concentric Circles</i>	1092
	<i>The Fibonacci Series</i>	1094
	<i>Finding the Greatest Common Divisor</i>	1096
16.4	A Recursive Binary Search Method	1097
16.5	The Towers of Hanoi	1100
16.6	Common Errors to Avoid	1105
	<i>Review Questions and Exercises</i>	1105
	<i>Programming Challenges</i>	1108

Chapter 17 **Databases** 1111

17.1	Introduction to Database Management Systems	1111
	<i>JDBC</i>	1112
	<i>SQL</i>	1113
	<i>Using a DBMS</i>	1113
	<i>Java DB</i>	1114
	<i>Creating the CoffeeDB Database</i>	1114


	<i>Connecting to the coffeeDB Database</i>	1114
	<i>Connecting to a Password-Protected Database</i>	1116
17.2	Tables, Rows, and Columns	1117
	<i>Column Data Types</i>	1119
	<i>Primary Keys</i>	1119
17.3	Introduction to the SQL <i>SELECT</i> Statement	1120
	<i>Passing an SQL Statement to the DBMS</i>	1122
	<i>Specifying Search Criteria with the WHERE Clause</i>	1132
	<i>Sorting the Results of a SELECT Query</i>	1138
	<i>Mathematical Functions</i>	1139
17.4	Inserting Rows	1142
	<i>Inserting Rows with JDBC</i>	1144
17.5	Updating and Deleting Existing Rows	1146
	<i>Updating Rows with JDBC</i>	1147
	<i>Deleting Rows with the DELETE Statement</i>	1151
	<i>Deleting Rows with JDBC</i>	1151
17.6	Creating and Deleting Tables	1155
	<i>Removing a Table with the DROP TABLE Statement</i>	1158
17.7	Creating a New Database with JDBC	1158
17.8	Scrollable Result Sets	1160
17.9	Result Set Metadata	1161
17.10	Displaying Query Results in a <i>JTable</i>	1165
17.11	Relational Data	1175
	<i>Joining Data from Multiple Tables</i>	1178
	<i>An Order Entry System</i>	1179
17.12	Advanced Topics	1197
	<i>Transactions</i>	1197
	<i>Stored Procedures</i>	1198
17.13	Common Errors to Avoid	1199
	<i>Review Questions and Exercises</i>	1199
	<i>Programming Challenges</i>	1204

Index 1207

Companion Website:

Appendix A	Working with Records and Random Access Files
Appendix B	The ASCII/Unicode Characters
Appendix C	Operator Precedence and Associativity
Appendix D	Java Key Words
Appendix E	Installing the JDK and JDK Documentation
Appendix F	Using the <i>javadoc</i> Utility
Appendix G	More about the <i>Math</i> Class
Appendix H	Packages
Appendix I	More about <i>JOptionPane</i> Dialog Boxes
Appendix J	Answers to Checkpoints
Appendix K	Answers to Odd-Numbered Review Questions

Appendix L	Getting Started with Alice
Appendix M	Configuring JavaDB
Case Study 1	Calculating Sales Commission
Case Study 2	The Amortization Class
Case Study 3	The PinTester Class
Case Study 4	Parallel Arrays
Case Study 5	The FeetInches Class
Case Study 6	The SerialNumber Class
Case Study 7	A Simple Text Editor Application

LOCATION OF VIDEONOTES IN THE TEXT		
Chapter 1	Compiling and Running a Java Program, p. 50 Using an IDE, p. 51 Your First Java Program, p. 61	
Chapter 2	Displaying Console Output, p. 69 Declaring Variables, p. 75 Simple Math Expressions, p. 91 The Miles-per-Gallon Problem, p. 142	
Chapter 3	The <code>if</code> Statement, p. 147 The <code>if-else</code> Statement, p. 157 The <code>if-else-if</code> Statement, p. 168 The Time Calculator Problem, p. 221	
Chapter 4	The <code>while</code> Loop, p. 229 The Pennies for Pay Problem, p. 299	
Chapter 5	Passing Arguments to a Method, p. 315 Returning a Value from a Method, p. 329 The Retail Price Calculator Problem, p. 348	
Chapter 6	Writing Classes and Creating Objects, p. 363 Initializing an Object with a Constructor, p. 384 The Personal Information Class Problem, p. 433	
Chapter 7	Accessing Array Elements in a Loop, p. 445 Passing an Array to a Method, p. 460 The Charge Account Validation Problem, p. 525	
Chapter 8	Returning Objects from Methods, p. 541 Aggregation, p. 553 The <code>BankAccount</code> , Class Copy Constructor Problem, p. 590	
Chapter 9	The Sentence Capitalizer Problem, p. 644	
Chapter 10	Inheritance, p. 649 Polymorphism, p. 693 The <code>Employee</code> and <code>Productionworker</code> Classes Problem, p. 734	
Chapter 11	Handling Exceptions, p. 739 The Exception Project Problem, p. 795	

(continued on the next page)

LOCATION OF VIDEONOTES IN THE TEXT *(continued)*



Chapter 12	Creating a Simple GUI Application, p. 800 Handling Events, p. 813 The Monthly Sales Tax Problem, p. 882
Chapter 13	The <code>JList</code> Component, p. 888 The <code>JComboBox</code> Component, p. 904 The Image Viewer Problem, p. 948
Chapter 14	Creating an Applet, p. 965 The <code>House</code> Applet Problem, p. 1024
Chapter 15	Using Scene Builder to Create the Kilometer Converter GUI, p. 1034 Learning More About the Main Application Class, p. 1046 Writing the Main Application Class For the Kilometer Converter GUI, p. 1047 Learning More About the Controller Class, p. 1049 Registering the Controller Class with the Application's GUI, p. 1050 JavaFX <code>RadioButtons</code> , p. 1055 JavaFX <code>CheckBoxes</code> , p. 1063 The Retail Price Calculator Problem, p. 1078
Chapter 16	Reducing a Problem with Recursion, p. 1087 The Recursive Power Problem, p. 1109
Chapter 17	Displaying Query Results in a <code>JTable</code> , p. 1165 The Customer Inserter Problem, p. 1204

Preface

Welcome to *Starting Out with Java: From Control Structures through Objects*, Sixth Edition. This book is intended for a one-semester or a two-quarter CS1 course. Although it is written for students with no prior programming background, even experienced students will benefit from its depth of detail.

Control Structures First, Then Objects

This text first introduces the student to the fundamentals of data types, input and output, control structures, methods, and objects created from standard library classes.

Next, the student learns to use arrays of primitive types and reference types. After this, the student progresses through more advanced topics, such as inheritance, polymorphism, the creation and management of packages, GUI applications, recursion, and database programming. From early in the book, applications are documented with javadoc comments. As the student progresses through the text, new javadoc tags are covered and demonstrated.

As with all the books in the *Starting Out With . . .* series, the hallmark of this text is its clear, friendly, and easy-to-understand writing. In addition, it is rich in example programs that are concise and practical.

Changes in This Edition

This book's pedagogy, organization, and clear writing style remain the same as in the previous edition. Many improvements have been made, which are summarized here:

- **A New Chapter on JavaFX:** New to this edition is *Chapter 15 Creating GUI Applications with JavaFX and Scene Builder*. JavaFX is the next generation toolkit for creating GUIs and graphical applications in Java, and is bundled with Java 8. This new chapter introduces the student to the JavaFX library, and shows how to use Scene Builder (a free download from Oracle) to visually design GUIs. The chapter is written in such a way that it is independent from the existing chapters on Swing and AWT. The instructor can choose to skip the Swing and AWT chapters and go straight to JavaFX, or cover all of the GUI chapters.

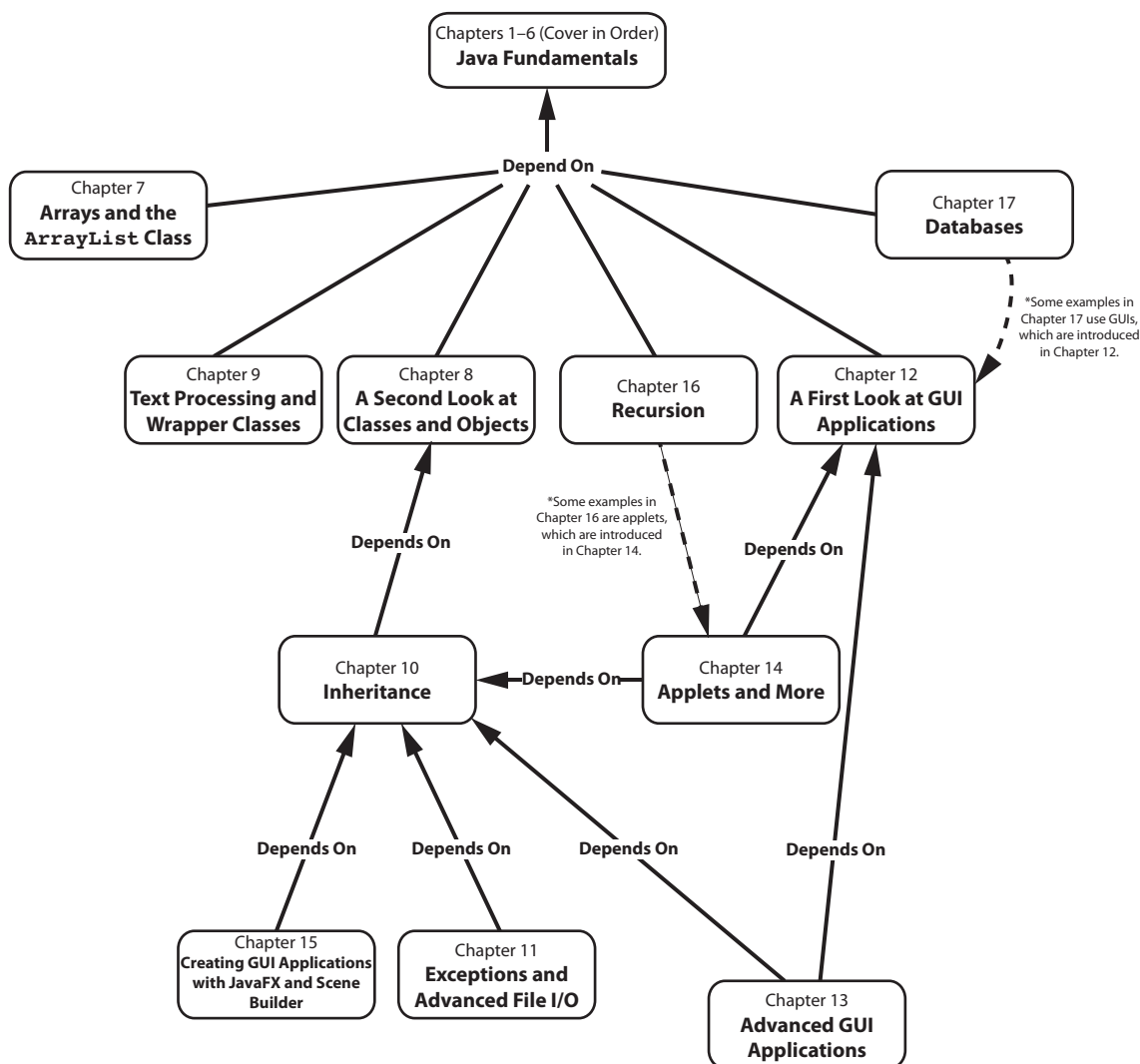
- **String.format Is Used Instead of DecimalFormat:** In previous editions, the `DecimalFormat` class was used to format strings for GUI output. In this edition, the `String.format` method is used instead. With `String.format`, the student can use the same format specifiers and flags that were learned with the `System.out.printf` method.
- **StringTokenizer Is No Longer Used:** In previous editions, the `StringTokenizer` class was introduced as a way to tokenize strings. In this edition, all string tokenizing is done with the `String.split` method.
- **Introduction of @Override annotation:** Chapter 10 now introduces the use of `@Override` annotation, and explains how it can prevent subtle errors.
- **A New Section on Anonymous Inner Classes:** Chapter 10 now has a new section that introduces anonymous inner classes.
- **The Introduction to Interfaces Has Been Improved:** The introductory material on interfaces in Chapter 10 has been revised for greater clarity.
- **Default Methods:** In this edition, Chapter 10 provides new material on default methods in interfaces, a new feature in Java 8.
- **Functional Interfaces and Lambda Expressions:** Java 8 introduces functional interfaces and lambda expressions, and in this edition, Chapter 10 has a new section on these topics. The new material gives a detailed, stepped-out explanation of lambda expressions, and discusses how they can be used to instantiate objects of anonymous classes that implement functional interfaces.
- **New Programming Problems:** Several new motivational programming problems have been added to many of the chapters.

Organization of the Text

The text teaches Java step-by-step. Each chapter covers a major set of topics and builds knowledge as students progress through the book. Although the chapters can be easily taught in their existing sequence, there is some flexibility. Figure P-1 shows chapter dependencies. Each box represents a chapter or a group of chapters. An arrow points from a chapter to the chapter that must be previously covered.

Brief Overview of Each Chapter

Chapter 1: Introduction to Computers and Java. This chapter provides an introduction to the field of computer science and covers the fundamentals of hardware, software, and programming languages. The elements of a program, such as key words, variables, operators, and punctuation, are discussed by examining a simple program. An overview of entering source code, compiling, and executing a program is presented. A brief history of Java is also given.

Figure P-1 Chapter dependencies

Chapter 2: Java Fundamentals. This chapter gets students started in Java by introducing data types, identifiers, variable declarations, constants, comments, program output, and simple arithmetic operations. The conventions of programming style are also introduced. Students learn to read console input with the `Scanner` class and with dialog boxes using `JOptionPane`.

Chapter 3: Decision Structures. In this chapter students explore relational operators and relational expressions and are shown how to control the flow of a program with the `if`, `if-else`, and `if-else-if` statements. Nested `if` statements, logical operators, the conditional operator, and the `switch` statement are also covered. The chapter discusses how to compare `String` objects with the `equals`, `compareTo`, `equalsIgnoreCase`, and `compareToIgnoreCase` methods. Formatting numeric output with the `System.out.printf` method and the `String.format` method is discussed.

Chapter 4: Loops and Files. This chapter covers Java’s repetition control structures. The `while` loop, `do-while` loop, and `for` loop are taught, along with common uses for these devices. Counters, accumulators, running totals, sentinels, and other application-related topics are discussed. Simple file operations for reading and writing text files are included.

Chapter 5: Methods. In this chapter students learn how to write `void` methods, value-returning methods, and methods that do and do not accept arguments. The concept of functional decomposition is discussed.

Chapter 6: A First Look at Classes. This chapter introduces students to designing classes for the purpose of instantiating objects. Students learn about class fields and methods, and UML diagrams are introduced as a design tool. Then constructors and overloading are discussed. A `BankAccount` class is presented as a case study, and a section on object-oriented design is included. This section leads the students through the process of identifying classes and their responsibilities within a problem domain. There is also a section that briefly explains packages and the `import` statement.

Chapter 7: Arrays and the `ArrayList` Class. In this chapter students learn to create and work with single and multi-dimensional arrays. Numerous array-processing techniques are demonstrated, such as summing the elements in an array, finding the highest and lowest values, and sequentially searching an array. Other topics, including ragged arrays and variable-length arguments (`varargs`), are also discussed. The `ArrayList` class is introduced, and Java’s generic types are briefly discussed and demonstrated.

Chapter 8: A Second Look at Classes and Objects. This chapter shows students how to write classes with added capabilities. Static methods and fields, interaction between objects, passing objects as arguments, and returning objects from methods are discussed. Aggregation and the “has a” relationship is covered, as well as enumerated types. A section on object-oriented design shows how to use CRC cards to determine the collaborations among classes.

Chapter 9: Text Processing and More about Wrapper Classes. This chapter discusses the numeric and `Character` wrapper classes. Methods for converting numbers to strings, testing the case of characters, and converting the case of characters are covered. Autoboxing and unboxing are also discussed. More `String` class methods are covered, including using the `split` method to tokenize strings. The chapter also covers the `StringBuilder` and `StringTokenizer` classes.

Chapter 10: Inheritance. The study of classes continues in this chapter with the subjects of inheritance and polymorphism. The topics covered include superclasses, subclasses, how constructors work in inheritance, method overriding, polymorphism and dynamic binding, protected and package access, class hierarchies, abstract classes, abstract methods, anonymous inner classes, interfaces, and lambda expressions.

Chapter 11: Exceptions and Advanced File I/O. In this chapter students learn to develop enhanced error trapping techniques using exceptions. Handling exceptions is covered, as well as developing and throwing custom exceptions. The chapter discusses advanced techniques for working with sequential access, random access, text, and binary files.

Chapter 12: A First Look at GUI Applications. This chapter presents the basics of developing GUI applications with Swing. Fundamental Swing components and the basic concepts of event-driven programming are covered.

Chapter 13: Advanced GUI Applications. This chapter continues the study of GUI application development with Swing. More advanced components, menu systems, and look-and-feel are covered.

Chapter 14: Applets and More. In this chapter students apply their knowledge of GUI development to the creation of applets. In addition to using Swing applet classes, AWT classes are discussed for portability. Drawing simple graphical shapes is discussed.

Chapter 15: Creating GUI Applications with JavaFX and Scene Builder. This chapter introduces JavaFX, which is the next generation library for creating graphical applications in Java. This chapter also shows how to use Scene Builder, a free screen designer from Oracle, to visually design GUIs. This chapter is written in such a way that it is independent from the existing chapters on Swing and AWT. You can choose to skip chapters 12, 13, and 14, and go straight to Chapter 15, or cover all of the GUI chapters.

Chapter 16: Recursion. This chapter presents recursion as a problem-solving technique. Numerous examples of recursive methods are demonstrated.

Chapter 17: Databases. This chapter introduces the student to database programming. The basic concepts of database management systems and SQL are first introduced. Then the student learns to use JDBC to write database applications in Java. Relational data is covered, and numerous example programs are presented throughout the chapter.

Features of the Text

Concept Statements. Each major section of the text starts with a concept statement that concisely summarizes the focus of the section.

Example Programs. The text has an abundant number of complete and partial example programs, each designed to highlight the current topic. In most cases the programs are practical, real-world examples.

Program Output. Each example program is followed by a sample of its output, which shows students how the program functions.



Checkpoints. Checkpoints, highlighted by the checkmark icon, appear at intervals throughout each chapter. They are designed to check students' knowledge soon after learning a new topic. Answers for all Checkpoint questions are provided in Appendix K, which can be downloaded from the book's resource page at www.pearsonhighered.com/cs-resources.



NOTE: Notes appear at several places throughout the text. They are short explanations of interesting or often misunderstood points relevant to the topic at hand.

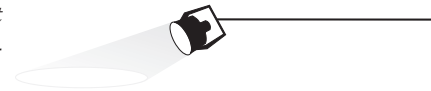


TIP: Tips advise the student on the best techniques for approaching different programming problems and appear regularly throughout the text.



WARNING! Warnings caution students about certain Java features, programming techniques, or practices that can lead to malfunctioning programs or lost data.

In the Spotlight. Many of the chapters provide an *In the Spotlight* section that presents a programming problem, along with detailed, step-by-step analysis showing the student how to solve it.



VideoNotes. A series of videos, developed specifically for this book, are available at www.pearsonglobaleditions.com/Gaddis. Icons appear throughout the text alerting the student to videos about specific topics.

Case Studies. Case studies that simulate real-world business applications are introduced throughout the text and are provided on the book's resource page at www.pearsonglobaleditions.com/Gaddis.

Common Errors to Avoid. Each chapter provides a list of common errors and explanations of how to avoid them.

Review Questions and Exercises. Each chapter presents a thorough and diverse set of review questions and exercises. They include Multiple Choice and True/False, Find the Error, Algorithm Workbench, and Short Answer.

Programming Challenges. Each chapter offers a pool of programming challenges designed to solidify students' knowledge of topics at hand. In most cases the assignments present real-world problems to be solved.

Supplements

Student Online Resources

Many student resources are available for this book from the publisher. The following items are available on the Gaddis Series resource page at www.pearsonglobaleditions.com/Gaddis:

- The source code for each example program in the book
- Access to the book's companion VideoNotes
- Appendixes A–M (listed in the Contents)
- A collection of seven valuable Case Studies (listed in the Contents)
- Links to download the Java™ Edition Development Kit
- Links to download numerous programming environments including jGRASP™, Eclipse™, TextPad™, NetBeans™, JCreator, and DrJava

Online Practice and Assessment with MyProgrammingLab

MyProgrammingLab helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyProgrammingLab improves the programming competence of beginning students, who often struggle with the basic concepts and paradigms of popular high-level programming languages. A self-study and homework tool, the MyProgrammingLab course consists of hundreds of small practice exercises organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of their code submissions and offers targeted hints that enable students to figure out what went wrong—and why. For instructors, a comprehensive gradebook tracks correct and incorrect answers and stores the code inputted by students for review.

MyProgrammingLab is offered to users of this book in partnership with Turing's Craft, the makers of the CodeLab interactive programming exercise system. For a full demonstration, to see feedback from instructors and students, or to get started using MyProgrammingLab in your course, visit www.myprogramminglab.com.

Instructor Resources

The following supplements are available to qualified instructors:

- Answers to all of the Review Questions
- Solutions for the Programming Challenges
- PowerPoint Presentation slides for each chapter
- Computerized Test Banks
- Source Code
- Lab Manual
- Student Files for the Lab Manual
- Solutions to the Lab Manual

Visit the Pearson Instructor Resource Center (www.pearsonglobaleditions.com/Gaddis) or contact your local Pearson representative for information on how to access these resources.

Acknowledgments

There have been many helping hands in the development and publication of this book. We would like to thank the following faculty reviewers for their helpful suggestions and expertise:

Reviewers For This Edition

Carl Stephen Guynes
University of North Texas

Alan G. Jackson
Oakland Community College

Zhen Jiang
West Chester University

Neven Jurkovic
Palo Alto College

Dennis Lang
Kansas State University

Jiang Li
Austin Peay State University

Cheng Luo
Coppin State University

Felix Rodriguez
Naugatuck Valley Community College

Diane Rudolph
John A Logan College

Timothy Urness
Drake University

Zijiang Yang
Western Michigan University

Reviewers of Previous Editions

Ahmad Abuhejleh
University of Wisconsin, River Falls

Colin Archibald
Valencia Community College

Ijaz Awani
Savannah State University

Bill Bane
Tarleton State University

N. Dwight Barnette
Virginia Tech

Asoke Bhattacharyya
Saint Xavier University, Chicago

Marvin Bishop
Manhattan College

Heather Booth
University of Tennessee, Knoxville

David Boyd
Valdosta University

Julius Brandstatter
Golden Gate University

Kim Cannon
Greenville Tech

Jesse Cecil
College of the Siskiyous

James Chegwidan
Tarrant County College

Kay Chen
Bucks County Community College

Brad Chilton
Tarleton State University

Diane Christie
University of Wisconsin, Stout

Cara Cocking
Marquette University

Jose Cordova
University of Louisiana, Monroe

Walter C. Daugherity
Texas A & M University

Michael Doherty
University of the Pacific

Jeanne M. Douglas
University of Vermont

Sander Eller
*California Polytechnic University,
Pomona*

Brooke Estabrook-Fishinghawk
Mesa Community College

Mike Fry
Lebanon Valley College

David Goldschmidt
College of St. Rose

Georgia R. Grant
College of San Mateo

Nancy Harris
James Madison University

Chris Haynes
Indiana University

Ric Heishman
Northern Virginia Community College

Deedee Herrera
Dodge City Community College

Mary Hovik
Lehigh Carbon Community College

Brian Howard
DePauw University

Alan Jackson
Oakland Community College (MI)

Norm Jacobson
University of California, Irvine

Stephen Judd
University of Pennsylvania

Harry Lichtbach
Evergreen Valley College

Michael A. Long
California State University, Chico

Tim Margush
University of Akron

Blayne E. Mayfield
Oklahoma State University

Scott McLeod
Riverside Community College

Dean Mellas
Cerritos College

Georges Merx
San Diego Mesa College

Martin Meyers
California State University, Sacramento

Pati Milligan
Baylor University

Laurie Murphy
Pacific Lutheran University

Steve Newberry
Tarleton State University

Lynne O'Hanlon
Los Angeles Pierce College

Merrill Parker
*Chattanooga State Technical
Community College*

Bryson R. Payne
*North Georgia College and State
University*

Rodney Pearson
Mississippi State University

Peter John Polito
Springfield College

Charles Robert Putnam
*California State University,
Northridge*

Y. B. Reddy
Grambling State University

Elizabeth Riley
Macon State College

Carolyn Schauble
Colorado State University

Bonnie Smith
Fresno City College

Daniel Spiegel
Kutztown University

Caroline St. Clair
North Central College

Karen Stanton
Los Medanos College

Peter van der Goes
Rose State College

Tuan A Vo
Mt. San Antonio College

Xiaoying Wang
University of Mississippi

Yu Wu
University of North Texas

I also want to thank everyone at Pearson for making the *Starting Out With . . .* series so successful. I have worked so closely with the team at Pearson that I consider them among my closest friends. I am extremely fortunate to have Matt Goldstein as my editor, and Kelsey Loanes as Editorial Assistant. They have guided me through the process of revising this book, as well as many others. I am also fortunate to have Demetrius Hall and Bram Van Kempen as Marketing Managers. Their hard work is truly inspiring, and they do a great job getting my books out to the academic community. The production team, led by Camille Trentacoste, worked tirelessly to make this book a reality. Thanks to you all!

About the Author

Tony Gaddis is the principal author of the *Starting Out With . . .* series of textbooks. He has nearly two decades of experience teaching computer science courses, primarily at Haywood Community College. Tony is a highly acclaimed instructor who was previously selected as the North Carolina Community College “Teacher of the Year” and has received the Teaching Excellence award from the National Institute for Staff and Organizational Development. The *Starting Out With . . .* series includes introductory textbooks covering programming logic and design, C++, Java™, Microsoft® Visual Basic®, Microsoft® Visual C#, Python, Alice, and App Inventor, all published by Pearson.

Pearson wishes to thank and acknowledge the following people for their work on the Global Edition:

Contributor

Ela Kashyap
Amity University

Reviewers

Muthuraj M
Android developer

Arup Kumar Bhattacharjee
RCC Institute of Information Technology

Soumen Mukherjee
RCC Institute of Information Technology

Mohit P. Tahliliani
*National Institute of Technology
Karnataka*

Harsh Bhasin
Jamia Hamdard



get with the programming

Through the power of practice and immediate personalized feedback, MyProgrammingLab improves your performance.

MyProgrammingLabTM

Learn more at www.myprogramminglab.com

This page intentionally left blank

STARTING OUT WITH

JAVATM



From Control Structures
through Objects

This page intentionally left blank

Introduction to Computers and Java

TOPICS

- | | |
|---|---------------------------------|
| 1.1 Introduction | 1.4 Programming Languages |
| 1.2 Why Program? | 1.5 What Is a Program Made Of? |
| 1.3 Computer Systems: Hardware and Software | 1.6 The Programming Process |
| | 1.7 Object-Oriented Programming |

1.1 Introduction

This book teaches programming using Java. Java is a powerful language that runs on practically every type of computer. It can be used to create large applications or small programs that are part of a Web site. Before plunging right into learning Java, however, this chapter will review the fundamentals of computer hardware and software, and then take a broad look at computer programming in general.

1.2 Why Program?

CONCEPT: Computers can do many different jobs because they are programmable.

Every profession has tools that make the job easier to do. Carpenters use hammers, saws, and measuring tapes. Mechanics use wrenches, screwdrivers, and ratchets. Electronics technicians use probes, scopes, and meters. Some tools are unique and can be categorized as belonging to a single profession. For example, surgeons have certain tools that are designed specifically for surgical operations. Those tools probably aren't used by anyone other than surgeons. There are some tools, however, that are used in several professions. Screwdrivers, for instance, are used by mechanics, carpenters, and many others.

The computer is a tool used by so many professions that it cannot be easily categorized. It can perform so many different jobs that it is perhaps the most versatile tool ever made. To the accountant, computers balance books, analyze profits and losses, and prepare tax reports. To the factory worker, computers control manufacturing machines and track production. To the mechanic, computers analyze the various systems in an automobile and pinpoint hard-to-find problems. The computer can do such a wide variety of tasks because it can

be *programmed*. It is a machine specifically designed to follow instructions. Because of the computer's programmability, it doesn't belong to any single profession. Computers are designed to do whatever job their programs, or *software*, tell them to do.

Computer programmers do a very important job. They create software that transforms computers into the specialized tools of many trades. Without programmers, the users of computers would have no software, and without software, computers would not be able to do anything.

Computer programming is both an art and a science. It is an art because every aspect of a program should be carefully designed. Here are a few of the things that must be designed for any real-world computer program:

- The logical flow of the instructions
- The mathematical procedures
- The layout of the programming statements
- The appearance of the screens
- The way information is presented to the user
- The program's "user friendliness"
- Manuals, help systems, and/or other forms of written documentation

There is also a science to programming. Because programs rarely work right the first time they are written, a lot of analyzing, experimenting, correcting, and redesigning is required. This demands patience and persistence of the programmer. Writing software demands discipline as well. Programmers must learn special languages such as Java because computers do not understand English or other human languages. Programming languages have strict rules that must be carefully followed.

Both the artistic and scientific nature of programming makes writing computer software like designing a car: Both cars and programs should be functional, efficient, powerful, easy to use, and pleasing to look at.

1.3

Computer Systems: Hardware and Software

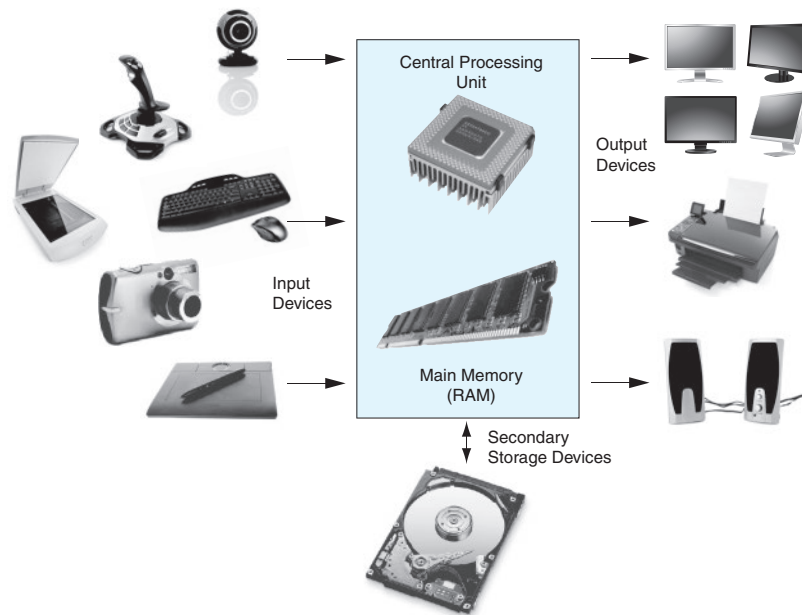
CONCEPT: All computer systems consist of similar hardware devices and software components.

Hardware

Hardware refers to the physical components that a computer is made of. A computer, as we generally think of it, is not an individual device, but a system of devices. Like the instruments in a symphony orchestra, each device plays its own part. A typical computer system consists of the following major components:

- The central processing unit (CPU)
- Main memory
- Secondary storage devices
- Input devices
- Output devices

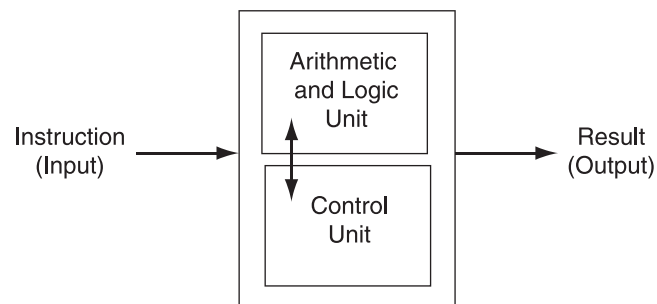
The organization of a computer system is shown in Figure 1-1.

Figure 1-1 The organization of a computer system

Let's take a closer look at each of these devices.

The CPU

At the heart of a computer is its *central processing unit*, or *CPU*. The CPU's job is to fetch instructions, follow the instructions, and produce some resulting data. Internally, the central processing unit consists of two parts: the *control unit* and the *arithmetic and logic unit (ALU)*. The control unit coordinates all of the computer's operations. It is responsible for determining where to get the next instruction and regulating the other major components of the computer with control signals. The arithmetic and logic unit, as its name suggests, is designed to perform mathematical operations. The organization of the CPU is shown in Figure 1-2.

Figure 1-2 The organization of the CPU

A program is a sequence of instructions stored in the computer's memory. When a computer is running a program, the CPU is engaged in a process known formally as the *fetch/decode/execute cycle*. The steps in the fetch/decode/execute cycle are as follows:

- Fetch* The CPU’s control unit fetches, from main memory, the next instruction in the sequence of program instructions.
- Decode* The instruction is encoded in the form of a number. The control unit decodes the instruction and generates an electronic signal.
- Execute* The signal is routed to the appropriate component of the computer (such as the ALU, a disk drive, or some other device). The signal causes the component to perform an operation.

These steps are repeated as long as there are instructions to perform.

Main Memory

Commonly known as *random access memory*, or *RAM*, the computer’s main memory is a device that holds information. Specifically, RAM holds the sequences of instructions in the programs that are running and the data those programs are using.

Memory is divided into sections that hold an equal amount of data. Each section is made of eight “switches” that may be either on or off. A switch in the on position usually represents the number 1, whereas a switch in the off position usually represents the number 0. The computer stores data by setting the switches in a memory location to a pattern that represents a character or a number. Each of these switches is known as a *bit*, which stands for *binary digit*. Each section of memory, which is a collection of eight bits, is known as a *byte*. Each byte is assigned a unique number known as an *address*. The addresses are ordered from lowest to highest. A byte is identified by its address in much the same way a post office box is identified by an address. Figure 1-3 shows a series of bytes with their addresses. In the illustration, sample data is stored in memory. The number 149 is stored in the byte at address 16, and the number 72 is stored in the byte at address 23.

RAM is usually a volatile type of memory, used only for temporary storage. When the computer is turned off, the contents of RAM are erased.

Figure 1-3 Memory bytes and their addresses

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16149	17	18	19
20	21	22	2372	24	25	26	27	28	29

Secondary Storage

Secondary storage is a type of memory that can hold data for long periods of time—even when there is no power to the computer. Frequently used programs are stored in secondary memory and loaded into main memory as needed. Important data, such as word processing documents, payroll data, and inventory figures, is saved to secondary storage as well.

The most common type of secondary storage device is the *disk drive*. A traditional disk drive stores data by magnetically encoding it onto a spinning circular disk. *Solid state drives*, which store data in solid-state memory, are increasingly becoming popular. A solid-state drive has no moving parts, and operates faster than a traditional disk drive.

Most computers have some sort of secondary storage device, either a traditional disk drive or a solid-state drive, mounted inside their case. External drives are also available, which connect to one of the computer's communication ports. External drives can be used to create backup copies of important data or to move data to another computer.

In addition to external drives, many types of devices have been created for copying data, and for moving it to other computers. *Universal Serial Bus drives*, or *USB drives* are small devices that plug into the computer's USB (Universal Serial Bus) port, and appear to the system as a disk drive. These drives do not actually contain a disk, however. They store data in a special type of memory known as *flash memory*. USB drives are inexpensive, reliable, and small enough to be carried in your pocket.

Optical devices such as the *CD* (compact disc) and the *DVD* (digital versatile disc) are also popular for data storage. Data is not recorded magnetically on an optical disc, but is encoded as a series of pits on the disc surface. CD and DVD drives use a laser to detect the pits and thus read the encoded data. Optical discs hold large amounts of data, and because recordable CD and DVD drives are now commonplace, they make a good medium for creating backup copies of data.

Input Devices

Input is any data the computer collects from the outside world. The device that collects the data and sends it to the computer is called an *input device*. Common input devices are the keyboard, mouse, scanner, and digital camera. Disk drives, optical drives, and USB drives can also be considered input devices because programs and data are retrieved from them and loaded into the computer's memory.

Output Devices

Output is any data the computer sends to the outside world. It might be a sales report, a list of names, or a graphic image. The data is sent to an output device, which formats and presents it. Common output devices are monitors and printers. Disk drives, USB drives, and CD recorders can also be considered output devices because the CPU sends data to them to be saved.

Software

As previously mentioned, software refers to the programs that run on a computer. There are two general categories of software: operating systems and application software. An operating system is a set of programs that manages the computer's hardware devices and controls their processes. Most all modern operating systems are multitasking, which means they are capable of running multiple programs at once. Through a technique called time sharing, a multitasking system divides the allocation of hardware resources and the attention of the CPU among all the executing programs. UNIX, Linux, Mac OS, and Windows are multitasking operating systems.

Application software refers to programs that make the computer useful to the user. These programs solve specific problems or perform general operations that satisfy the needs of the user. Word processing, spreadsheet, and database packages are all examples of application software.

**Checkpoint**MyProgrammingLab™ www.myprogramminglab.com

- 1.1 Why is the computer used by so many different people, in so many different professions?
- 1.2 List the five major hardware components of a computer system.
- 1.3 Internally, the CPU consists of what two units?
- 1.4 Describe the steps in the fetch/decode/execute cycle.
- 1.5 What is a memory address? What is its purpose?
- 1.6 Explain why computers have both main memory and secondary storage.
- 1.7 What does the term *multitasking* mean?

1.4 Programming Languages

CONCEPT: A program is a set of instructions a computer follows in order to perform a task. A programming language is a special language used to write computer programs.

What Is a Program?

Computers are designed to follow instructions. A computer program is a set of instructions that enable the computer to solve a problem or perform a task. For example, suppose we want the computer to calculate someone's gross pay. The following is a list of things the computer should do to perform this task.

1. Display a message on the screen: "How many hours did you work?"
2. Allow the user to enter the number of hours worked.
3. Once the user enters a number, store it in memory.
4. Display a message on the screen: "How much do you get paid per hour?"
5. Allow the user to enter an hourly pay rate.
6. Once the user enters a number, store it in memory.
7. Once both the number of hours worked and the hourly pay rate are entered, multiply the two numbers and store the result in memory.
8. Display a message on the screen that shows the amount of money earned. The message must include the result of the calculation performed in Step 7.

Collectively, these instructions are called an *algorithm*. An algorithm is a set of well-defined steps for performing a task or solving a problem. Notice that these steps are sequentially ordered. Step 1 should be performed before Step 2, and so forth. It is important that these instructions be performed in their proper sequence.

Although you and I might easily understand the instructions in the pay-calculating algorithm, it is not ready to be executed on a computer. A computer's CPU can only process instructions that are written in *machine language*. If you were to look at a machine language program, you would see a stream of binary numbers (numbers consisting of only 1s and 0s). The binary numbers form machine language instructions, which the CPU interprets as commands. Here is an example of what a machine language instruction might look like:

```
1011010000000101
```

As you can imagine, the process of encoding an algorithm in machine language is very tedious and difficult. In addition, each different type of CPU has its own machine language. If you wrote a machine language program for computer A and then wanted to run it on computer B, which has a different type of CPU, you would have to rewrite the program in computer B's machine language.

Programming languages, which use words instead of numbers, were invented to ease the task of programming. A program can be written in a programming language, which is much easier to understand than machine language, and then translated into machine language. Programmers use software to perform this translation. Many programming languages have been created. Table 1-1 lists a few of the well-known ones.

Table 1-1 Programming languages

Language	Description
BASIC	Beginners All-purpose Symbolic Instruction Code is a general-purpose, procedural programming language. It was originally designed to be simple enough for beginners to learn.
FORTRAN	FORmula TRANslator is a procedural language designed for programming complex mathematical algorithms.
COBOL	Common Business-Oriented Language is a procedural language designed for business applications.
Pascal	Pascal is a structured, general-purpose, procedural language designed primarily for teaching programming.
C	C is a structured, general-purpose, procedural language developed at Bell Laboratories.
C++	Based on the C language, C++ offers object-oriented features not found in C. C++ was also invented at Bell Laboratories.
C#	Pronounced “C sharp.” It is a language invented by Microsoft for developing applications based on the Microsoft .NET platform.
Java	Java is an object-oriented language invented at Sun Microsystems, and is now owned by Oracle. It may be used to develop stand-alone applications that operate on a single computer, applications that run over the Internet from a Web server, and applets that run in a Web browser.
JavaScript	JavaScript is a programming language that can be used in a Web site to perform simple operations. Despite its name, JavaScript is not related to Java.
Perl	A general-purpose programming language used widely on Internet servers.
PHP	A programming language used primarily for developing Web server applications and dynamic Web pages.
Python	Python is an object-oriented programming language used in both business and academia. Many popular Web sites contain features developed in Python.
Ruby	Ruby is a simple but powerful object-oriented programming language. It can be used for a variety of purposes, from small utility programs to large Web applications.
Visual Basic	Visual Basic is a Microsoft programming language and software development environment that allows programmers to create Windows-based applications quickly.

A History of Java

In 1991 a team was formed at Sun Microsystems (a company that is now owned by Oracle) to speculate about the important technological trends that might emerge in the near future. The team, which was named the Green Team, concluded that computers would merge with consumer appliances. Their first project was to develop a handheld device named *7 (pronounced star seven) that could be used to control a variety of home entertainment devices. For the unit to work, it had to use a programming language that could be processed by all the devices it controlled. This presented a problem because different brands of consumer devices use different processors, each with its own machine language.

Because no such universal language existed, James Gosling, the team's lead engineer, created one. Programs written in this language, which was originally named Oak, were not translated into the machine language of a specific processor, but were translated into an intermediate language known as *byte code*. Another program would then translate the byte code into machine language that could be executed by the processor in a specific consumer device.

Unfortunately, the technology developed by the Green Team was ahead of its time. No customers could be found, mostly because the computer-controlled consumer appliance industry was just beginning. But rather than abandoning their hard work and moving on to other projects, the team saw another opportunity: the Internet. The Internet is a perfect environment for a universal programming language such as Oak. It consists of numerous different computer platforms connected together in a single network.

To demonstrate the effectiveness of its language, which was renamed Java, the team used it to develop a Web browser. The browser, named HotJava, was able to download and run small Java programs known as applets. This gave the browser the capability to display animation and interact with the user. HotJava was demonstrated at the 1995 SunWorld conference before a wowed audience. Later the announcement was made that Netscape would incorporate Java technology into its Navigator browser. Other Internet companies rapidly followed, increasing the acceptance and the influence of the Java language. Today, Java is very popular for developing not only applets for the Internet but also stand-alone applications.

Java Applications and Applets

There are two types of programs that may be created with Java: applications and applets. An application is a stand-alone program that runs on your computer. You have probably used several applications already, such as word processors, spreadsheets, database managers, and graphics programs. Although Java may be used to write these types of applications, other languages such as C, C++, and Visual Basic are also used.

In the previous section you learned that Java may also be used to create applets. The term *applet* refers to a small application, in the same way that the term *piglet* refers to a small pig. Unlike applications, an applet is designed to be transmitted over the Internet from a Web server, and then executed in a Web browser. Applets are important because they can be used to extend the capabilities of a Web page significantly.

Web pages are normally written in Hypertext Markup Language (HTML). HTML is limited, however, because it merely describes the content and layout of a Web page. HTML does not have sophisticated abilities such as performing math calculations and interacting with the user. A Web designer can write a Java applet to perform operations that are

normally performed by an application and embed it in a Web site. When someone visits the Web site, the applet is downloaded to the visitor's browser and executed.

Security

Any time content is downloaded from a Web server to a visitor's computer, security is an important concern. Because Java is a full-featured programming language, at first you might be suspicious of any Web site that transmits an applet to your computer. After all, couldn't a Java applet do harmful things, such as deleting the contents of the disk drive or transmitting private information to another computer? Fortunately, the answer is no. Web browsers run Java applets in a secure environment within your computer's memory and do not allow them to access resources, such as a disk drive, that are outside that environment.

1.5

What Is a Program Made Of?

CONCEPT: There are certain elements that are common to all programming languages.

Language Elements

All programming languages have some things in common. Table 1-2 lists the common elements you will find in almost every language.

Table 1-2 The common elements of a programming language

Language Element	Description
Key Words	These are words that have a special meaning in the programming language. They may be used for their intended purpose only. Key words are also known as <i>reserved words</i> .
Operators	Operators are symbols or words that perform operations on one or more operands. An operand is usually an item of data, such as a number.
Punctuation	Most programming languages require the use of punctuation characters. These characters serve specific purposes, such as marking the beginning or ending of a statement, or separating items in a list.
Programmer-Defined Names	Unlike key words, which are part of the programming language, these are words or names that are defined by the programmer. They are used to identify storage locations in memory and parts of the program that are created by the programmer. Programmer-defined names are often called <i>identifiers</i> .
Syntax	These are rules that must be followed when writing a program. Syntax dictates how key words and operators may be used, and where punctuation symbols must appear.

Let’s look at an example Java program and identify an instance of each of these elements. Code Listing 1-1 shows the code listing with each line numbered.



NOTE: The line numbers are not part of the program. They are included to help point out specific parts of the program.

Code Listing 1-1 Payroll.java

```
1 public class Payroll
2 {
3     public static void main(String[] args)
4     {
5         int hours = 40;
6         double grossPay, payRate = 25.0;
7
8         grossPay = hours * payRate;
9         System.out.println("Your gross pay is $" + grossPay);
10    }
11 }
```

Key Words (Reserved Words)

Two of Java’s key words appear in line 1: `public` and `class`. In line 3, the words `public`, `static`, and `void` are all key words. The words `int` in line 5 and `double` in line 6 are also key words. These words, which are always written in lowercase, each have a special meaning in Java and can only be used for their intended purpose. As you will see, the programmer is allowed to make up his or her own names for certain things in a program. Key words, however, are reserved and cannot be used for anything other than their designated purpose. Part of learning a programming language is learning the commonly used key words, what they mean, and how to use them.

Table 1-3 shows a list of the Java key words.

Table 1-3 The Java key words

<code>abstract</code>	<code>const</code>	<code>final</code>	<code>int</code>	<code>public</code>	<code>throw</code>
<code>assert</code>	<code>continue</code>	<code>finally</code>	<code>interface</code>	<code>return</code>	<code>throws</code>
<code>boolean</code>	<code>default</code>	<code>float</code>	<code>long</code>	<code>short</code>	<code>transient</code>
<code>break</code>	<code>do</code>	<code>for</code>	<code>native</code>	<code>static</code>	<code>true</code>
<code>byte</code>	<code>double</code>	<code>goto</code>	<code>new</code>	<code>strictfp</code>	<code>try</code>
<code>case</code>	<code>else</code>	<code>if</code>	<code>null</code>	<code>super</code>	<code>void</code>
<code>catch</code>	<code>enum</code>	<code>implements</code>	<code>package</code>	<code>switch</code>	<code>volatile</code>
<code>char</code>	<code>extends</code>	<code>import</code>	<code>private</code>	<code>synchronized</code>	<code>while</code>
<code>class</code>	<code>false</code>	<code>instanceof</code>	<code>protected</code>	<code>this</code>	

Programmer-Defined Names

The words `hours`, `payRate`, and `grossPay` that appear in the program in lines 5, 6, 8, and 9 are programmer-defined names. They are not part of the Java language but are names made up by the programmer. In this particular program, these are the names of variables. As you will learn later in this chapter, variables are the names of memory locations that may hold data.

Operators

In line 8 the following line appears:

```
grossPay = hours * payRate;
```

The `=` and `*` symbols are both operators. They perform operations on items of data, known as operands. The `*` operator multiplies its two operands, which in this example are the variables `hours` and `payRate`. The `=` symbol is called the assignment operator. It takes the value of the expression that appears at its right and stores it in the variable whose name appears at its left. In this example, the `=` operator stores in the `grossPay` variable the result of the `hours` variable multiplied by the `payRate` variable. In other words, the statement says, “the `grossPay` variable is assigned the value of `hours` times `payRate`.”

Punctuation

Notice that lines 5, 6, 8, and 9 end with a semicolon. A semicolon in Java is similar to a period in English: It marks the end of a complete sentence (or statement, as it is called in programming jargon). Semicolons do not appear at the end of every line in a Java program, however. There are rules that govern where semicolons are required and where they are not. Part of learning Java is learning where to place semicolons and other punctuation symbols.

Lines and Statements

Often, the contents of a program are thought of in terms of lines and statements. A *line* is just that—a single line as it appears in the body of a program. Code Listing 1-1 is shown with each of its lines numbered. Most of the lines contain something meaningful; however, line 7 is empty. Blank lines are only used to make a program more readable.

A statement is a complete instruction that causes the computer to perform some action. Here is the statement that appears in line 9 of Code Listing 1-1:

```
System.out.println("Your gross pay is $" + grossPay);
```

This statement causes the computer to display a message on the screen. Statements can be a combination of key words, operators, and programmer-defined names. Statements often occupy only one line in a program, but sometimes they are spread out over more than one line.

Variables

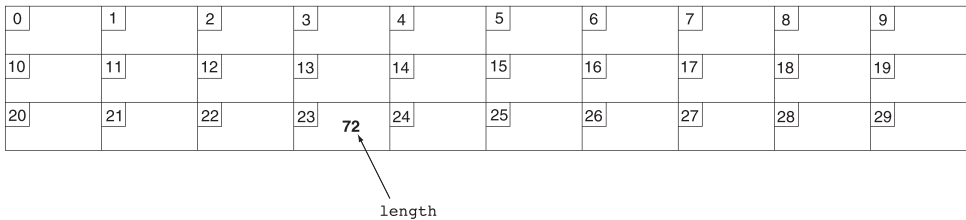
The most fundamental way that a Java program stores an item of data in memory is with a variable. A *variable* is a named storage location in the computer’s memory. The data stored in a variable may change while the program is running (hence the name “variable”). Notice that in Code Listing 1-1 the programmer-defined names `hours`, `payRate`, and `grossPay`

appear in several places. All three of these are the names of variables. The `hours` variable is used to store the number of hours the user has worked. The `payRate` variable stores the user's hourly pay rate. The `grossPay` variable holds the result of hours multiplied by `payRate`, which is the user's gross pay.

Variables are symbolic names made up by the programmer that represent locations in the computer's RAM. When data is stored in a variable, it is actually stored in RAM. Assume that a program has a variable named `length`. Figure 1-4 illustrates the way the variable name represents a memory location.

In Figure 1-4, the variable `length` is holding the value 72. The number 72 is actually stored in RAM at address 23, but the name `length` symbolically represents this storage location. If it helps, you can think of a variable as a box that holds data. In Figure 1-4, the number 72 is stored in the box named `length`. Only one item may be stored in the box at any given time. If the program stores another value in the box, it will take the place of the number 72.

Figure 1-4 A variable name represents a location in memory



The Compiler and the Java Virtual Machine

When a Java program is written, it must be typed into the computer and saved to a file. A *text editor*, which is similar to a word processing program, is used for this task. The Java programming statements written by the programmer are called *source code*, and the file they are saved in is called a *source file*. Java source files end with the *.java* extension.

After the programmer saves the source code to a file, he or she runs the Java compiler. A *compiler* is a program that translates source code into an executable form. During the translation process, the compiler uncovers any syntax errors that may be in the program. *Syntax errors* are mistakes that the programmer has made that violate the rules of the programming language. These errors must be corrected before the compiler can translate the source code. Once the program is free of syntax errors, the compiler creates another file that holds the translated instructions.

Most programming language compilers translate source code directly into files that contain machine language instructions. These are called *executable files* because they may be executed directly by the computer's CPU. The Java compiler, however, translates a Java source file into a file that contains byte code instructions. Byte code instructions are not machine language, and therefore cannot be directly executed by the CPU. Instead, they are executed by the *Java Virtual Machine* (JVM). The JVM is a program that reads Java byte code instructions and executes them as they are read. For this reason, the JVM is often called an interpreter, and Java is often referred to as an interpreted language. Figure 1-5 illustrates the process of writing a Java program, compiling it to byte code, and running it.

Although Java byte code is not machine language for a CPU, it can be considered as machine language for the JVM. You can think of the JVM as a program that simulates a computer whose machine language is Java byte code.

Portability

The term *portable* means that a program may be written on one type of computer and then run on a wide variety of computers, with little or no modification necessary. Because Java byte code is the same on all computers, compiled Java programs are highly portable. In fact, a compiled Java program may be run on any computer that has a JVM. Figure 1-6 illustrates the concept of a compiled Java program running on Windows, Linux, Mac, and UNIX computers.

With most other programming languages, portability is achieved by the creation of a compiler for each type of computer that the language is to run on. For example, in order for the C++ language to be supported by Windows, Linux, and Mac computers, a separate C++ compiler must be created for each of those environments. Compilers are very complex programs, and more difficult to develop than interpreters. For this reason, a JVM has been developed for many types of computers.

Figure 1-5
Program development process

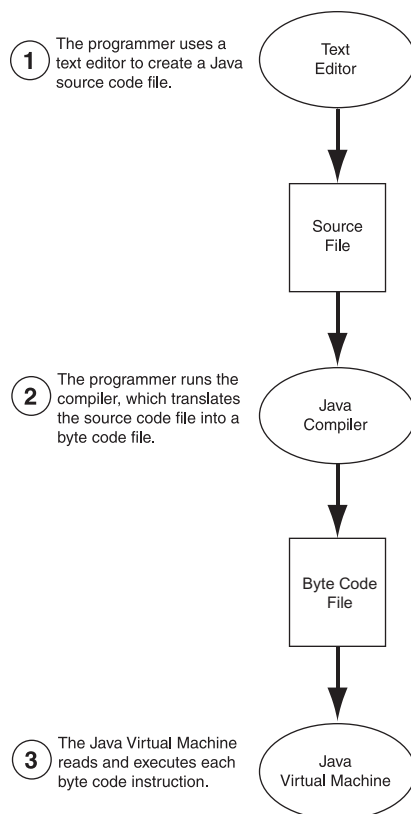
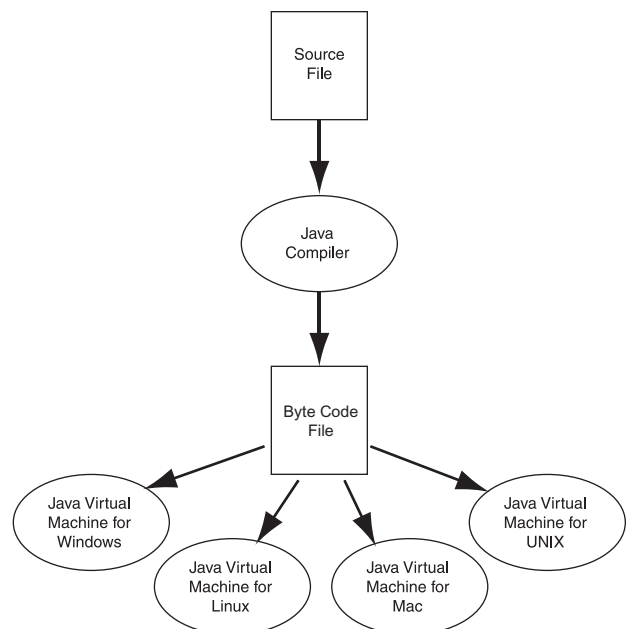


Figure 1-6 Java byte code may be run on any computer with a Java Virtual Machine



Java Software Editions

The software that you use to create Java programs is referred to as the *JDK* (Java Development Kit) or the *SDK* (Software Development Kit). There are the following different editions of the JDK available from Oracle:

- *Java SE*—The Java Standard Edition provides all the essential software tools necessary for writing Java applications and applets.
- *Java EE*—The Java Enterprise Edition provides tools for creating large business applications that employ servers and provide services over the Web.
- *Java ME*—The Java Micro Edition provides a small, highly optimized runtime environment for consumer products such as cell phones, pagers, and appliances.

These editions of Java may be downloaded from Oracle by going to:

<http://java.oracle.com>



NOTE: You can follow the instructions in Appendix E, which can be downloaded from the book's companion Web site, to install the JDK on your system. You can access the book's companion Web site by going to www.pearsonglobaleditions.com/Gaddis.

Compiling and Running a Java Program

Compiling a Java program is a simple process. Once you have installed the JDK, go to your operating system's command prompt.



TIP: In Windows click Start, go to All Programs, and then go to Accessories. Click Command Prompt on the Accessories menu. A command prompt window should open.



VideoNote

Compiling and
Running a Java
Program

At the operating system command prompt, make sure you are in the same directory or folder where the Java program that you want to compile is located. Then, use the `javac` command, in the following form:

```
javac Filename
```

Filename is the name of a file that contains the Java source code. As mentioned earlier, this file has the *.java* extension. For example, if you want to compile the *Payroll.java* file, you would execute the following command:

```
javac Payroll.java
```

This command runs the compiler. If the file contains any syntax errors, you will see one or more error messages and the compiler will not translate the file to byte code. When this happens you must open the source file in a text editor and fix the error. Then you can run the compiler again. If the file has no syntax errors, the compiler will translate it to byte code. Byte code is stored in a file with the *.class* extension, so the byte code for the *Payroll.java* file will be stored in *Payroll.class*, which will be in the same directory or folder as the source file.

To run the Java program, you use the `java` command in the following form:

```
java ClassFilename
```

ClassName is the name of the *.class* file that you wish to execute; however, you do not type the *.class* extension. For example, to run the program that is stored in the *Payroll.class* file, you would enter the following command:

```
java Payroll
```

This command runs the Java interpreter (the JVM) and executes the program.

Integrated Development Environments

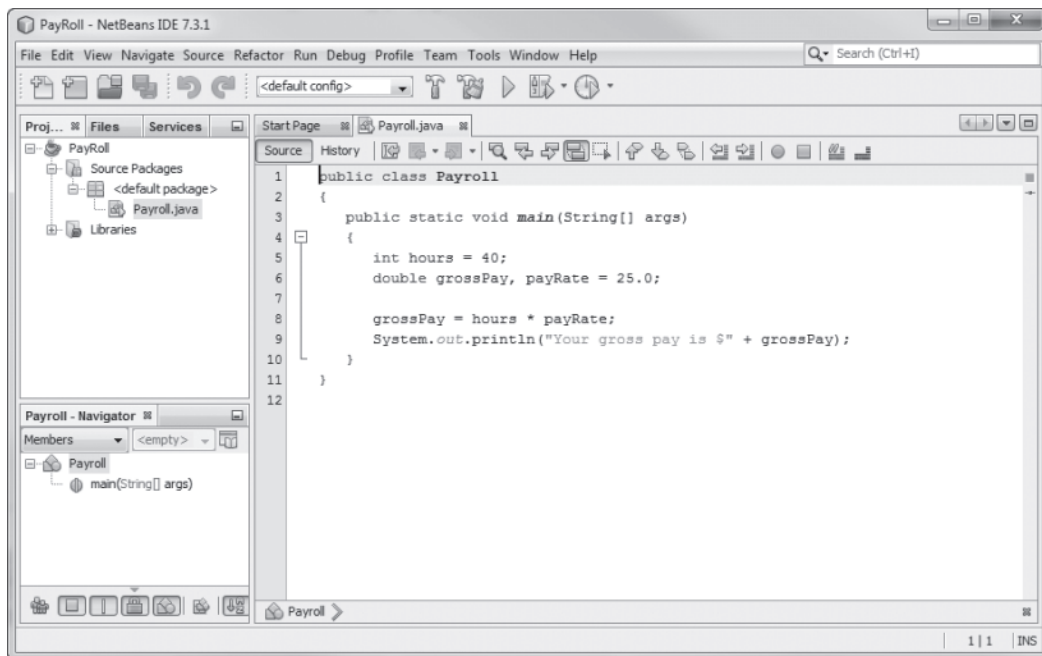


VideoNote

Using an IDE

In addition to the command prompt programs, there are also several Java integrated development environments (IDEs). These environments consist of a text editor, compiler, debugger, and other utilities integrated into a package with a single set of menus. A program is compiled and executed with a single click of a button, or by selecting a single item from a menu. Figure 1-7 shows a screen from the NetBeans IDE.

Figure 1-7 An integrated development environment (IDE) (Oracle Corporate Counsel)



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 1.8 Describe the difference between a key word and a programmer-defined symbol.
- 1.9 Describe the difference between operators and punctuation symbols.

- 1.10 Describe the difference between a program line and a statement.
- 1.11 Why are variables called “variable”?
- 1.12 What happens to a variable’s current contents when a new value is stored there?
- 1.13 What is a compiler?
- 1.14 What is a syntax error?
- 1.15 What is byte code?
- 1.16 What is the JVM?

1.6 The Programming Process

CONCEPT: The programming process consists of several steps, which include design, creation, testing, and debugging activities.

Now that you have been introduced to what a program is, it’s time to consider the process of creating a program. Quite often when inexperienced students are given programming assignments, they have trouble getting started because they don’t know what to do first. If you find yourself in this dilemma, the following steps may help.

1. Clearly define what the program is to do.
2. Visualize the program running on the computer.
3. Use design tools to create a model of the program.
4. Check the model for logical errors.
5. Enter the code and compile it.
6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary.
7. Run the program with test data for input.
8. Correct any runtime errors found while running the program. Repeat Steps 5 through 8 as many times as necessary.
9. Validate the results of the program.

These steps emphasize the importance of planning. Just as there are good ways and bad ways to paint a house, there are good ways and bad ways to create a program. A good program always begins with planning. With the pay-calculating algorithm that was presented earlier in this chapter serving as our example, let’s look at each of the steps in more detail.

1. Clearly define what the program is to do

This step commonly requires you to identify the purpose of the program, the data that is to be input, the processing that is to take place, and the desired output. Let’s examine each of these requirements for the pay-calculating algorithm.

Purpose To calculate the user’s gross pay.

Input Number of hours worked, hourly pay rate.

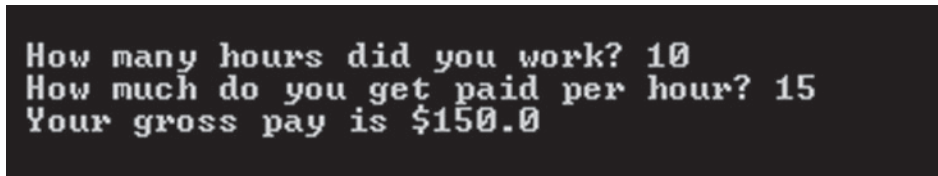
Process Multiply number of hours worked by hourly pay rate. The result is the user’s gross pay.

Output Display a message indicating the user’s gross pay.

2. Visualize the program running on the computer

Before you create a program on the computer, you should first create it in your mind. Try to imagine what the computer screen will look like while the program is running. If it helps, draw pictures of the screen, with sample input and output, at various points in the program. For instance, Figure 1-8 shows the screen we might want produced by a program that implements the pay-calculating algorithm.

Figure 1-8 Screen produced by the pay-calculating algorithm



```
How many hours did you work? 10
How much do you get paid per hour? 15
Your gross pay is $150.0
```

In this step, you must put yourself in the shoes of the user. What messages should the program display? What questions should it ask? By addressing these concerns, you can determine most of the program's output.

3. Use design tools to create a model of the program

While planning a program, the programmer uses one or more design tools to create a model of the program. For example, *pseudocode* is a cross between human language and a programming language and is especially helpful when designing an algorithm. Although the computer can't understand pseudocode, programmers often find it helpful to write an algorithm in a language that's "almost" a programming language, but still very similar to natural language. For example, here is pseudocode that describes the pay-calculating algorithm:

```
Get payroll data.
Calculate gross pay.
Display gross pay.
```

Although this pseudocode gives a broad view of the program, it doesn't reveal all the program's details. A more detailed version of the pseudocode follows:

```
Display "How many hours did you work?"
Input hours.
Display "How much do you get paid per hour?"
Input rate.
Store the value of hours times rate in the pay variable.
Display the value in the pay variable.
```

Notice that the pseudocode uses statements that look more like commands than the English statements that describe the algorithm in Section 1.4. The pseudocode even names variables and describes mathematical operations.

4. Check the model for logical errors

Logical errors are mistakes that cause the program to produce erroneous results. Once a model of the program is assembled, it should be checked for these errors. For example, if pseudocode is used, the programmer should trace through it, checking the logic of each step. If an error is found, the model can be corrected before the next step is attempted.

5. Enter the code and compile it

Once a model of the program has been created, checked, and corrected, the programmer is ready to write source code on the computer. The programmer saves the source code to a file and begins the process of compiling it. During this step the compiler will find any syntax errors that may exist in the program.

6. Correct any errors found during compilation. Repeat Steps 5 and 6 as many times as necessary

If the compiler reports any errors, they must be corrected. Steps 5 and 6 must be repeated until the program is free of compile-time errors.

7. Run the program with test data for input

Once an executable file is generated, the program is ready to be tested for runtime errors. A runtime error is an error that occurs while the program is running. These are usually logical errors, such as mathematical mistakes.

Testing for runtime errors requires that the program be executed with sample data or sample input. The sample data should be such that the correct output can be predicted. If the program does not produce the correct output, a logical error is present in the program.

8. Correct any runtime errors found while running the program. Repeat Steps 5 through 8 as many times as necessary

When runtime errors are found in a program, they must be corrected. You must identify the step where the error occurred and determine the cause. If an error is a result of incorrect logic (such as an improperly stated math formula), you must correct the statement or statements involved in the logic. If an error is due to an incomplete understanding of the program requirements, then you must restate the program purpose and modify the program model and source code. The program must then be saved, recompiled, and retested. This means Steps 5 through 8 must be repeated until the program reliably produces satisfactory results.

9. Validate the results of the program

When you believe you have corrected all the runtime errors, enter test data and determine whether the program solves the original problem.

Software Engineering

The field of software engineering encompasses the whole process of crafting computer software. It includes designing, writing, testing, debugging, documenting, modifying, and maintaining complex software development projects. Like traditional engineers, software engineers use a number of tools in their craft. Here are a few examples:

- Program specifications
- Diagrams of screen output
- Diagrams representing the program components and the flow of data
- Pseudocode
- Examples of expected input and desired output
- Special software designed for testing programs

Most commercial software applications are large and complex. Usually a team of programmers, not a single individual, develops them. It is important that the program requirements be thoroughly analyzed and divided into subtasks that are handled by individual teams, or individuals within a team.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 1.17 What four items should you identify when defining what a program is to do?
- 1.18 What does it mean to “visualize a program running”? What is the value of such an activity?
- 1.19 What is pseudocode?
- 1.20 Describe what a compiler does with a program’s source code.
- 1.21 What is a runtime error?
- 1.22 Is a syntax error (such as misspelling a key word) found by the compiler or when the program is running?
- 1.23 What is the purpose of testing a program with sample data or input?

1.7

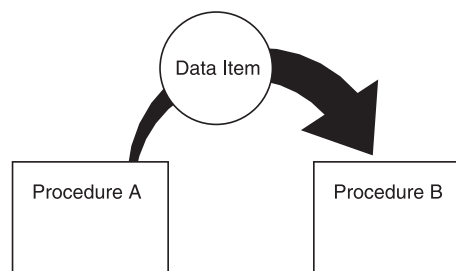
Object-Oriented Programming

CONCEPT: Java is an object-oriented programming (OOP) language. OOP is a method of software development that has its own practices, concepts, and vocabulary.

There are primarily two methods of programming in use today: procedural and object-oriented. The earliest programming languages were procedural, meaning a program was made of one or more procedures. A *procedure* is a set of programming statements that, together, perform a specific task. The statements might gather input from the user, manipulate data stored in the computer’s memory, and perform calculations or any other operation necessary to complete the procedure’s task.

Procedures typically operate on data items that are separate from the procedures. In a procedural program, the data items are commonly passed from one procedure to another, as shown in Figure 1-9.

Figure 1-9 Data is passed among procedures



As you might imagine, the focus of procedural programming is on the creation of procedures that operate on the program's data. The separation of data and the code that operates on the data often leads to problems, however. For example, the data is stored in a particular format, which consists of variables and more complex structures that are created from variables. The procedures that operate on the data must be designed with that format in mind. But, what happens if the format of the data is altered? Quite often, a program's specifications change, resulting in a redesigned data format. When the structure of the data changes, the code that operates on the data must also be changed to accept the new format. This results in added work for programmers and a greater opportunity for bugs to appear in the code.

This has helped influence the shift from procedural programming to object-oriented programming (OOP). Whereas procedural programming is centered on creating procedures, object-oriented programming is centered on creating objects. An object is a software entity that contains data and procedures. The data contained in an object is known as the object's *attributes*. The procedures, or behaviors, that an object performs are known as the object's *methods*. The object is, conceptually, a self-contained unit consisting of data (attributes) and procedures (methods). This is illustrated in Figure 1-10.

OOP addresses the problem of code/data separation through encapsulation and data hiding. *Encapsulation* refers to the combining of data and code into a single object. *Data hiding* refers to an object's ability to hide its data from code that is outside the object. Only the object's methods may then directly access and make changes to the object's data. An object typically hides its data, but allows outside code to access the methods that operate on the data. As shown in Figure 1-11, the object's methods provide programming statements outside the object with indirect access to the object's data.

When an object's internal data is hidden from outside code and access to that data is restricted to the object's methods, the data is protected from accidental corruption. In addition, the programming code outside the object does not need to know about the format or

Figure 1-10 An object contains data and procedures

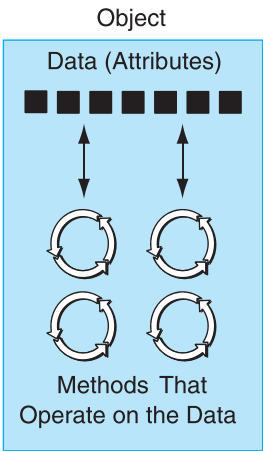
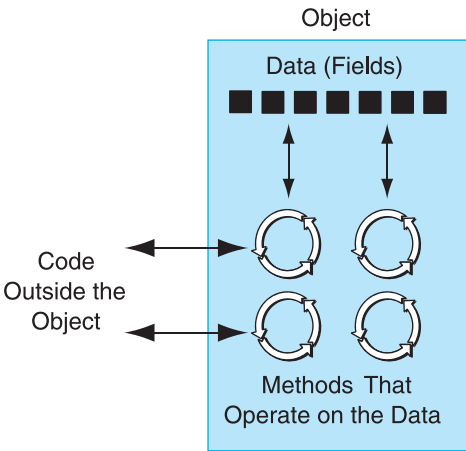


Figure 1-11 Code outside the object interacts with the object's methods



internal structure of the object's data. The code only needs to interact with the object's methods. When a programmer changes the structure of an object's internal data, he or she also modifies the object's methods so they may properly operate on the data. The way in which outside code interacts with the methods, however, does not change.

These are just a few of the benefits of object-oriented programming. Because Java is fully object-oriented, you will learn much more about OOP practices, concepts, and terms as you progress through this book.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 1.24 In procedural programming, what two parts of a program are typically separated?
- 1.25 What are an object's attributes?
- 1.26 What are an object's methods?
- 1.27 What is encapsulation?
- 1.28 What is data hiding?

Review Questions and Exercises

Multiple Choice

1. This part of the computer fetches instructions, carries out the operations commanded by the instructions, and produces some outcome or resultant information.
 - a. memory
 - b. CPU
 - c. secondary storage
 - d. input device
2. A byte is made up of eight
 - a. CPUs
 - b. addresses
 - c. variables
 - d. bits
3. The coordination of all the operations of a computer is controlled by _____.
 - a. the ALU
 - b. the control unit
 - c. memory
 - d. a storage device
4. This type of memory can hold data for long periods of time—even when there is no power to the computer.
 - a. RAM
 - b. primary storage
 - c. secondary storage
 - d. CPU storage

5. A runtime error is usually the result of
 - a. a syntax error
 - b. a logical error
 - c. a compilation error
 - d. bad data
6. This type of program is designed to be transmitted over the Internet and run in a Web browser.
 - a. application
 - b. applet
 - c. machine language
 - d. source code
7. These are words that have a special meaning in the programming language.
 - a. punctuation
 - b. programmer-defined names
 - c. key words
 - d. operators
8. These are symbols or words that perform operations on one or more operands.
 - a. punctuation
 - b. programmer-defined names
 - c. key words
 - d. operators
9. These are collections of well-defined instructions to perform a task or to solve a problem.
 - a. program
 - b. statement
 - c. operation
 - d. circuits
10. These are words or names that are used to identify storage locations in memory and parts of the program that are created by the programmer.
 - a. punctuation
 - b. programmer-defined names
 - c. key words
 - d. operators
11. These are the rules that must be followed when writing a program.
 - a. syntax
 - b. punctuation
 - c. key words
 - d. operators
12. This is a named storage location in the computer's memory.
 - a. class
 - b. key word
 - c. variable
 - d. operator

13. The Java compiler generates _____.
 - a. machine code
 - b. byte code
 - c. source code
 - d. HTML
14. IDE stands for _____.
 - a. Integrated Design Enhancement
 - b. Integrated Design Environments
 - c. Integrated Data Environments
 - d. Integrated Development Environments

Find the Error

1. The following pseudocode algorithm has an error. The program is supposed to ask the user for the length and width of a rectangular room, and then display the room's area. The program must multiply the width by the length to determine the area. Find the error.

area = width × length

Display "What is the room's width?"

Input width.

Display "What is the room's length?"

Input length.

Display area.

Algorithm Workbench

Write pseudocode algorithms for the programs described as follows:

1. Available Credit

A program that calculates a customer's available credit should ask the user for the following:

- The customer's maximum amount of credit
- The amount of credit used by the customer

Once these items have been entered, the program should calculate and display the customer's available credit. You can calculate available credit by subtracting the amount of credit used from the maximum amount of credit.

2. Sales Tax

A program that calculates the total of a retail sale should ask the user for the following:

- The retail price of the item being purchased
- The sales tax rate

Once these items have been entered, the program should calculate and display the following:

- The sales tax for the purchase
- The total of the sale

3. Discount Calculation

A program that calculates the total discount obtained on the online purchase of two products must ask the user for the following:

- The price of each product before discount
- The percentage of discount on each product

Once the program calculates the total discount, it should be displayed on the screen.

Predict the Result

The following are programs expressed as English statements. What would each display on the screen if they were actual programs?

1. The variable *x* starts with the value 4.
The variable *y* starts with the value 3.
Multiply *x* by 2.
Multiply *y* by 3.
Subtract *x* from *y* and store the result in *y*.
Display the value in *y* on the screen.
2. The variable *a* starts with the value 10.
The variable *b* starts with the value 2.
The variable *c* starts with the value 4.
Store the value of *a* times *b* in *a*.
Store the value of *b* times *c* in *c*.
Add *a* and *c*, and store the result in *b*.
Display the value in *b* on the screen.

Short Answer

1. Both main memory and secondary storage are types of memory. Describe the difference between the two.
2. What type of memory is usually volatile?
3. What is the difference between operating system software and application software?
4. Why must programs written in a high-level language be translated into machine language before they can be run?
5. Why is it easier to write a program in a high-level language than in machine language?
6. What is a source file?
7. What is the difference between a syntax error and a logical error?
8. What is an algorithm?
9. What is a compiler?
10. What is the difference between an application and an applet?
11. Why are Java applets safe to download and execute?
12. What must a computer have in order for it to execute Java programs?
13. What is the difference between machine language code and byte code?

14. Why does byte code make Java a portable language?
15. Is encapsulation a characteristic of procedural or object-oriented programming?
16. Why should an object hide its data?
17. What part of an object forms an interface through which outside code may access the object's data?
18. What are the steps involved in converting Java source code into machine code?
19. Will the Java compiler translate a source file that contains syntax errors?
20. What does the Java compiler translate Java source code to?
21. Assuming you are using the JDK, what command would you type at the operating system command prompt to compile the program `LabAssignment.java`?
22. Assuming there are no syntax errors in the *LabAssignment.java* program when it is compiled, answer the following questions.
 - a. What file will be produced?
 - b. What will the file contain?
 - c. What command would you type at the operating system command prompt to run the program?

Programming Challenge

MyProgrammingLab™ Visit www.myprogramminglab.com to complete many of these Programming Challenges online and get instant feedback.

1. Your First Java Program

This assignment will help you get acquainted with your Java development software. Here is the Java program you will enter:



VideoNote

Your First Java
Program

```
// This is my first Java program.
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

If You Are Using the JDK at the Command Prompt:

1. Use a text editor to type the source code exactly as it is shown. Be sure to place all the punctuation characters and be careful to match the case of the letters as they are shown. Save it to a file named *MyFirstProgram.java*.
2. After saving the program, go to your operating system's command prompt and change your current directory or folder to the one that contains the Java program you just created. Then use the following command to compile the program:

```
javac MyFirstProgram.java
```

If you typed the contents of the file exactly as shown, you shouldn't have any syntax errors. If you see error messages, open the file in the editor and compare your code to that shown. Correct any mistakes you have made, save the file, and run the compiler again. If you see no error messages, the file was successfully compiled.

3. Next, enter the following command to run the program:

```
java MyFirstProgram
```

Be sure to use the capitalization of `MyFirstProgram` exactly as it is shown here. You should see the message "Hello World!" displayed on the screen.

If You Are Using an IDE:

Because there are many Java IDEs, we cannot include specific instructions for all of these. The following are general steps that should apply to most of them. You will need to consult your IDE's documentation for specific instructions.

1. Start your Java IDE and perform any necessary setup operations, such as starting a new project and creating a new Java source file.
2. Use the IDE's text editor to type the source code exactly as it is shown. Be sure to place all the punctuation characters and be careful to match the case of the letters as they are shown. Save it to a file named *MyFirstProgram.java*.
3. After saving the program, use your IDE's command to compile the program. If you typed the contents of the file exactly as shown, you shouldn't have any syntax errors. If you see error messages, compare your code to that shown. Correct any mistakes you have made, save the file, and run the compiler again. If you see no error messages, the file was successfully compiled.

Use your IDE's command to run the program. You should see the message "Hello World!" displayed.

TOPICS

- | | |
|---|--|
| 2.1 The Parts of a Java Program | 2.8 Creating Named Constants with <code>final</code> |
| 2.2 The <code>print</code> and <code>println</code> Methods, and the Java API | 2.9 The <code>String</code> Class |
| 2.3 Variables and Literals | 2.10 Scope |
| 2.4 Primitive Data Types | 2.11 Comments |
| 2.5 Arithmetic Operators | 2.12 Programming Style |
| 2.6 Combined Assignment Operators | 2.13 Reading Keyboard Input |
| 2.7 Conversion between Primitive Data Types | 2.14 Dialog Boxes |
| | 2.15 Common Errors to Avoid |

2.1 The Parts of a Java Program

CONCEPT: A Java program has parts that serve specific purposes.

Java programs are made up of different parts. Your first step in learning Java is to learn what the parts are. We will begin by looking at a simple example, shown in Code Listing 2-1.

Code Listing 2-1 (Simple.java)

```
1 // This is a simple Java program.
2
3 public class Simple
4 {
5     public static void main(String[] args)
6     {
7         System.out.println("Programming is great fun!");
8     }
9 }
```




TIP: Remember, the line numbers shown in the program listings are not part of the program. The numbers are shown so we can refer to specific lines in the programs.

As mentioned in Chapter 1, the names of Java source code files end with *.java*. The program shown in Code Listing 2-1 is named *Simple.java*. Using the Java compiler, this program may be compiled with the following command:

```
javac Simple.java
```

The compiler will create another file named *Simple.class*, which contains the translated Java byte code. This file can be executed with the following command:

```
java Simple
```



TIP: Remember, you do not type the *.class* extension when using the `java` command.

The output of the program is as follows. This is what appears on the screen when the program runs.

Program Output

```
Programming is great fun!
```

Let's examine the program line by line. Here's the statement in line 1:

```
// This is a simple Java program.
```

Other than the two slash marks that begin this line, it looks pretty much like an ordinary sentence. The `//` marks the beginning of a comment. The compiler ignores everything from the double-slash to the end of the line. That means you can type anything you want on that line and the compiler never complains. Although comments are not required, they are very important to programmers. Most programs are much more complicated than this example, and comments help explain what's going on.

Line 2 is blank. Programmers often insert blank lines in programs to make them easier to read. Line 3 reads:

```
public class Simple
```

This line is known as a *class header*, and it marks the beginning of a *class definition*. One of the uses of a class is to serve as a container for an application. As you progress through this book, you will learn more and more about classes. For now, just remember that a Java program must have at least one class definition. This line of code consists of three words: `public`, `class`, and `Simple`. Let's take a closer look at each word.

- `public` is a Java key word, and it must be written in all lowercase letters. It is known as an *access specifier*, and it controls where the class may be accessed from. The `public` specifier means access to the class is unrestricted. (In other words, the class is “open to the public.”)
- `class`, which must also be written in lowercase letters, is a Java key word that indicates the beginning of a class definition.

- `Simple` is the class name. This name was made up by the programmer. The class could have been called `Pizza`, or `Dog`, or anything else the programmer wanted. Programmer-defined names may be written in lowercase letters, uppercase letters, or a mixture of both.

In a nutshell, this line of code tells the compiler that a publicly accessible class named `Simple` is being defined. Here are two more points to know about classes:

- You may create more than one class in a file, but you may have only one `public class` per Java file.
- When a Java file has a `public class`, the name of the public class must be the same as the name of the file (without the `.java` extension). For instance, the program in Code Listing 2-1 has a `public class` named `Simple`, so it is stored in a file named *Simple.java*.



NOTE: Java is a case-sensitive language. That means it regards uppercase letters as being entirely different characters than their lowercase counterparts. The word `Public` is not the same as `public`, and `Class` is not the same as `class`. Some words in a Java program must be entirely in lowercase, while other words may use a combination of lower and uppercase characters. Later in this chapter, you will see a list of all the Java key words, which must appear in lowercase.

Line 4 contains only a single character:

```
{
```

This is called a left brace, or an opening brace, and is associated with the beginning of the class definition. All of the programming statements that are part of the class are enclosed in a set of braces. If you glance at the last line in the program, line 9, you'll see the closing brace. Everything between the two braces is the *body* of the class named `Simple`. Here is the program code again, this time the body of the class definition is shaded.

```
// This is a simple Java program.  
public class Simple  
{  
    public static void main(String[] args)  
    {  
        System.out.println("Programming is great fun!");  
    }  
}
```



WARNING! Make sure you have a closing brace for every opening brace in your program!

Line 5 reads:

```
    public static void main(String[] args)
```

This line is known as a *method header*. It marks the beginning of a *method*. A method can be thought of as a group of one or more programming statements that collectively has a name. When creating a method, you must tell the compiler several things about it. That is

why this line contains so many words. At this point, the only thing you should be concerned about is that the name of the method is `main`, and the rest of the words are required for the method to be properly defined. This is shown in Figure 2-1.

Recall from Chapter 1 that a stand-alone Java program that runs on your computer is known as an application. Every Java application must have a method named `main`. The `main` method is the starting point of an application.

Figure 2-1 The `main` method header

Name of the Method
↓
`public static void main(String[] args)`
The other parts of this line are necessary
for the method to be properly defined.



NOTE: For the time being, all the programs you will write will consist of a class with a `main` method whose header looks exactly like the one shown in Code Listing 2-1. As you progress through this book you will learn what `public static void` and `(String[] args)` mean. For now, just assume that you are learning a “recipe” for assembling a Java program.

Line 6 has another opening brace:

```
{
```

This opening brace belongs to the `main` method. Remember that braces enclose statements, and every opening brace must have an accompanying closing brace. If you look at line 8 you will see the closing brace that corresponds with this opening brace. Everything between these braces is the *body* of the `main` method.

Line 7 appears as follows:

```
System.out.println("Programming is great fun!");
```

To put it simply, this line displays a message on the screen. The message, “Programming is great fun!” is printed without the quotation marks. In programming terms, the group of characters inside the quotation marks is called a *string literal*.



NOTE: This is the only line in the program that causes anything to be printed on the screen. The other lines, like `public class Simple` and `public static void main(String[] args)`, are necessary for the framework of your program, but they do not cause any screen output. Remember, a program is a set of instructions for the computer. If something is to be displayed on the screen, you must use a programming statement for that purpose.

At the end of the line is a *semicolon*. Just as a period marks the end of a sentence, a semicolon marks the end of a statement in Java. Not every line of code ends with a semicolon, however. Here is a summary of where you do not place a semicolon:

- Comments do not have to end with a semicolon because they are ignored by the compiler.
- Class headers and method headers do not end with a semicolon because they are terminated with a body of code inside braces.
- The brace characters, { and }, are not statements, so you do not place a semicolon after them.

It might seem that the rules for where to put a semicolon are not clear at all. For now, just concentrate on learning the parts of a program. You'll soon get a feel for where you should and should not use semicolons.

As has already been pointed out, lines 8 and 9 contain the closing braces for the `main` method and the class definition:

```
    }
}
```

Before continuing, let's review the points we just covered, including some of the more elusive rules.

- Java is a case-sensitive language. It does not regard uppercase letters as being the same character as their lowercase equivalents.
- All Java programs must be stored in a file with a name that ends with `.java`.
- Comments are ignored by the compiler.
- A `.java` file may contain many classes, but may have only one `public` class. If a `.java` file has a `public` class, the class must have the same name as the file. For instance, if the file `Pizza.java` contains a `public class`, the class's name would be `Pizza`.
- Every Java application program must have a method named `main`.
- For every left brace, or opening brace, there must be a corresponding right brace, or closing brace.
- Statements are terminated with semicolons. This does not include comments, class headers, method headers, or braces.

In the sample program, you encountered several special characters. Table 2-1 summarizes how they were used.

Table 2-1 Special characters

Characters	Name	Meaning
//	Double slash	Marks the beginning of a comment
()	Opening and closing parentheses	Used in a method header
{ }	Opening and closing braces	Encloses a group of statements, such as the contents of a class or a method
" "	Quotation marks	Encloses a string of characters, such as a message that is to be printed on the screen
;	Semicolon	Marks the end of a complete programming statement



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 2.1 The following program will not compile because the lines have been mixed up.

```
public static void main(String[] args)
}
// A crazy mixed up program
public class Columbus
{
System.out.println("In 1492 Columbus sailed the ocean blue.");
{
}
```

When the lines are properly arranged, the program should display the following on the screen:

```
In 1492 Columbus sailed the ocean blue.
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

- 2.2 When the program in Question 2.1 is saved to a file, what should the file be named?
- 2.3 Complete the following program skeleton so it displays the message “Hello World” on the screen.

```
public class Hello
{
    public static void main(String[] args)
    {
        // Insert code here to complete the program
    }
}
```

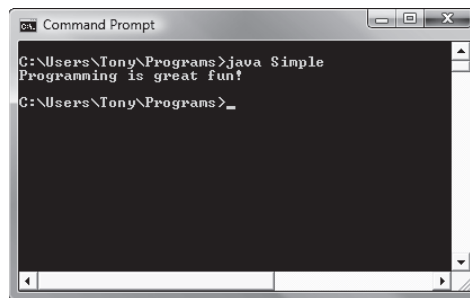
- 2.4 On paper, write a program that will display your name on the screen. Place a comment with today’s date at the top of the program. Test your program by entering, compiling, and running it.
- 2.5 All Java source code filenames must end with _____.
 a) a semicolon
 b) *.class*
 c) *.java*
 d) none of the above
- 2.6 Every Java application program must have _____.
 a) a method named *main*
 b) more than one class definition
 c) one or more comments

2.2 The print and println Methods, and the Java API

CONCEPT: The `print` and `println` methods are used to display text output. They are part of the Java API, which is a collection of prewritten classes and methods for performing specific operations.

In this section, you will learn how to write programs that produce output on the screen. The simplest type of output that a program can display on the screen is console output. *Console output* is merely plain text. When you display console output in a system that uses a graphical user interface, such as Windows or Mac OS, the output usually appears in a window similar to the one shown in Figure 2-2.

Figure 2-2 A console window (Microsoft Corporation)



VideoNote

Displaying
Console Output

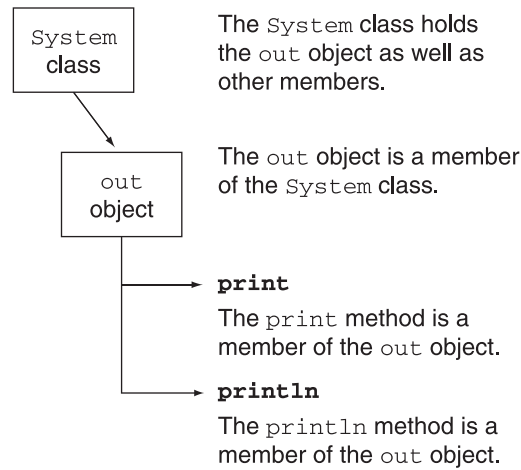
The word *console* is an old computer term. It comes from the days when the operator of a large computer system interacted with the system by typing on a terminal that consisted of a simple screen and keyboard. This terminal was known as the *console*. The console screen, which displayed only text, was known as the standard output device. Today, the term *standard output device* typically refers to the device that displays console output.

Performing output in Java, as well as many other tasks, is accomplished by using the Java API. The term *API* stands for *Application Programmer Interface*. The API is a standard library of prewritten classes for performing specific operations. These classes and their methods are available to all Java programs. The `print` and `println` methods are part of the API and provide ways for output to be displayed on the standard output device.

The program in Code Listing 2-1 (`Simple.java`) uses the following statement to print a message on the screen:

```
System.out.println("Programming is great fun!");
```

`System` is a class that is part of the Java API. The `System` class contains objects and methods that perform system-level operations. One of the objects contained in the `System` class is named `out`. The `out` object has methods, such as `print` and `println`, for performing output on the system console, or standard output device. The hierarchical relationship among `System`, `out`, `print`, and `println` is shown in Figure 2-3.

Figure 2-3 Relationship among the `System` class, the `out` object, and the `print` and `println` methods

Here is a brief summary of how it all works together:

- The `System` class is part of the Java API. It has member objects and methods for performing system-level operations, such as sending output to the console.
- The `out` object is a member of the `System` class. It provides methods for sending output to the screen.
- The `print` and `println` methods are members of the `out` object. They actually perform the work of writing characters on the screen.

This hierarchy explains why the statement that executes `println` is so long. The sequence `System.out.println` specifies that `println` is a member of `out`, which is a member of `System`.



NOTE: The period that separates the names of the objects is pronounced “dot.” `System.out.println` is pronounced “system dot out dot print line.”

The value that is to be displayed on the screen is placed inside the parentheses. This value is known as an *argument*. For example, the following statement executes the `println` method using the string “King Arthur” as its argument. This will print “King Arthur” on the screen. (The quotation marks are not displayed.)

```
System.out.println("King Arthur");
```

An important thing to know about the `println` method is that after it displays its message, it advances the cursor to the beginning of the next line. The next item printed on the screen will begin in this position. For example, look at the program in Code Listing 2-2.

Because each string is printed with separate `println` statements in Code Listing 2-2, they appear on separate lines in the Program Output.

Code Listing 2-2 (TwoLines.java)

```
1 // This is another simple Java program.
2
3 public class TwoLines
4 {
5     public static void main(String[] args)
6     {
7         System.out.println("Programming is great fun!");
8         System.out.println("I can't get enough of it!");
9     }
10 }
```

Program Output

```
Programming is great fun!
I can't get enough of it!
```

The print Method

The print method, which is also part of the `System.out` object, serves a purpose similar to that of `println`—to display output on the screen. The print method, however, does not advance the cursor to the next line after its message is displayed. Look at Code Listing 2-3.

Code Listing 2-3 (GreatFun.java)

```
1 // This is another simple Java program.
2
3 public class GreatFun
4 {
5     public static void main(String[] args)
6     {
7         System.out.print("Programming is ");
8         System.out.println("great fun!");
9     }
10 }
```

Program Output

```
Programming is great fun!
```

An important concept to understand about Code Listing 2-3 is that, although the output is broken up into two programming statements, this program will still display the message on one line. The data that you send to the print method is displayed in a continuous stream. Sometimes this can produce less-than-desirable results. The program in Code Listing 2-4 is an example.

Code Listing 2-4 (Unruly.java)

```
1 // An unruly printing program
2
3 public class Unruly
4 {
5     public static void main(String[] args)
6     {
7         System.out.print("These are our top sellers:");
8         System.out.print("Computer games");
9         System.out.print("Coffee");
10        System.out.println("Aspirin");
11    }
12 }
```

Program Output

```
These are our top sellers:Computer gamesCoffeeAspirin
```

The layout of the actual output looks nothing like the arrangement of the strings in the source code. First, even though the output is broken up into four lines in the source code (lines 7 through 10), it comes out on the screen as one line. Second, notice that some of the words that are displayed are not separated by spaces. The strings are displayed exactly as they are sent to the print method. If spaces are to be displayed, they must appear in the strings.

There are two ways to fix this program. The most obvious way is to use `println` methods instead of `print` methods. Another way is to use escape sequences to separate the output into different lines. An *escape sequence* starts with the backslash character (`\`), and is followed by one or more *control characters*. It allows you to control the way output is displayed by embedding commands within the string itself. The escape sequence that causes the output cursor to go to the next line is `\n`. Code Listing 2-5 illustrates its use.

Code Listing 2-5 (Adjusted.java)

```
1 // A well adjusted printing program
2
3 public class Adjusted
4 {
5     public static void main(String[] args)
6     {
7         System.out.print("These are our top sellers:\n");
8         System.out.print("Computer games\nCoffee\n");
9         System.out.println("Aspirin");
10    }
11 }
```

Program Output

```
These are our top sellers:
Computer games
Coffee
Aspirin
```

The `\n` characters are called the newline escape sequence. When the `print` or `println` method encounters `\n` in a string, it does not print the `\n` characters on the screen, but interprets them as a special command to advance the output cursor to the next line. There are several other escape sequences as well. For instance, `\t` is the tab escape sequence. When `print` or `println` encounters it in a string, it causes the output cursor to advance to the next tab position. Code Listing 2-6 shows it in use.

Code Listing 2-6 (Tabs.java)

```

1  // Another well-adjusted printing program
2
3  public class Tabs
4  {
5      public static void main(String[] args)
6      {
7          System.out.print("These are our top sellers:\n");
8          System.out.print("\tComputer games\n\tCoffee\n");
9          System.out.println("\tAspirin");
10     }
11 }
```

Program Output

```

These are our top sellers:
    Computer games
    Coffee
    Aspirin
```



NOTE: Although you have to type two characters to write an escape sequence, they are stored in memory as a single character.

Table 2-2 lists the common escape sequences and describes them.

Table 2-2 Common escape sequences

Escape Sequence	Name	Description
<code>\n</code>	Newline	Advances the cursor to the next line for subsequent printing
<code>\t</code>	Horizontal tab	Causes the cursor to skip over to the next tab stop
<code>\b</code>	Backspace	Causes the cursor to back up, or move left, one position
<code>\r</code>	Return	Causes the cursor to go to the beginning of the current line, not the next line
<code>\\</code>	Backslash	Causes a backslash to be printed
<code>\'</code>	Single quote	Causes a single quotation mark to be printed
<code>\"</code>	Double quote	Causes a double quotation mark to be printed



WARNING! Do not confuse the backslash (\) with the forward slash (/). An escape sequence will not work if you accidentally start it with a forward slash. Also, do not put a space between the backslash and the control character.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

2.7 The following program will not compile because the lines have been mixed up.

```
System.out.print("Success\n");
}
public class Success
{
System.out.print("Success\n");
public static void main(String[] args)
System.out.print("Success ");
}
// It's a mad, mad program.
System.out.println("\nSuccess");
{
```

When the lines are arranged properly, the program should display the following output on the screen:

Program Output

```
Success
Success Success

Success
```

Rearrange the lines in the correct order. Test the program by entering it on the computer, compiling it, and running it.

2.8 Study the following program and show what it will print on the screen.

```
// The Works of Wolfgang
public class Wolfgang
{
    public static void main(String[] args)
    {
        System.out.print("The works of Wolfgang\ninclude ");
        System.out.print("the following");
        System.out.print("\nThe Turkish March ");
        System.out.print("and Symphony No. 40 ");
        System.out.println("in G minor.");
    }
}
```

2.9 On paper, write a program that will display your name on the first line; your street address on the second line; your city, state, and ZIP code on the third line; and your telephone number on the fourth line. Place a comment with today's date at the top of the program. Test your program by entering, compiling, and running it.

2.3 Variables and Literals

CONCEPT: A variable is a named storage location in the computer's memory. A literal is a value that is written into the code of a program.

As you discovered in Chapter 1, variables allow you to store and work with data in the computer's memory. Part of the job of programming is to determine how many variables a program will need and what types of data they will hold. The program in Code Listing 2-7 is an example of a Java program with a variable.

Code Listing 2-7 (Variable.java)

```
1 // This program has a variable.
2
3 public class Variable
4 {
5     public static void main(String[] args)
6     {
7         int value;
8
9         value = 5;
10        System.out.print("The value is ");
11        System.out.println(value);
12    }
13 }
```

Program Output

The value is 5

Let's look more closely at this program. Here is line 7:

```
int value;
```



This is called a *variable declaration*. Variables must be declared before they can be used. A variable declaration tells the compiler the variable's name and the type of data it will hold. This line indicates the variable's name is value. The word `int` stands for integer, so value will only be used to hold integer numbers. Notice that variable declarations end with a semicolon. The next statement in this program appears in line 9:

```
value = 5;
```

This is called an *assignment statement*. The equal sign is an operator that stores the value on its right (in this case 5) into the variable named on its left. After this line executes, the value variable will contain the value 5.



NOTE: This line does not print anything on the computer screen. It runs silently behind the scenes.

Now look at lines 10 and 11:

```
System.out.print("The value is ");  
System.out.println(value);
```

The statement in line 10 sends the string literal “The value is ” to the `print` method. The statement in line 11 sends the name of the `value` variable to the `println` method. When you send a variable name to `print` or `println`, the variable’s contents are displayed. Notice there are no quotation marks around `value`. Look at what happens in Code Listing 2-8.

Code Listing 2-8 (Variable2.java)

```
1 // This program has a variable.  
2  
3 public class Variable2  
4 {  
5     public static void main(String[] args)  
6     {  
7         int value;  
8  
9         value = 5;  
10        System.out.print("The value is ");  
11        System.out.println("value");  
12    }  
13 }
```

Program Output

The value is value

When double quotation marks are placed around the word `value` it becomes a string literal, not a variable name. When string literals are sent to `print` or `println`, they are displayed exactly as they appear inside the quotation marks.

Displaying Multiple Items with the + Operator

When the `+` operator is used with strings, it is known as the *string concatenation operator*. To concatenate means to append, so the string concatenation operator appends one string to another. For example, look at the following statement:

```
System.out.println("This is " + "one string.");
```

This statement will print:

```
This is one string.
```

The `+` operator produces a string that is the combination of the two strings used as its operands. You can also use the `+` operator to concatenate the contents of a variable to a string. The following code shows an example:

```
number = 5;  
System.out.println("The value is " + number);
```

The second line uses the + operator to concatenate the contents of the number variable with the string “The value is ”. Although number is not a string, the + operator converts its value to a string and then concatenates that value with the first string. The output that will be displayed is:

```
The value is 5
```

Sometimes the argument you use with `print` or `println` is too long to fit on one line in your program code. However, a string literal cannot begin on one line and end on another. For example, the following will cause an error:

```
// This is an error!  
System.out.println("Enter a value that is greater than zero  
                    and less than 10." );
```

You can remedy this problem by breaking the argument up into smaller string literals, and then using the string concatenation operator to spread them out over more than one line. Here is an example:

```
System.out.println("Enter a value that is " +  
                  "greater than zero and less " +  
                  "than 10." );
```

In this statement, the argument is broken up into three strings and joined using the + operator. The following example shows the same technique used when the contents of a variable are part of the concatenation:

```
sum = 249;  
System.out.println("The sum of the three " +  
                  "numbers is " + sum);
```

Be Careful with Quotation Marks

As shown in Code Listing 2-8, placing quotation marks around a variable name changes the program’s results. In fact, placing double quotation marks around anything that is not intended to be a string literal will create an error of some type. For example, in Code Listings 2-7 and 2-8, the number 5 was assigned to the variable `value`. It would have been an error to perform the assignment this way:

```
value = "5";    // Error!
```

In this statement, 5 is no longer an integer, but a string literal. Because `value` was declared as an integer variable, you can only store integers in it. In other words, 5 and “5” are not the same thing.

The fact that numbers can be represented as strings frequently confuses students who are new to programming. Just remember that strings are intended for humans to read. They are to be printed on computer screens or paper. Numbers, however, are intended primarily for mathematical operations. You cannot perform math on strings, and before numbers can be displayed on the screen, first they must be converted to strings. (Fortunately, `print` and `println` handle the conversion automatically when you send numbers to them.) Don’t fret if this still bothers you. Later in this chapter, we will shed more light on the differences among numbers, characters, and strings by discussing their internal storage.

More about Literals

A literal is a value that is written in the code of a program. Literals are commonly assigned to variables or displayed. Code Listing 2-9 contains both literals and a variable.

Code Listing 2-9 (Literals.java)

```
1 // This program has literals and a variable.
2
3 public class Literals
4 {
5     public static void main(String[] args)
6     {
7         int apples;
8
9         apples = 20;
10        System.out.println("Today we sold " + apples +
11                           " bushels of apples.");
12    }
13 }
```

Program Output

Today we sold 20 bushels of apples.

Of course, the variable in this program is `apples`. It is declared as an integer. Table 2-3 shows a list of the literals found in the program.

Table 2-3 Literals

Literal	Type of Literal
20	Integer literal
“Today we sold ”	String literal
“ bushels of apples.”	String literal

Identifiers

An *identifier* is a programmer-defined name that represents some element of a program. Variable names and class names are examples of identifiers. You may choose your own variable names and class names in Java, as long as you do not use any of the Java key words. The *key words* make up the core of the language and each has a specific purpose. Table 1-3 in Chapter 1 and Appendix D (available on the book’s companion Web site) show a complete list of Java key words.

You should always choose names for your variables that give an indication of what they are used for. You may be tempted to declare variables with names like this:

```
int x;
```

The rather nondescript name, `x`, gives no clue as to what the variable's purpose is. Here is a better example.

```
int itemsOrdered;
```

The name `itemsOrdered` gives anyone reading the program an idea of what the variable is used for. This method of coding helps produce *self-documenting programs*, which means you get an understanding of what the program is doing just by reading its code. Because real-world programs usually have thousands of lines of code, it is important that they be as self-documenting as possible.

You have probably noticed the mixture of uppercase and lowercase letters in the name `itemsOrdered`. Although all of Java's key words must be written in lowercase, you may use uppercase letters in variable names. The reason the `O` in `itemsOrdered` is capitalized is to improve readability. Normally "items ordered" is used as two words. Variable names cannot contain spaces, however, so the two words must be combined. When "items" and "ordered" are stuck together, you get a variable declaration like this:

```
int itemsordered;
```

Capitalization of the letter `O` makes `itemsOrdered` easier to read. Typically, variable names begin with a lowercase letter, and after that, the first letter of each individual word that makes up the variable name is capitalized.

The following are some specific rules that must be followed with all identifiers:

- The first character must be one of the letters `a–z` or `A–Z`, an underscore (`_`), or a dollar sign (`$`).
- After the first character, you may use the letters `a–z` or `A–Z`, the digits `0–9`, underscores (`_`), or dollar signs (`$`).
- Uppercase and lowercase characters are distinct. This means `itemsOrdered` is not the same as `itemsordered`.
- Identifiers cannot include spaces.



NOTE: Although the `$` is a legal identifier character, it is normally used for special purposes. So, don't use it in your variable names.

Table 2-4 shows a list of variable names and tells whether each is legal or illegal in Java.

Table 2-4 Some variable names

Variable Name	Legal or Illegal?
<code>dayOfWeek</code>	Legal
<code>3dGraph</code>	Illegal because identifiers cannot begin with a digit
<code>june1997</code>	Legal
<code>mixture#3</code>	Illegal because identifiers may use only alphabetic letters, digits, underscores, or dollar signs
<code>week day</code>	Illegal because identifiers cannot contain spaces

Class Names

As mentioned before, it is standard practice to begin variable names with a lowercase letter, and then capitalize the first letter of each subsequent word that makes up the name. It is also a standard practice to capitalize the first letter of a class name, as well as the first letter of each subsequent word it contains. This helps differentiate the names of variables from the names of classes. For example, `payRate` would be a variable name, and `Employee` would be a class name.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

2.10 Examine the following program.

```
// This program uses variables and literals.
public class BigLittle
{
    public static void main(String[] args)
    {
        int little;
        int big;
        little = 2;
        big = 2000;
        System.out.println("The little number is " + little);
        System.out.println("The big number is " + big);
    }
}
```

List the variables and literals found in the program.

2.11 What will the following program display on the screen?

```
public class CheckPoint
{
    public static void main(String[] args)
    {
        int number;
        number = 712;
        System.out.println("The value is " + "number");
    }
}
```

2.4

Primitive Data Types

CONCEPT: There are many different types of data. Variables are classified according to their data type, which determines the kind of data that may be stored in them.

Computer programs collect pieces of data from the real world and manipulate them in various ways. There are many different types of data. In the realm of numeric data, for example, there are whole and fractional numbers, negative and positive numbers, and numbers so large and others so small that they don't even have a name. Then there is textual information. Names and addresses, for instance, are stored as strings of characters. When you write a program you must determine what types of data it is likely to encounter.

Each variable has a *data type*, which is the type of data that the variable can hold. Selecting the proper data type is important because a variable's data type determines the amount of memory the variable uses, and the way the variable formats and stores data. It is important to select a data type that is appropriate for the type of data that your program will work with. If you are writing a program to calculate the number of miles to a distant star, you need variables that can hold very large numbers. If you are designing software to record microscopic dimensions, you need variables that store very small and precise numbers. If you are writing a program that must perform thousands of intensive calculations, you want variables that can be processed quickly. The data type of a variable determines all of these factors.

Table 2-5 shows all of the Java *primitive data types* for holding numeric data.

The words listed in the left column of Table 2-5 are the key words that you use in variable declarations. A variable declaration takes the following general format:

DataType VariableName;

Table 2-5 Primitive data types for numeric data

Data Type	Size	Range
byte	1 byte	Integers in the range of -128 to +127
short	2 bytes	Integers in the range of -32,768 to +32,767
int	4 bytes	Integers in the range of -2,147,483,648 to +2,147,483,647
long	8 bytes	Integers in the range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
float	4 bytes	Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of accuracy
double	8 bytes	Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy

DataType is the name of the data type and *VariableName* is the name of the variable. Here are some examples of variable declarations:

```
byte inches;
int speed;
short month;
float salesCommission;
double distance;
```

The size column in Table 2-5 shows the number of bytes that a variable of each of the data types uses. For example, an int variable uses 4 bytes, and a double variable uses 8 bytes.

The range column shows the ranges of numbers that may be stored in variables of each data type. For example, an `int` variable can hold numbers from `-2,147,483,648` up to `+2,147,483,647`. One of the appealing characteristics of the Java language is that the sizes and ranges of all the primitive data types are the same on all computers.



NOTE: These data types are called “primitive” because you cannot use them to create objects. Recall from Chapter 1’s discussion on object-oriented programming that an object has attributes and methods. With the primitive data types, you can only create variables, and a variable can only be used to hold a single value. Such variables do not have attributes or methods.

The Integer Data Types

The first four data types listed in Table 2-5, `byte`, `int`, `short`, and `long`, are all integer data types. An integer variable can hold whole numbers such as 7, 125, `-14`, and 6928. The program in Code Listing 2-10 shows several variables of different integer data types being used.

Code Listing 2-10 (IntegerVariables.java)

```

1  // This program has variables of several of the integer types.
2
3  public class IntegerVariables
4  {
5      public static void main(String[] args)
6      {
7          int checking; // Declare an int variable named checking.
8          byte miles; // Declare a byte variable named miles.
9          short minutes; // Declare a short variable named minutes.
10         long days; // Declare a long variable named days.
11
12         checking = -20;
13         miles = 105;
14         minutes = 120;
15         days = 189000;
16         System.out.println("We have made a journey of " + miles +
17                             " miles.");
18         System.out.println("It took us " + minutes + " minutes.");
19         System.out.println("Our account balance is $" + checking);
20         System.out.println("About " + days + " days ago Columbus " +
21                             "stood on this spot.");
22     }
23 }
```

Program Output

```

We have made a journey of 105 miles.
It took us 120 minutes.
Our account balance is $-20
About 189000 days ago Columbus stood on this spot.
```

In most programs you will need more than one variable of any given data type. If a program uses three integers, `length`, `width`, and `area`, they could be declared separately, as follows:

```
int length;  
int width;  
int area;
```

It is easier, however, to combine the three variable declarations:

```
int length, width, area;
```

You can declare several variables of the same type, simply by separating their names with commas.

Integer Literals

When you write an integer literal in your program code, Java assumes it to be of the `int` data type. For example, in Code Listing 2-10, the literals `-20`, `105`, `120`, and `189000` are all treated as `int` values. You can force an integer literal to be treated as a `long`, however, by suffixing it with the letter `L`. For example, the value `57L` would be treated as a `long`. Although you can use either an uppercase or a lowercase `L`, it is advisable to use the uppercase `L` because the lowercase `l` looks too much like the number `1`.



WARNING! You cannot embed commas in numeric literals. For example, the following statement will cause an error:

```
number = 1,257,649;           // ERROR!
```

This statement must be written as:

```
number = 1257649;           // Correct.
```

Floating-Point Data Types

Whole numbers are not adequate for many jobs. If you are writing a program that works with dollar amounts or precise measurements, you need a data type that allows fractional values. In programming terms, these are called *floating-point* numbers. Values such as `1.7` and `-45.316` are floating-point numbers.

In Java there are two data types that can represent floating-point numbers. They are `float` and `double`. The `float` data type is considered a single precision data type. It can store a floating-point number with 7 digits of accuracy. The `double` data type is considered a double precision data type. It can store a floating-point number with 15 digits of accuracy. The `double` data type uses twice as much memory as the `float` data type, however. A `float` variable occupies 4 bytes of memory, whereas a `double` variable uses 8 bytes.

Code Listing 2-11 shows a program that uses three `double` variables.

Code Listing 2-11 (Sale.java)

```
1 // This program demonstrates the double data type.  
2  
3 public class Sale  
4 {
```

```

5    public static void main(String[] args)
6    {
7        double price, tax, total;
8
9        price = 29.75;
10       tax = 1.76;
11       total = 31.51;
12       System.out.println("The price of the item " +
13                           "is " + price);
14       System.out.println("The tax is " + tax);
15       System.out.println("The total is " + total);
16   }
17 }

```

Program Output

```

The price of the item is 29.75
The tax is 1.76
The total is 31.51

```

Floating-Point Literals

When you write a floating-point literal in your program code, Java assumes it to be of the `double` data type. For example, in Code Listing 2-11, the literals 29.75, 1.76, and 31.51 are all treated as `double` values. Because of this, a problem can arise when assigning a floating-point literal to a `float` variable. Java is a *strongly typed language*, which means that it only allows you to store values of compatible data types in variables. A `double` value is not compatible with a `float` variable because a `double` can be much larger or much smaller than the allowable range for a `float`. As a result, code such as the following will cause an error:

```

float number;
number = 23.5;           // Error!

```

You can force a `double` literal to be treated as a `float`, however, by suffixing it with the letter F or f. The preceding code can be rewritten in the following manner to prevent an error:

```

float number;
number = 23.5F;          // This will work.

```



WARNING! If you are working with literals that represent dollar amounts, remember that you cannot embed currency symbols (such as \$) or commas in the literal. For example, the following statement will cause an error:

```
grossPay = $1,257.00;    // ERROR!
```

This statement must be written as:

```
grossPay = 1257.00;      // Correct.
```

Scientific and E Notation

Floating-point literals can be represented in scientific notation. Take the number 47,281.97. In scientific notation this number is 4.728197×10^4 . (10^4 is equal to 10,000, and $4.728197 \times 10,000$ is 47,281.97.)

Java uses E notation to represent values in scientific notation. In E notation, the number 4.728197×10^4 would be 4.728197E4. Table 2-6 shows other numbers represented in scientific and E notation.

Table 2-6 Floating-point representations

Decimal Notation	Scientific Notation	E Notation
247.91	2.4791×10^2	2.4791E2
0.00072	7.2×10^{-4}	7.2E-4
2,900,000	2.9×10^6	2.9E6



NOTE: The E can be uppercase or lowercase.

Code Listing 2-12 demonstrates the use of floating-point literals expressed in E notation.

Code Listing 2-12 (SunFacts.java)

```
1 // This program uses E notation.
2
3 public class SunFacts
4 {
5     public static void main(String[] args)
6     {
7         double distance, mass;
8
9         distance = 1.495979E11;
10        mass = 1.989E30;
11        System.out.println("The sun is " + distance +
12                           " meters away.");
13        System.out.println("The sun's mass is " + mass +
14                           " kilograms.");
15    }
16 }
```

Program Output

```
The sun is 1.495979E11 meters away.
The sun's mass is 1.989E30 kilograms.
```

The boolean Data Type

The boolean data type allows you to create variables that may hold one of two possible values: true or false. Code Listing 2-13 demonstrates the declaration and assignment of a boolean variable.

Code Listing 2-13 (TrueFalse.java)

```
1 // A program for demonstrating boolean variables
2
3 public class TrueFalse
4 {
5     public static void main(String[] args)
6     {
7         boolean bool;
8
9         bool = true;
10        System.out.println(bool);
11        bool = false;
12        System.out.println(bool);
13    }
14 }
```

Program Output

```
true
false
```

Variables of the boolean data type are useful for evaluating conditions that are either true or false. You will not be using them until Chapter 3, however, so for now just remember the following things:

- boolean variables may hold only the value true or false.
- The contents of a boolean variable may not be copied to a variable of any type other than boolean.

The char Data Type

The char data type is used to store characters. A variable of the char data type can hold one character at a time. Character literals are enclosed in *single quotation marks*. The program in Code Listing 2-14 uses a char variable. The character literals 'A' and 'B' are assigned to the variable.

Code Listing 2-14 (Letters.java)

```
1 // This program demonstrates the char data type.
2
3 public class Letters
4 {
5     public static void main(String[] args)
```

```
6    {
7        char letter;
8
9        letter = 'A';
10       System.out.println(letter);
11       letter = 'B';
12       System.out.println(letter);
13   }
14 }
```

Program Output

```
A
B
```

It is important that you do not confuse character literals with string literals, which are enclosed in double quotation marks. String literals cannot be assigned to char variables.

Unicode

Characters are internally represented by numbers. Each printable character, as well as many non-printable characters, is assigned a unique number. Java uses Unicode, which is a set of numbers that are used as codes for representing characters. Each Unicode number requires two bytes of memory, so char variables occupy two bytes. When a character is stored in memory, it is actually the numeric code that is stored. When the computer is instructed to print the value on the screen, it displays the character that corresponds with the numeric code.

You may want to refer to Appendix B, available on the book's companion Web site (at www.pearsonglobaleditions.com/Gaddis), which shows a portion of the Unicode character set. Notice that the number 65 is the code for A, 66 is the code for B, and so on. Code Listing 2-15 demonstrates that when you work with characters, you are actually working with numbers.

Code Listing 2-15 (Letters2.java)

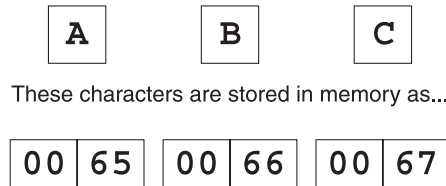
```
1 // This program demonstrates the close relationship between
2 // characters and integers.
3
4 public class Letters2
5 {
6     public static void main(String[] args)
7     {
8         char letter;
9
10        letter = 65;
11        System.out.println(letter);
12        letter = 66;
13        System.out.println(letter);
14    }
15 }
```


Program Output

A
B

Figure 2-4 illustrates that when you think of the characters A, B, and C being stored in memory, it is really the numbers 65, 66, and 67 that are stored.

Figure 2-4 Characters and how they are stored in memory



Variable Assignment and Initialization

As you have already seen in several examples, a value is put into a variable with an *assignment statement*. For example, the following statement assigns the value 12 to the variable `unitsSold`:

```
unitsSold = 12;
```

The `=` symbol is called the assignment operator. Operators perform operations on data. The data that operators work with are called operands. The assignment operator has two operands. In the statement above, the operands are `unitsSold` and 12.

In an assignment statement, the name of the variable receiving the assignment must appear on the left side of the operator, and the value being assigned must appear on the right side. The following statement is incorrect:

```
12 = unitsSold;            // ERROR!
```

The operand on the left side of the `=` operator must be a variable name. The operand on the right side of the `=` symbol must be an expression that has a value. The assignment operator takes the value of the right operand and puts it in the variable identified by the left operand. Assuming that `length` and `width` are both `int` variables, the following code illustrates that the assignment operator's right operand may be a literal or a variable:

```
length = 20;
width = length;
```

It is important to note that the assignment operator only changes the contents of its left operand. The second statement assigns the value of the `length` variable to the `width` variable. After the statement has executed, `length` still has the same value, 20.

You may also assign values to variables as part of the declaration statement. This is known as *initialization*. Code Listing 2-16 shows how it is done.

The variable declaration statement in this program is in line 7:

```
int month = 2, days = 28;
```

Code Listing 2-16 (Initialize.java)

```
1 // This program shows variable initialization.
2
3 public class Initialize
4 {
5     public static void main(String[] args)
6     {
7         int month = 2, days = 28;
8
9         System.out.println("Month " + month + " has " +
10                             days + " days.");
11     }
12 }
```

Program Output

Month 2 has 28 days.

This statement declares the month variable and initializes it with the value 2, and declares the days variable and initializes it with the value 28. As you can see, this simplifies the program and reduces the number of statements that must be typed by the programmer. Here are examples of other declaration statements that perform initialization:

```
double payRate = 25.52;
float interestRate = 12.9F;
char stockCode = 'D';
int customerNum = 459;
```

Of course, there are always variations on a theme. Java allows you to declare several variables and initialize only some of them. Here is an example of such a declaration:

```
int flightNum = 89, travelTime, departure = 10, distance;
```

The variable `flightNum` is initialized to 89 and `departure` is initialized to 10. The `travelTime` and `distance` variables remain uninitialized.



WARNING! When a variable is declared inside a method, it must have a value stored in it before it can be used. If the compiler determines that the program might be using such a variable before a value has been stored in it, an error will occur. You can avoid this type of error by initializing the variable with a value.

Variables Hold Only One Value at a Time

Remember, a variable can hold only one value at a time. When you assign a new value to a variable, the new value takes the place of the variable's previous contents. For example, look at the following code.

```
int x = 5;
System.out.println(x);
x = 99;
System.out.println(x);
```

In this code, the variable `x` is initialized with the value 5 and its contents are displayed. Then the variable is assigned the value 99. This value overwrites the value 5 that was previously stored there. The code will produce the following output:

```
5
99
```



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

2.12 Which of the following are illegal variable names and why?

```
x
99bottles
july97
theSalesFigureForFiscalYear98
r&d
grade_report
```

2.13 Is the variable name `Sales` the same as `sales`? Why or why not?

2.14 Refer to the Java primitive data types listed in Table 2-5 for this question.

- If a variable needs to hold whole numbers in the range 32 to 6,000, what primitive data type would be best?
- If a variable needs to hold whole numbers in the range -40,000 to +40,000, what primitive data type would be best?
- Which of the following literals use more memory? `22.1` or `22.1F`?

2.15 How would the number 6.31×10^{17} be represented in E notation?

2.16 A program declares a `float` variable named `number`, and the following statement causes an error. What can be done to fix the error?

```
number = 7.4;
```

2.17 What values can `boolean` variables hold?

2.18 Write statements that do the following:

- Declare a `char` variable named `letter`.
- Assign the letter A to the `letter` variable.
- Display the contents of the `letter` variable.

2.19 What are the Unicode codes for the characters 'C', 'F', and 'W'?

(You may need to refer to Appendix B on the book's companion Web site, at www.pearsonglobaleditions.com/Gaddis.)

2.20 Which is a character literal, `'B'` or `"B"`?

2.21 What is wrong with the following statement?

```
char letter = "Z";
```

2.5

Arithmetic Operators

CONCEPT: There are many operators for manipulating numeric values and performing arithmetic operations.

Java offers a multitude of operators for manipulating data. Generally, there are three types of operators: *unary*, *binary*, and *ternary*. These terms reflect the number of operands an operator requires.



VideoNote

Simple Math
Expressions

Unary operators require only a single operand. For example, consider the following expression:

```
-5
```

Of course, we understand this represents the value negative five. We can also apply the operator to a variable, as follows:

```
-number
```

This expression gives the negative of the value stored in `number`. The minus sign, when used this way, is called the *negation operator*. Because it requires only one operand, it is a unary operator.

Binary operators work with two operands. The assignment operator is in this category. Ternary operators, as you may have guessed, require three operands. Java has only one ternary operator, which is discussed in Chapter 3.

Arithmetic operations are very common in programming. Table 2-7 shows the arithmetic operators in Java.

Table 2-7 Arithmetic operators

Operator	Meaning	Type	Example
+	Addition	Binary	<code>total = cost + tax;</code>
-	Subtraction	Binary	<code>cost = total - tax;</code>
*	Multiplication	Binary	<code>tax = cost * rate;</code>
/	Division	Binary	<code>salePrice = original / 2;</code>
%	Modulus	Binary	<code>remainder = value % 3;</code>

Each of these operators works as you probably expect. The addition operator returns the sum of its two operands. Here are some example statements that use the addition operator:

```
amount = 4 + 8;           // Assigns 12 to amount
total = price + tax;      // Assigns price + tax to total
number = number + 1;     // Assigns number + 1 to number
```

The subtraction operator returns the value of its right operand subtracted from its left operand. Here are some examples:

```
temperature = 112 - 14;   // Assigns 98 to temperature
sale = price - discount;  // Assigns price - discount to sale
number = number - 1;     // Assigns number - 1 to number
```

The multiplication operator returns the product of its two operands. Here are some examples:

```
markUp = 12 * 0.25;       // Assigns 3 to markUp
commission = sales * percent; // Assigns sales * percent to commission
population = population * 2; // Assigns population * 2 to population
```

The division operator returns the quotient of its left operand divided by its right operand. Here are some examples:

```
points = 100 / 20;           // Assigns 5 to points
teams = players / maxEach;   // Assigns players / maxEach to teams
half = number / 2;           // Assigns number / 2 to half
```

The modulus operator returns the remainder of a division operation involving two integers. The following statement assigns 2 to `leftOver`:

```
leftOver = 17 % 3;
```

Situations arise where you need to get the remainder of a division. Computations that detect odd numbers or are required to determine how many items are left over after division use the modulus operator.

The program in Code Listing 2-17 demonstrates some of these operators used in a simple payroll calculation.

Code Listing 2-17 (Wages.java)

```
1 // This program calculates hourly wages plus overtime.
2
3 public class Wages
4 {
5     public static void main(String[] args)
6     {
7         double regularWages;           // The calculated regular wages.
8         double basePay = 25;           // The base pay rate.
9         double regularHours = 40;      // The hours worked less overtime.
10        double overtimeWages;          // Overtime wages
11        double overtimePay = 37.5;      // Overtime pay rate
12        double overtimeHours = 10;      // Overtime hours worked
13        double totalWages;              // Total wages
14
15        regularWages = basePay * regularHours;
16        overtimeWages = overtimePay * overtimeHours;
17        totalWages = regularWages + overtimeWages;
18        System.out.println("Wages for this week are $" +
19                            totalWages);
20    }
21 }
```

Program Output

```
Wages for this week are $1375.0
```

Code Listing 2-17 calculates the total wages an hourly paid worker earned in one week. As mentioned in the comments, there are variables for regular wages, base pay rate, regular hours worked, overtime wages, overtime pay rate, overtime hours worked, and total wages.

Line 15 in the program multiplies `basePay` times `regularHours` and stores the result, which is 1000, in `regularWages`:

```
regularWages = basePay * regularHours;
```

Line 16 multiplies `overtimePay` times `overtimeHours` and stores the result, which is 375, in `overtimeWages`:

```
overtimeWages = overtimePay * overtimeHours;
```

Line 17 adds the regular wages and the overtime wages and stores the result, 1375, in `totalWages`:

```
totalWages = regularWages + overtimeWages;
```

The `println` statement in lines 18 and 19 displays the message on the screen reporting the week's wages.

Integer Division

When both operands of the division operator are integers, the operator will perform *integer division*. This means the result of the division will be an integer as well. If there is a remainder, it will be discarded. For example, look at the following code:

```
double number;  
number = 5 / 2;
```

This code divides 5 by 2 and assigns the result to the `number` variable. What value will be stored in `number`? You would probably assume that 2.5 would be stored in `number` because that is the result your calculator shows when you divide 5 by 2; however, that is not what happens when the previous Java code is executed. Because the numbers 5 and 2 are both integers, the fractional part of the result will be thrown away, or *truncated*. As a result, the value 2 will be assigned to the `number` variable.

In the previous code, it doesn't matter that `number` is declared as a `double` because the fractional part of the result is discarded before the assignment takes place. In order for a division operation to return a floating-point value, one of the operands must be of a floating-point data type. For example, the previous code could be written as follows:

```
double number;  
number = 5.0 / 2;
```

In this code, 5.0 is treated as a floating-point number, so the division operation will return a floating-point number. The result of the division is 2.5.

Operator Precedence

It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, `x`, 21, and `y` to the variable `answer`:

```
answer = 17 + x + 21 + y;
```

Some expressions are not that straightforward, however. Consider the following statement:

```
outcome = 12 + 6 / 3;
```

What value will be stored in `outcome`? The 6 is used as an operand for both the addition and division operators. The `outcome` variable could be assigned either 6 or 14, depending on when the division takes place. The answer is 14 because the division operator has higher *precedence* than the addition operator.

Mathematical expressions are evaluated from left to right. When two operators share an operand, the operator with the highest precedence works first. Multiplication and division have higher precedence than addition and subtraction, so the statement above works like this:

- 1. 6 is divided by 3, yielding a result of 2
- 2. 12 is added to 2, yielding a result of 14

It could be diagrammed as shown in Figure 2-5.

Figure 2-5 Precedence illustrated

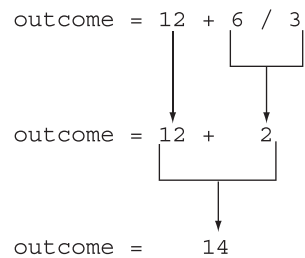


Table 2-8 shows the precedence of the arithmetic operators. The operators at the top of the table have higher precedence than the ones below them.

Table 2-8 Precedence of arithmetic operators (highest to lowest)

Highest Precedence →	- (unary negation)
	* / %
Lowest Precedence →	+ -

The multiplication, division, and modulus operators have the same precedence. The addition and subtraction operators have the same precedence. If two operators sharing an operand have the same precedence, they work according to their *associativity*. Associativity is either *left to right* or *right to left*. Table 2-9 shows the arithmetic operators and their associativity.

Table 2-9 Associativity of arithmetic operators

Operator	Associativity
- (unary negation)	Right to left
* / %	Left to right
+ -	Left to right

Table 2-10 shows some expressions and their values.

Table 2-10 Some expressions and their values

Expression	Value
$5 + 2 * 4$	13
$10 / 2 - 3$	2
$8 + 12 * 2 - 4$	28
$4 + 17 \% 2 - 1$	4
$6 - 3 * 2 + 7 - 1$	6

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the statement below, the sum of a, b, c, and d is divided by 4.0.

```
average = (a + b + c + d) / 4.0;
```

Without the parentheses, however, d would be divided by 4 and the result added to a, b, and c. Table 2-11 shows more expressions and their values.

Table 2-11 More expressions and their values

Expression	Value
$(5 + 2) * 4$	28
$10 / (5 - 3)$	5
$8 + 12 * (6 - 2)$	56
$(4 + 17) \% 2 - 1$	0
$(6 - 3) * (2 + 7) / 3$	9

In the Spotlight:

Calculating Percentages and Discounts



Determining percentages is a common calculation in computer programming. Although the % symbol is used in general mathematics to indicate a percentage, most programming languages (including Java) do not use the % symbol for this purpose. In a program, you have to convert a percentage to a floating-point number, just as you would if you were using a calculator. For example, 50 percent would be written as 0.5 and 2 percent would be written as 0.02.

Let's look at an example. Suppose you earn \$6,000 per month and you are allowed to contribute a portion of your gross monthly pay to a retirement plan. You want to determine the amount of your pay that will go into the plan if you contribute 5 percent, 8 percent, or 10 percent of your gross wages. To make this determination you write a program like the one shown in Code Listing 2-18.

Code Listing 2-18 (Contribution.java)

```
1 // This program calculates the amount of pay that
2 // will be contributed to a retirement plan if 5%,
3 // 8%, or 10% of monthly pay is withheld.
4
5 public class Contribution
6 {
7     public static void main(String[] args)
8     {
9         // Variables to hold the monthly pay and
10        // the amount of contribution.
11        double monthlyPay = 6000.0;
12        double contribution;
13
14        // Calculate and display a 5% contribution.
15        contribution = monthlyPay * 0.05;
16        System.out.println("5 percent is $" +
17                           contribution +
18                           " per month.");
19
20        // Calculate and display an 8% contribution.
21        contribution = monthlyPay * 0.08;
22        System.out.println("8 percent is $" +
23                           contribution +
24                           " per month.");
25
26        // Calculate and display a 10% contribution.
27        contribution = monthlyPay * 0.1;
28        System.out.println("10 percent is $" +
29                           contribution +
30                           " per month.");
31    }
32 }
```

Program Output

```
5 percent is $300.0 per month.
8 percent is $480.0 per month.
10 percent is $600.0 per month.
```

Lines 11 and 12 declare two variables: `monthlyPay` and `contribution`. The `monthlyPay` variable, which is initialized with the value `6000.0`, holds the amount of your monthly pay. The `contribution` variable will hold the amount of a contribution to the retirement plan.

The statements in lines 15 through 18 calculate and display 5 percent of the monthly pay. The calculation is done in line 15, where the `monthlyPay` variable is multiplied by `0.05`. The result is assigned to the `contribution` variable, which is then displayed by the statement in lines 16 through 18.

Similar steps are taken in lines 21 through 24, which calculate and display 8 percent of the monthly pay, and lines 27 through 30, which calculate and display 10 percent of the monthly pay.

Calculating a Percentage Discount

Another common calculation is determining a percentage discount. For example, suppose a retail business sells an item that is regularly priced at \$59, and is planning to have a sale where the item's price will be reduced by 20 percent. You have been asked to write a program to calculate the sale price of the item.

To determine the sale price you perform two calculations:

- First, you get the amount of the discount, which is 20 percent of the item's regular price.
- Second, you subtract the discount amount from the item's regular price. This gives you the sale price.

Code Listing 2-19 shows how this is done in Java.

Code Listing 2-19 (Discount.java)

```
1 // This program calculates the sale price of an
2 // item that is regularly priced at $59, with
3 // a 20 percent discount subtracted.
4
5 public class Discount
6 {
7     public static void main(String[] args)
8     {
9         // Variables to hold the regular price, the
10        // amount of a discount, and the sale price.
11        double regularPrice = 59.0;
12        double discount;
13        double salePrice;
14
15        // Calculate the amount of a 20% discount.
16        discount = regularPrice * 0.2;
17
18        // Calculate the sale price by subtracting
19        // the discount from the regular price.
20        salePrice = regularPrice - discount;
21
22        // Display the results.
23        System.out.println("Regular price: $" + regularPrice);
24        System.out.println("Discount amount $" + discount);
25        System.out.println("Sale price: $" + salePrice);
26    }
27 }
```

Program Output

```
Regular price: $59.0  
Discount amount $11.8  
Sale price: $47.2
```

Lines 11 through 13 declare three variables. The `regularPrice` variable holds the item's regular price, and is initialized with the value 59.0. The `discount` variable will hold the amount of the discount once it is calculated. The `salePrice` variable will hold the item's sale price.

Line 16 calculates the amount of the 20 percent discount by multiplying `regularPrice` by 0.2. The result is stored in the `discount` variable. Line 20 calculates the sale price by subtracting `discount` from `regularPrice`. The result is stored in the `salePrice` variable. The statements in lines 23 through 25 display the item's regular price, the amount of the discount, and the sale price.

The Math Class

The Java API provides a class named `Math`, which contains numerous methods that are useful for performing complex mathematical operations. In this section we will briefly look at the `Math.pow` and `Math.sqrt` methods.

The Math.pow Method

In Java, raising a number to a power requires the `Math.pow` method. Here is an example of how the `Math.pow` method is used:

```
result = Math.pow(4.0, 2.0);
```

The method takes two `double` arguments. It raises the first argument to the power of the second argument, and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

$$result = 4^2$$

Here is another example of a statement using the `Math.pow` method. It assigns 3 times 6³ to `x`:

```
x = 3 * Math.pow(6.0, 3.0);
```

And the following statement displays the value of 5 raised to the power of 4:

```
System.out.println(Math.pow(5.0, 4.0));
```

The Math.sqrt Method

The `Math.sqrt` method accepts a `double` value as its argument and returns the square root of the value. Here is an example of how the method is used:

```
result = Math.sqrt(9.0);
```

In this example the value 9.0 is passed as an argument to the `Math.sqrt` method. The method will return the square root of 9.0, which is assigned to the `result` variable. The following statement shows another example. In this statement the square root of 25.0 (which is 5.0) is displayed on the screen:

```
System.out.println(Math.sqrt(25.0));
```

For more information about the `Math` class, see Appendix G, available on the book's companion Web site at www.pearsonglobaleditions.com/Gaddis.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

2.22 Complete the following table by writing the value of each expression in the Value column.

Expression	Value
6 + 3 * 5	_____
12 / 2 - 4	_____
9 + 14 * 2 - 6	_____
5 + 19 % 3 - 1	_____
(6 + 2) * 3	_____
14 / (11 - 4)	_____
9 + 12 * (8 - 3)	_____

2.23 Is the division statement in the following code an example of integer division or floating-point division? What value will be stored in `portion`?

```
double portion;
portion = 70 / 3;
```

2.6

Combined Assignment Operators

CONCEPT: The combined assignment operators combine the assignment operator with the arithmetic operators.

Quite often, programs have assignment statements of the following form:

```
x = x + 1;
```

On the right side of the assignment operator, 1 is added to `x`. The result is then assigned to `x`, replacing the value that was previously there. Effectively, this statement adds 1 to `x`. Here is another example:

```
balance = balance + deposit;
```

Assuming that `balance` and `deposit` are variables, this statement assigns the value of `balance + deposit` to `balance`. The effect of this statement is that `deposit` is added to the value stored in `balance`. Here is another example:

```
balance = balance - withdrawal;
```

Assuming that `balance` and `withdrawal` are variables, this statement assigns the value of `balance - withdrawal` to `balance`. The effect of this statement is that `withdrawal` is subtracted from the value stored in `balance`.

If you have not seen these types of statements before, they might cause some initial confusion because the same variable name appears on both sides of the assignment operator. Table 2-12 shows other examples of statements written this way.

Table 2-12 Various assignment statements (assume `x = 6` in each statement)

Statement	What It Does	Value of <code>x</code> after the Statement
<code>x = x + 4;</code>	Adds 4 to <code>x</code>	10
<code>x = x - 3;</code>	Subtracts 3 from <code>x</code>	3
<code>x = x * 10;</code>	Multiplies <code>x</code> by 10	60
<code>x = x / 2;</code>	Divides <code>x</code> by 2	3
<code>x = x % 4</code>	Assigns the remainder of <code>x / 4</code> to <code>x</code> .	2

These types of operations are common in programming. For convenience, Java offers a special set of operators designed specifically for these jobs. Table 2-13 shows the *combined assignment operators*, also known as *compound operators*.

Table 2-13 Combined assignment operators

Operator	Example Usage	Equivalent To
<code>+=</code>	<code>x += 5;</code>	<code>x = x + 5;</code>
<code>-=</code>	<code>y -= 2;</code>	<code>y = y - 2;</code>
<code>*=</code>	<code>z *= 10;</code>	<code>z = z * 10;</code>
<code>/=</code>	<code>a /= b;</code>	<code>a = a / b;</code>
<code>%=</code>	<code>c %= 3;</code>	<code>c = c % 3;</code>

As you can see, the combined assignment operators do not require the programmer to type the variable name twice. The following statement:

```
balance = balance + deposit;
```

could be rewritten as

```
balance += deposit;
```

Similarly, the statement

```
balance = balance - withdrawal;
```

could be rewritten as

```
balance -= withdrawal;
```



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

2.24 Write statements using combined assignment operators to perform the following:

- a) Add 6 to `x`
- b) Subtract 4 from `amount`

- c) Multiply `y` by 4
- d) Divide `total` by 27
- e) Store in `x` the remainder of `x` divided by 7

2.7**Conversion between Primitive Data Types**

CONCEPT: Before a value can be stored in a variable, the value's data type must be compatible with the variable's data type. Java performs some conversions between data types automatically, but does not automatically perform any conversion that can result in the loss of data. Java also follows a set of rules when evaluating arithmetic expressions containing mixed data types.

Java is a *strongly typed* language. This means that before a value is assigned to a variable, Java checks the data types of the variable and the value being assigned to it to determine whether they are compatible. For example, look at the following statements:

```
int x;  
double y = 2.5;  
x = y;
```

The assignment statement is attempting to store a `double` value (2.5) in an `int` variable. When the Java compiler encounters this line of code, it will respond with an error message. (The JDK displays the message “possible loss of precision.”)

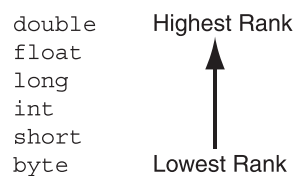
Not all assignment statements that mix data types are rejected by the compiler, however. For instance, look at the following program segment:

```
int x;  
short y = 2;  
x = y;
```

This assignment statement, which stores a `short` in an `int`, will work with no problems. So why does Java permit a `short` to be stored in an `int`, but does not permit a `double` to be stored in an `int`? The obvious reason is that a `double` can store fractional numbers and can hold values much larger than an `int` can hold. If Java were to permit a `double` to be assigned to an `int`, a loss of data would be likely.

Just like officers in the military, the primitive data types are ranked. One data type outranks another if it can hold a larger number. For example, a `float` outranks an `int`, and an `int` outranks a `short`. Figure 2-6 shows the numeric data types in order of their rank. The higher a data type appears in the list, the higher is its rank.

Figure 2-6 Primitive data type ranking



In assignment statements where values of lower-ranked data types are stored in variables of higher-ranked data types, Java automatically converts the lower-ranked value to the higher-ranked type. This is called a *widening conversion*. For example, the following code demonstrates a widening conversion, which takes place when an `int` value is stored in a `double` variable:

```
double x;  
int y = 10;  
x = y;           // Performs a widening conversion
```

A *narrowing conversion* is the conversion of a value to a lower-ranked type. For example, converting a `double` to an `int` would be a narrowing conversion. Because narrowing conversions can potentially cause a loss of data, Java does not automatically perform them.

Cast Operators

The *cast operator* lets you manually convert a value, even if it means that a narrowing conversion will take place. Cast operators are unary operators that appear as a data type name enclosed in a set of parentheses. The operator precedes the value being converted. Here is an example:

```
x = (int)number;
```

The cast operator in this statement is the word `int` inside the parentheses. It returns the value in `number`, converted to an `int`. This converted value is then stored in `x`. If `number` were a floating-point variable, such as a `float` or a `double`, the value that is returned would be *truncated*, which means the fractional part of the number is lost. The original value in the `number` variable is not changed, however.

Table 2-14 shows several statements using cast operators.

Table 2-14 Example uses of cast operators

Statement	Description
<code>littleNum = (short)bigNum;</code>	The cast operator returns the value in <code>bigNum</code> , converted to a <code>short</code> . The converted value is assigned to the variable <code>littleNum</code> .
<code>x = (long)3.7;</code>	The cast operator is applied to the expression <code>3.7</code> . The operator returns the value <code>3</code> , which is assigned to the variable <code>x</code> .
<code>number = (int)72.567;</code>	The cast operator is applied to the expression <code>72.567</code> . The operator returns <code>72</code> , which is used to initialize the variable <code>number</code> .
<code>value = (float)x;</code>	The cast operator returns the value in <code>x</code> , converted to a <code>float</code> . The converted value is assigned to the variable <code>value</code> .
<code>value = (byte)number;</code>	The cast operator returns the value in <code>number</code> , converted to a <code>byte</code> . The converted value is assigned to the variable <code>value</code> .

Note that when a cast operator is applied to a variable, it does not change the contents of the variable. It only returns the value stored in the variable, converted to the specified data type.

Recall from our earlier discussion that when both operands of a division are integers, the operation will result in integer division. This means that the result of the division will be

an integer, with any fractional part of the result thrown away. For example, look at the following code:

```
int pies = 10, people = 4;
double piesPerPerson;
piesPerPerson = pies / people;
```

Although 10 divided by 4 is 2.5, this code will store 2 in the `piesPerPerson` variable. Because both `pies` and `people` are `int` variables, the result will be an `int`, and the fractional part will be thrown away. We can modify the code with a cast operator, however, so it gives the correct result as a floating-point value:

```
piesPerPerson = (double)pies / people;
```

The variable `pies` is an `int` and holds the value 10. The expression `(double)pies` returns the value in `pies` converted to a `double`. This means that one of the division operator's operands is a `double`, so the result of the division will be a `double`. The statement could also have been written as follows:

```
piesPerPerson = pies / (double)people;
```

In this statement, the cast operator returns the value of the `people` variable converted to a `double`. In either statement, the result of the division is a `double`.



WARNING! The cast operator can be applied to an entire expression enclosed in parentheses. For example, look at the following statement:

```
piesPerPerson = (double)(pies / people);
```

This statement does not convert the value in `pies` or `people` to a `double`, but converts the result of the expression `pies / people`. If this statement were used, an integer division operation would still have been performed. Here's why: The result of the expression `pies / people` is 2 (because integer division takes place). The value 2 converted to a `double` is 2.0. To prevent the integer division from taking place, one of the operands must be converted to a `double`.

Mixed Integer Operations

One of the nuances of the Java language is the way it internally handles arithmetic operations on `int`, `byte`, and `short` variables. When values of the `byte` or `short` data types are used in arithmetic expressions, they are temporarily converted to `int` values. The result of an arithmetic operation using only a mixture of `byte`, `short`, or `int` values will always be an `int`.

For example, assume that `b` and `c` in the following expression are `short` variables:

```
b + c
```

Although both `b` and `c` are `short` variables, the result of the expression `b + c` is an `int`. This means that when the result of such an expression is stored in a variable, the variable must be an `int` or higher data type. For example, look at the following code:

```
short firstNumber = 10,
      secondNumber = 20,
      thirdNumber;
```



```
// The following statement causes an error!
thirdNumber = firstNumber + secondNumber;
```

When this code is compiled, the following statement causes an error:

```
thirdNumber = firstNumber + secondNumber;
```

The error results from the fact that `thirdNumber` is a `short`. Although `firstNumber` and `secondNumber` are also `short` variables, the expression `firstNumber + secondNumber` results in an `int` value. The program can be corrected if `thirdNumber` is declared as an `int`, or if a cast operator is used in the assignment statement, as shown here:

```
thirdNumber = (short)(firstNumber + secondNumber);
```

Other Mixed Mathematical Expressions

In situations where a mathematical expression has one or more values of the `double`, `float`, or `long` data types, Java strives to convert all of the operands in the expression to the same data type. Let's look at the specific rules that govern evaluation of these types of expressions.

1. If one of an operator's operands is a `double`, the value of the other operand will be converted to a `double`. The result of the expression will be a `double`. For example, in the following statement assume that `b` is a `double` and `c` is an `int`:

```
a = b + c;
```

The value in `c` will be converted to a `double` prior to the addition. The result of the addition will be a `double`, so the variable `a` must also be a `double`.

2. If one of an operator's operands is a `float`, the value of the other operand will be converted to a `float`. The result of the expression will be a `float`. For example, in the following statement assume that `x` is a `short` and `y` is a `float`:

```
z = x * y;
```

The value in `x` will be converted to a `float` prior to the multiplication. The result of the multiplication will be a `float`, so the variable `z` must also be either a `double` or a `float`.

3. If one of an operator's operands is a `long`, the value of the other operand will be converted to a `long`. The result of the expression will be a `long`. For example, in the following statement assume that `a` is a `long` and `b` is a `short`:

```
c = a - b;
```

The variable `b` will be converted to a `long` prior to the subtraction. The result of the subtraction will be a `long`, so the variable `c` must also be a `long`, `float`, or `double`.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

2.25 The following declaration appears in a program:

```
short totalPay, basePay = 500, bonus = 1000;
```

The following statement appears in the same program:

```
totalPay = basePay + bonus;
```

- a) Will the statement compile properly or cause an error?
- b) If the statement causes an error, why? How can you fix it?

- 2.26 The variable `a` is a `float` and the variable `b` is a `double`. Write a statement that will assign the value of `b` to `a` without causing an error when the program is compiled.

2.8

Creating Named Constants with `final`

CONCEPT: The `final` key word can be used in a variable declaration to make the variable a named constant. Named constants are initialized with a value, and that value cannot change during the execution of the program.

Assume that the following statement appears in a banking program that calculates data pertaining to loans:

```
amount = balance * 0.069;
```

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 is. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 6.9 percent to 8.2 percent? The programmer would have to search through the source code for every occurrence of the number.

Both of these problems can be addressed by using named constants. A *named constant* is a variable whose value is read only and cannot be changed during the program's execution. You can create such a variable in Java by using the `final` key word in the variable declaration. The word `final` is written just before the data type. Here is an example:

```
final double INTEREST_RATE = 0.069;
```

This statement looks just like a regular variable declaration except that the word `final` appears before the data type, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

An initialization value must be given when declaring a variable with the `final` modifier, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a `final` variable.

An advantage of using named constants is that they make programs more self-documenting. The following statement:

```
amount = balance * 0.069;
```

can be changed to read

```
amount = balance * INTEREST_RATE;
```

A new programmer can read the second statement and know what is happening. It is evident that `balance` is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization

value in the definition of the named constant is the only value that needs to be modified. If the rate increases to 8.2 percent, the declaration can be changed to the following:

```
final double INTEREST_RATE = 0.082;
```

The program is then ready to be recompiled. Every statement that uses `INTEREST_RATE` will use the new value.

The `Math.PI` Named Constant

The `Math` class, which is part of the Java API, provides a predefined named constant, `Math.PI`. This constant is assigned the value 3.14159265358979323846, which is an approximation of the mathematical value pi. For example, look at the following statement:

```
area = Math.PI * radius * radius;
```

Assuming the `radius` variable holds the radius of a circle, this statement uses the `Math.PI` constant to calculate the area of the circle.

For more information about the `Math` class, see Appendix F, available on the book's companion Web site at www.pearsonglobaleditions.com/Gaddis.

2.9 The String Class

CONCEPT: The **`String`** class allows you to create objects for holding strings. It also has various methods that allow you to work with strings.

You have already encountered strings and examined programs that display them on the screen, but let's take a moment to make sure you understand what a string is. A string is a sequence of characters. It can be used to represent any type of data that contains text, such as names, addresses, warning messages, and so forth. String literals are enclosed in double-quotation marks, such as the following:

```
"Hello World"  
"Joe Mahoney"
```

Although programs commonly encounter strings and must perform a variety of tasks with them, Java does not have a primitive data type for storing them in memory. Instead, the Java API provides a class for handling strings. You use this class to create objects that are capable of storing strings and performing operations on them. Before discussing this class, let's briefly discuss how classes and objects are related.

Objects Are Created from Classes

Chapter 1 introduced you to objects as software entities that can contain attributes and methods. An object's attributes are data values that are stored in the object. An object's methods are procedures that perform operations on the object's attributes. Before an object can be created, however, it must be designed by a programmer. The programmer determines the attributes and methods that are necessary, and then creates a class that describes the object.

You have already seen classes used as containers for applications. A class can also be used to specify the attributes and methods that a particular type of object may have. Think of a class

as a “blueprint” that objects may be created from. So a class is not an object, but a description of an object. When the program is running, it can use the class to create, in memory, as many objects as needed. Each object that is created from a class is called an *instance* of the class.



TIP: Don’t worry if these concepts seem a little fuzzy to you. As you progress through this book, the concepts of classes and objects will be reinforced again and again.

The String Class

The class that is provided by the Java API for handling strings is named `String`. The first step in using the `String` class is to declare a variable of the `String` class data type. Here is an example of a `String` variable declaration:

```
String name;
```



TIP: The `s` in `String` is written in an uppercase letter. By convention, the first character of a class name is always written in an uppercase letter.

This statement declares `name` as a `String` variable. Remember that `String` is a class, not a primitive data type. Let’s briefly look at the difference between primitive type variables and class type variables.

Primitive Type Variables and Class Type Variables

A variable of any type can be associated with an item of data. *Primitive type variables* hold the actual data items with which they are associated. For example, assume that `number` is an `int` variable. The following statement stores the value 25 in the variable:

```
number = 25;
```

This is illustrated in Figure 2-7.

Figure 2-7 A primitive type variable holds the data with which it is associated

The `number` variable holds the actual data with which it is associated.

25

A *class type variable* does not hold the actual data item that it is associated with, but holds the memory address of the data item it is associated with. If `name` is a `String` class variable, then `name` can hold the memory address of a `String` object. This is illustrated in Figure 2-8.

Figure 2-8 A `String` class variable can hold the address of a `String` object

The `name` variable can hold the address of a `String` object.

address

A `String` object



When a class type variable holds the address of an object, it is said that the variable references the object. For this reason, class type variables are commonly known as *reference variables*.

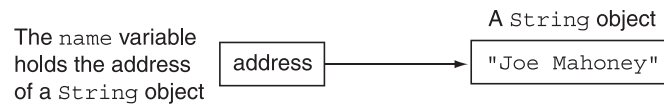
Creating a String Object

Any time you write a string literal in your program, Java will create a `String` object in memory to hold it. You can create a `String` object in memory and store its address in a string variable with a simple assignment statement. Here is an example:

```
name = "Joe Mahoney";
```

Here, the string literal causes a `String` object to be created in memory with the value “Joe Mahoney” stored in it. Then the assignment operator stores the address of that object in the `name` variable. After this statement executes, it is said that the `name` variable references a string object. This is illustrated in Figure 2-9.

Figure 2-9 The name variable holds the address of a `String` object



You can also use the `=` operator to initialize a `String` variable, as shown here:

```
String name = "Joe Mahoney";
```

This statement declares `name` as a `String` variable, creates a `String` object with the value “Joe Mahoney” stored in it, and assigns the object’s memory address to the `name` variable. Code Listing 2-20 shows `String` variables being declared, initialized, and then used in a `println` statement.

Code Listing 2-20 (StringDemo.java)

```

1 // A simple program demonstrating String objects.
2
3 public class StringDemo
4 {
5     public static void main(String[] args)
6     {
7         String greeting = "Good morning, ";
8         String name = "Herman";
9
10        System.out.println(greeting + name);
11    }
12 }
```

Program Output

```
Good morning, Herman
```

Because the `String` type is a class instead of a primitive data type, it provides numerous methods for working with strings. For example, the `String` class has a method named `length` that returns the length of the string stored in an object. Assuming the `name` variable references a `String` object, the following statement stores the length of its string in the variable `stringSize` (assume that `stringSize` is an `int` variable):

```
stringSize = name.length();
```

This statement calls the `length` method of the object that `name` refers to. To *call* a method means to execute it. The general form of a method call is as follows:

```
referenceVariable.method(arguments. . .)
```

referenceVariable is the name of a variable that references an object, *method* is the name of a method, and *arguments. . .* is zero or more arguments that are passed to the method. If no arguments are passed to the method, as is the case with the `length` method, a set of empty parentheses must follow the name of the method.

The `String` class's `length` method *returns* an `int` value. This means that the method sends an `int` value back to the statement that called it. This value can be stored in a variable, displayed on the screen, or used in calculations. Code Listing 2-21 demonstrates the `length` method.

Code Listing 2-21 (StringLength.java)

```
1 // This program demonstrates the String class's length method.
2
3 public class StringLength
4 {
5     public static void main(String[] args)
6     {
7         String name = "Herman";
8         int stringSize;
9
10        stringSize = name.length();
11        System.out.println(name + " has " + stringSize +
12                           " characters.");
13    }
14 }
```

Program Output

Herman has 6 characters.



NOTE: The `String` class's `length` method returns the number of characters in the string, including spaces.

You will study the `String` class methods in detail in Chapter 9, but let's look at a few more examples now. In addition to `length`, Table 2-15 describes the `charAt`, `toLowerCase`, and `toUpperCase` methods.

Table 2-15 A few `String` class methods

Method	Description and Example
<code>charAt(index)</code>	<p>The argument <i>index</i> is an <code>int</code> value and specifies a character position in the string. The first character is at position 0, the second character is at position 1, and so forth. The method returns the character at the specified position. The return value is of the type <code>char</code>.</p> <p>Example:</p> <pre>char letter; String name = "Herman"; letter = name.charAt(3);</pre> <p>After this code executes, the variable <code>letter</code> will hold the character ‘m’.</p>
<code>length()</code>	<p>This method returns the number of characters in the string. The return value is of the type <code>int</code>.</p> <p>Example:</p> <pre>int stringSize; String name = "Herman"; stringSize = name.length();</pre> <p>After this code executes, the <code>stringSize</code> variable will hold the value 6.</p>
<code>toLowerCase()</code>	<p>This method returns a new string that is the lowercase equivalent of the string contained in the calling object.</p> <p>Example:</p> <pre>String bigName = "HERMAN"; String littleName = bigName.toLowerCase();</pre> <p>After this code executes, the object referenced by <code>littleName</code> will hold the string “herman”.</p>
<code>toUpperCase()</code>	<p>This method returns a new string that is the uppercase equivalent of the string contained in the calling object.</p> <p>Example:</p> <pre>String littleName = "herman"; String bigName = littleName.toUpperCase();</pre> <p>After this code executes, the object referenced by <code>bigName</code> will hold the string “HERMAN”.</p>

The program in Code Listing 2-22 demonstrates these methods.

Code Listing 2-22 (`StringMethods.java`)

```
1 // This program demonstrates a few of the String methods.
2
3 public class StringMethods
4 {
5     public static void main(String[] args)
6     {
7         String message = "Java is Great Fun!";
8         String upper = message.toUpperCase();
```

```

 9      String lower = message.toLowerCase();
10      char letter = message.charAt(2);
11      int stringSize = message.length();
12
13      System.out.println(message);
14      System.out.println(upper);
15      System.out.println(lower);
16      System.out.println(letter);
17      System.out.println(stringSize);
18  }
19  }

```

Program Output

```

Java is Great Fun!
JAVA IS GREAT FUN!
java is great fun!
v
18

```



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 2.27 Write a statement that declares a `String` variable named `city`. The variable should be initialized so it references an object with the string “San Francisco”.
- 2.28 Assume that `stringLength` is an `int` variable. Write a statement that stores the length of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `stringLength`.
- 2.29 Assume that `oneChar` is a `char` variable. Write a statement that stores the first character in the string referenced by the `city` variable (declared in Checkpoint 2.27) in `oneChar`.
- 2.30 Assume that `upperCity` is a `String` reference variable. Write a statement that stores the uppercase equivalent of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `upperCity`.
- 2.31 Assume that `lowerCity` is a `String` reference variable. Write a statement that stores the lowercase equivalent of the string referenced by the `city` variable (declared in Checkpoint 2.27) in `lowerCity`.

2.10 Scope

CONCEPT: A variable’s scope is the part of the program that has access to the variable.

Every variable has a *scope*. The scope of a variable is the part of the program where the variable may be accessed by its name. A variable is visible only to statements inside the variable’s scope. The rules that define a variable’s scope are complex, and you are only

introduced to the concept here. In other chapters of the book we revisit this topic and expand on it.

So far, you have only seen variables declared inside the `main` method. Variables that are declared inside a method are called *local variables*. Later you will learn about variables that are declared outside a method, but for now, let's focus on the use of local variables.

A local variable's scope begins at the variable's declaration and ends at the end of the method in which the variable is declared. The variable cannot be accessed by statements that are outside this region. This means that a local variable cannot be accessed by code that is outside the method, or inside the method but before the variable's declaration. The program in Code Listing 2-23 shows an example.

Code Listing 2-23 (Scope.java)

```

1 // This program can't find its variable.
2
3 public class Scope
4 {
5     public static void main(String[] args)
6     {
7         System.out.println(value); // ERROR!
8         int value = 100;
9     }
10 }
```

The program does not compile because it attempts to send the contents of the variable `value` to `println` before the variable is declared. It is important to remember that the compiler reads your program from top to bottom. If it encounters a statement that uses a variable before the variable is declared, an error will result. To correct the program, the variable declaration must be written before any statement that uses it.



NOTE: If you compile this program, the compiler will display an error message such as “cannot resolve symbol.” This means that the compiler has encountered a name for which it cannot determine a meaning.

Another rule that you must remember about local variables is that you cannot have two local variables with the same name in the same scope. For example, look at the following method.

```

public static void main(String[] args)
{
    // Declare a variable named number and
    // display its value.
    int number = 7;
    System.out.println(number);
}
```

```
// Declare another variable named number and
// display its value.
int number = 100;           // ERROR!!!
System.out.println(number); // ERROR!!!
}
```

This method declares a variable named `number` and initializes it with the value 7. The variable's scope begins at the declaration statement and extends to the end of the method. Inside the variable's scope a statement appears that declares another variable named `number`. This statement will cause an error because you cannot have two local variables with the same name in the same scope.

2.11 Comments

CONCEPT: Comments are notes of explanation that document lines or sections of a program. Comments are part of the program, but the compiler ignores them. They are intended for people who may be reading the source code.

Comments are short notes that are placed in different parts of a program, explaining how those parts of the program work. Comments are not intended for the compiler. They are intended for programmers to read, to help them understand the code. The compiler skips all of the comments that appear in a program.

As a beginning programmer, you might resist the idea of writing a lot of comments in your programs. After all, it's a lot more fun to write code that actually does something! However, it's crucial that you take the extra time to write comments. They will almost certainly save you time in the future when you have to modify or debug the program. Even large and complex programs can be made easy to read and understand if they are properly commented.

In Java there are three types of comments: single-line comments, multiline comments, and documentation comments. Let's briefly discuss each type.

Single-Line Comments

You have already seen the first way to write comments in a Java program. You simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Code Listing 2-24 shows that comments may be placed liberally throughout a program.

Code Listing 2-24 (Comment1.java)

```
1 // PROGRAM: Comment1.java
2 // Written by Herbert Dorfmann
3 // This program calculates company payroll
4
5 public class Comment1
```

```

6  {
7      public static void main(String[] args)
8      {
9          double payRate;           // Holds the hourly pay rate
10         double hours;             // Holds the hours worked
11         int employeeNumber;       // Holds the employee number
12
13         // The Remainder of This Program is Omitted.
14     }
15 }

```

In addition to telling who wrote the program and describing the purpose of variables, comments can also be used to explain complex procedures in your code.

Multi-Line Comments

The second type of comment in Java is the multi-line comment. *Multi-line comments* start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Code Listing 2-25 illustrates how multi-line comments may be used.

Code Listing 2-25 (Comment2.java)

```

1  /*
2      PROGRAM: Comment2.java
3      Written by Herbert Dorfmann
4      This program calculates company payroll
5  */
6
7  public class Comment2
8  {
9      public static void main(String[] args)
10     {
11         double payRate;           // Holds the hourly pay rate
12         double hours;             // Holds the hours worked
13         int employeeNumber;       // Holds the employee number
14
15         // The Remainder of This Program is Omitted.
16     }
17 }

```

Unlike a comment started with `//`, a multi-line comment can span several lines. This makes it more convenient to write large blocks of comments because you do not have to

mark every line. Consequently, the multi-line comment is inconvenient for writing single-line comments because you must type both a beginning and an ending comment symbol.

Remember the following advice when using multi-line comments:

- Be careful not to reverse the beginning symbol with the ending symbol.
- Be sure not to forget the ending symbol.

Many programmers use asterisks or other characters to draw borders or boxes around their comments. This helps to visually separate the comments from surrounding code. These are called block comments. Table 2-16 shows four examples of block comments.

Table 2-16 Block comments

<code>/*</code>	<code>//*****</code>
<code>* This program demonstrates the</code>	<code>// This program demonstrates the *</code>
<code>* way to write comments.</code>	<code>// way to write comments. *</code>
<code>*/</code>	<code>//*****</code>
<code>////////////////////</code>	<code>//-----</code>
<code>// This program demonstrates the</code>	<code>// This program demonstrates the</code>
<code>// way to write comments.</code>	<code>// way to write comments.</code>
<code>////////////////////</code>	<code>//-----</code>

Documentation Comments

The third type of comment is known as a documentation comment. *Documentation comments* can be read and processed by a program named `javadoc`, which comes with the JDK. The purpose of the `javadoc` program is to read Java source code files and generate attractively formatted HTML files that document the source code. If the source code files contain any documentation comments, the information in the comments becomes part of the HTML documentation. The HTML documentation files may be viewed in a Web browser.

Any comment that starts with `/**` and ends with `*/` is considered a documentation comment. Normally you write a documentation comment just before a class header, giving a brief description of the class. You also write a documentation comment just before each method header, giving a brief description of the method. For example, Code Listing 2-26 shows a program with documentation comments. This program has a documentation comment just before the class header, and just before the main method header.

Code Listing 2-26 (Comment3.java)

```

1  /**
2   This class creates a program that calculates company payroll.
3  */
4
5  public class Comment3
6  {
7      /**
8       The main method is the program's starting point.
9      */
10
11     public static void main(String[] args)
12     {
13         double payRate;        // Holds the hourly pay rate
14         double hours;          // Holds the hours worked
15         int employeeNumber;    // Holds the employee number
16
17         // The Remainder of This Program is Omitted.
18     }
19 }

```

You run the javadoc program from the operating system command prompt. Here is the general format of the javadoc command:

```
javadoc SourceFile.java
```

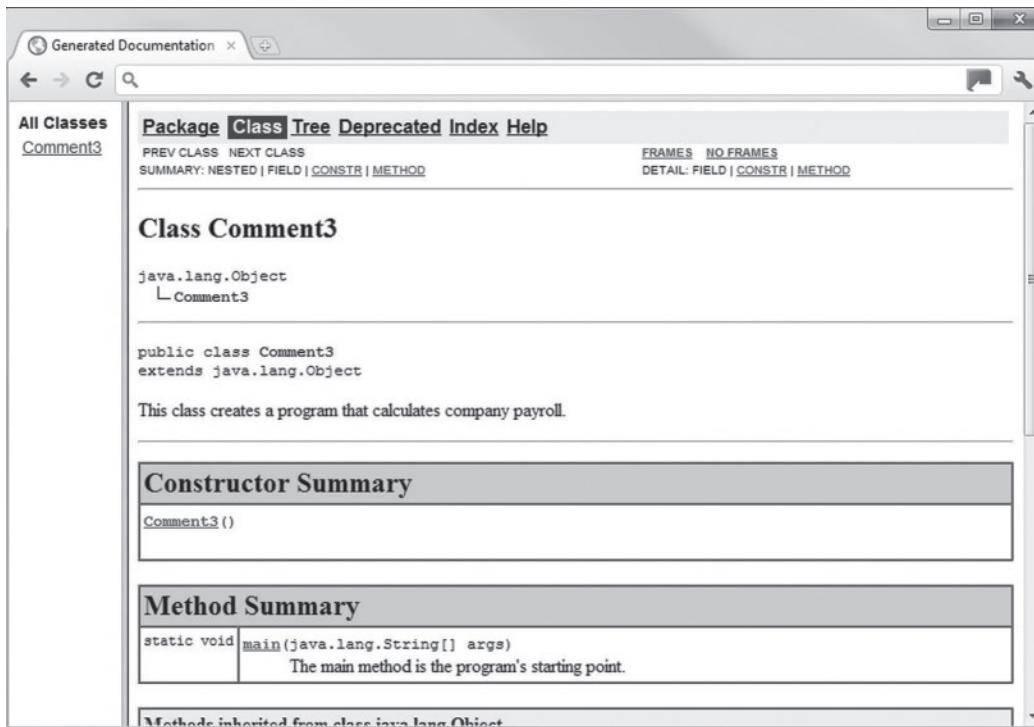
SourceFile.java is the name of a Java source code file, including the .java extension. The file will be read by javadoc and documentation will be produced for it. For example, the following command will produce documentation for the *Comment3.java* source code file, which is shown in Code Listing 2-26:

```
javadoc Comment3.java
```

After this command executes, several documentation files will be created in the same directory as the source code file. One of these files will be named *index.html*. Figure 2-10 shows the *index.html* file being viewed in a Web browser. Notice that the text that was written in the documentation comments appears in the file.



TIP: When you write a documentation comment for a method, the HTML documentation file that is produced by javadoc will have two sections for the method: a summary section and a detail section. The first sentence in the method's documentation comment is used as the summary of the method. Note that javadoc considers the end of the sentence as a period followed by a whitespace character. For this reason, when a method description contains more than one sentence, you should always end the first sentence with a period followed by a whitespace character. The method's detail section will contain all of the description that appears in the documentation comment.

Figure 2-10 Documentation generated by javadoc (Google Inc.)

If you look at the JDK documentation, which are HTML files that you view in a Web browser, you will see that they are formatted in the same way as the files generated by javadoc. A benefit of using javadoc to document your source code is that your documentation will have the same professional look and feel as the standard Java documentation.

From this point forward in the book, we will use documentation comments in the example source code. As we progress through various topics, you will see additional uses of documentation comments and the javadoc program.



Checkpoint

MyProgrammingLab™ www.myprogramminglab.com

- 2.32 How do you write a single line comment? How do you write a multi-line comment? How do you write a documentation comment?
- 2.33 How are documentation comments different from other types of comments?

2.12 Programming Style

CONCEPT: Programming style refers to the way a programmer uses spaces, indentations, blank lines, and punctuation characters to visually arrange a program's source code.

In Chapter 1, you learned that syntax rules govern the way a language may be used. The syntax rules of Java dictate how and where to place key words, semicolons, commas, braces, and other elements of the language. The compiler checks for syntax errors, and if there are none, generates byte code.

When the compiler reads a program it processes it as one long stream of characters. The compiler doesn't care that each statement is on a separate line, or that spaces separate operators from operands. Humans, on the other hand, find it difficult to read programs that aren't written in a visually pleasing manner. Consider Code Listing 2-27 for example.

Code Listing 2-27 (Compact.java)

```
1 public class Compact {public static void main(String [] args){int
2 shares=220; double averagePrice=14.67; System.out.println(
3 "There were "+shares+" shares sold at $" +averagePrice+
4 " per share.");//}}
```

Program Output

There were 220 shares sold at \$14.67 per share.

Although the program is syntactically correct (it doesn't violate any rules of Java), it is very difficult to read. The same program is shown in Code Listing 2-28, written in a more understandable style.

Code Listing 2-28 (Readable.java)

```

1  /**
2      This example is much more readable than Compact.java.
3  */
4
5  public class Readable
6  {
7      public static void main(String[] args)
8      {
9          int shares = 220;
10         double averagePrice = 14.67;
11
12         System.out.println("There were " + shares +
13                             " shares sold at $" +
14                             averagePrice + " per share.");

```

```

15     }
16 }

```

Program Output

There were 220 shares sold at \$14.67 per share.

The term *programming style* usually refers to the way source code is visually arranged. It includes techniques for consistently putting spaces and indentations in a program so visual cues are created. These cues quickly tell a programmer important information about a program.

For example, notice in Code Listing 2-28 that inside the class's braces each line is indented, and inside the main method's braces each line is indented again. It is a common programming style to indent all the lines inside a set of braces, as shown in Figure 2-11.

Figure 2-11 Indentation

```

/**
 * This example is much more readable than Compact.java.
 */

public class Readable
{
    public static void main(String[] args)
    {
        int shares = 220;
        double averagePrice = 14.67;

        System.out.println("There were " + shares +
            " shares sold at $" +
            averagePrice + " per share.");
    }
}

```

Another aspect of programming style is how to handle statements that are too long to fit on one line. Notice that the `println` statement is spread out over three lines. Extra spaces are inserted at the beginning of the statement's second and third lines, which indicate that they are continuations.

When declaring multiple variables of the same type with a single statement, it is a common practice to write each variable name on a separate line with a comment explaining the variable's purpose. Here is an example:

```

int fahrenheit,    // To hold the Fahrenheit temperature
    celsius,       // To hold the Celsius temperature
    kelvin;        // To hold the Kelvin temperature

```

You may have noticed in the example programs that a blank line is inserted between the variable declarations and the statements that follow them. This is intended to separate the declarations visually from the executable statements.

There are many other issues related to programming style. They will be presented throughout the book.

2.13 Reading Keyboard Input

CONCEPT: Objects of the **Scanner** class can be used to read input from the keyboard.

Previously we discussed the `System.out` object, and how it refers to the standard output device. The Java API has another object, `System.in`, which refers to the standard input device. The *standard input device* is normally the keyboard. You can use the `System.in` object to read keystrokes that have been typed at the keyboard. However, using `System.in` is not as simple and straightforward as using `System.out` because the `System.in` object reads input only as byte values. This isn't very useful because programs normally require values of other data types as input. To work around this, you can use the `System.in` object in conjunction with an object of the `Scanner` class. The `Scanner` class is designed to read input from a source (such as `System.in`), and it, provides methods that you can use to retrieve the input formatted as primitive values or strings.

First, you create a `Scanner` object and connect it to the `System.in` object. Here is an example of a statement that does just that:

```
Scanner keyboard = new Scanner(System.in);
```

Let's dissect the statement into two parts. The first part of the statement,

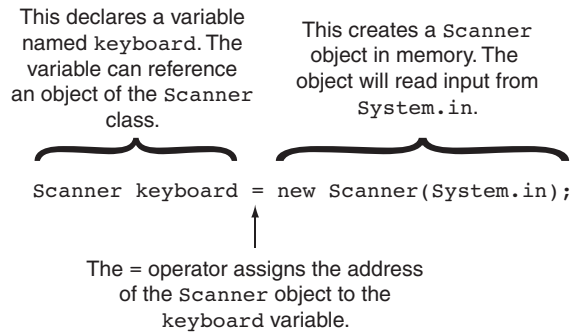
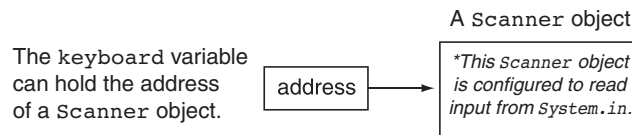
```
Scanner keyboard
```

declares a variable named `keyboard`. The data type of the variable is `Scanner`. Because `Scanner` is a class, the `keyboard` variable is a class type variable. Recall from our discussion on `String` objects that a class type variable holds the memory address of an object. Therefore, the `keyboard` variable will be used to hold the address of a `Scanner` object. The second part of the statement is as follows:

```
= new Scanner(System.in);
```

The first thing we see in this part of the statement is the assignment operator (`=`). The assignment operator will assign something to the `keyboard` variable. After the assignment operator we see the word `new`, which is a Java key word. The purpose of the `new` key word is to create an object in memory. The type of object that will be created is listed next. In this case, we see `Scanner(System.in)` listed after the `new` key word. This specifies that a `Scanner` object should be created, and it should be connected to the `System.in` object. The memory address of the object is assigned (by the `=` operator) to the variable `keyboard`. After the statement executes, the `keyboard` variable will reference the `Scanner` object that was created in memory.

Figure 2-12 points out the purpose of each part of this statement. Figure 2-13 illustrates how the `keyboard` variable references an object of the `Scanner` class.

Figure 2-12 The parts of the statement**Figure 2-13** The keyboard variable references a Scanner object

NOTE: In the preceding code, we chose keyboard as the variable name. There is nothing special about the name keyboard. We simply chose that name because we will use the variable to read input from the keyboard.

The Scanner class has methods for reading strings, bytes, integers, long integers, short integers, floats, and doubles. For example, the following code uses an object of the Scanner class to read an int value from the keyboard and assign the value to the number variable.

```
int number;
Scanner keyboard = new Scanner(System.in);
System.out.print("Enter an integer value: ");
number = keyboard.nextInt();
```

The last statement shown here calls the Scanner class's nextInt method. The nextInt method formats an input value as an int, and then returns that value. Therefore, this statement formats the input that was entered at the keyboard as an int, and then returns it. The value is assigned to the number variable.

Table 2-17 lists several of the Scanner class's methods and describes their use.

Table 2-17 Some of the `Scanner` class methods

Method	Example and Description
<code>nextByte</code>	<p>Example Usage:</p> <pre>byte x; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a byte value: "); x = keyboard.nextByte();</pre> <p>Description: Returns input as a byte.</p>
<code>nextDouble</code>	<p>Example Usage:</p> <pre>double number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a double value: "); number = keyboard.nextDouble();</pre> <p>Description: Returns input as a double.</p>
<code>nextFloat</code>	<p>Example Usage:</p> <pre>float number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a float value: "); number = keyboard.nextFloat();</pre> <p>Description: Returns input as a float.</p>
<code>nextInt</code>	<p>Example Usage:</p> <pre>int number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter an integer value: "); number = keyboard.nextInt();</pre> <p>Description: Returns input as an int.</p>
<code>nextLine</code>	<p>Example Usage:</p> <pre>String name; Scanner keyboard = new Scanner(System.in); System.out.print("Enter your name: "); name = keyboard.nextLine();</pre> <p>Description: Returns input as a String.</p>
<code>nextLong</code>	<p>Example Usage:</p> <pre>long number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a long value: "); number = keyboard.nextLong();</pre> <p>Description: Returns input as a long.</p>
<code>nextShort</code>	<p>Example Usage:</p> <pre>short number; Scanner keyboard = new Scanner(System.in); System.out.print("Enter a short value: "); number = keyboard.nextShort();</pre> <p>Description: Returns input as a short.</p>